UNIÓN INTERNACIONAL DE TELECOMUNICACIONES

UIT-T Z.140

(07/2001)

SECTOR DE NORMALIZACIÓN
DE LAS TELECOMUNICACIONES
DE LA UIT

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

Técnicas de descripción formal

Notación combinada arborescente y tabular versión 3: Lenguaje núcleo

Recomendación UIT-T Z.140

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 alto nivel 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.140

Notación combinada arborescente y tabular versión 3: Lenguaje núcleo

Resumen

Esta Recomendación define el lenguaje núcleo de la versión 3 de la notación combinada arborescente y tabular (TTCN-3). Esta notación se puede utilizar para la especificación de todos los tipos de pruebas de sistemas reactivos en diversos puertos de comunicación. Las aplicaciones típicas de la TTCN-3 son las pruebas de protocolos (incluidos los protocolos móviles e Internet) y las pruebas de servicios (incluidos los servicios suplementarios), las pruebas de módulos, las pruebas de plataformas basadas en la arquitectura de intermediario de petición de objeto común (CORBA), las pruebas de interfaces de programas de aplicación (API), etc. La TTCN-3 no está restringida a las pruebas de conformidad: se puede utilizar para muchas otras clases de pruebas, a saber, pruebas de interoperabilidad, robustez, regresión, sistemas e integración. La especificación de series de pruebas para protocolos de capa física está fuera del ámbito de esta Recomendación.

Orígenes

La Recomendación UIT-T Z.140, preparada por la Comisión de Estudio 10 (2001-2004) del UIT-T, fue aprobada por el procedimiento de la Resolución 1 de la AMNT el 22 de julio de 2001.

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.

ÍNDICE

Página

Página

Página

Página

Página

Recomendación UIT-T Z.140

Notación combinada arborescente y tabular versión 3: Lenguaje núcleo

1 Alcance

La presente Recomendación define el lenguaje núcleo de la versión 3 de la notación combinada arborescente y tabular (o TTCN-3). Esta notación se puede utilizar para la especificación de todos los tipos de pruebas de sistemas en una variedad de puertos de comunicación. Las aplicaciones típicas de TTCN-3 son la prueba de protocolos (incluidos los protocolos móviles e Internet), la prueba de servicios (incluidos los servicios suplementarios), la prueba de módulos, la prueba de plataformas basadas en CORBA, las interfaces de programación de aplicación, etc. TTCN-3 no está restringida a la prueba de conformidad y se puede utilizar para muchas otras clases de pruebas, a saber, prueba de interoperabilidad, robustez, regresión, sistemas e integración. La especificación de sucesión de prueba para protocolos de capa física está fuera del ámbito de la presente Recomendación.

Se prevé utilizar TTCN-3 para la especificación de sucesión de pruebas que son independientes de los métodos de prueba, capas y protocolos. Se definen varios formatos de presentación para TTCN-3: un formato de presentación tabular [1] y un formato de presentación gráfica [2]. La especificación de estos formatos está fuera del ámbito de la presente Recomendación.

La presente Recomendación define una manera normativa de utilizar la notación de sintaxis abstracta Uno definida en las Recomendaciones de la serie UIT-T X.680 [7], [8], [9] y [10] con TTCN-3. La armonización de otros lenguajes con TTCN-3 está fuera del ámbito de la presente Recomendación.

Aunque en el diseño de TTCN-3 se ha tomado en consideración la implementación de traductores y compiladores TTCN-3, los medios de realización de sucesión de pruebas ejecutables (ETS, *executable test suites*) a partir de sucesión de pruebas abstractas (ATS, *abstract test suites*) está fuera del ámbito de la presente Recomendación.

2 Referencias

Los siguientes documentos contienen disposiciones que, mediante la referencia hecha en este texto, constituyen disposiciones de la presente Recomendación:

Las referencias son específicas (identificadas por la fecha de publicación y/o el número de edición o número de versión) o no específicas.

– Para una referencia específica, no se aplican las revisiones subsiguientes.

– Para una referencia no específica, se aplica la última versión.

[1] Recomendación UIT-T Z.141 (2001), *Notación combinada arborescente y tabular – Versión 3 (TTCN-3): Formato de presentación tabular*.

[2] Recomendación UIT-T Z.142 (Proyecto), *Notación combinada arborescente y tabular – Versión 3 (TTCN-3): Formato gráfico*.

[3] Recomendación UIT-T X.290 (1995), *Metodología y marco de las pruebas de conformidad de interconexión de sistemas abiertos de las Recomendaciones sobre los protocolos para aplicaciones del UIT-T – Conceptos generales*.

ISO/CEI 9646-1:1994, Information technology – *Open Systems Interconnection – Conformance testing methodology and framework – Part 1: General concepts*.

[4] Recomendación UIT-T X.292 (1998), *Metodología y marco de las pruebas de conformidad para interconexión de sistemas abiertos de las Recomendaciones sobre los protocolos para aplicaciones del UIT-T – Notación combinada arborescente y tabular*.

ISO/CEI 9646-3:1998, *Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 3: The Tree and Tabular Combined Notation (TTCN)*.

[5] Recomendación UIT-T T.50 (1992), *Alfabeto internacional de referencia (anteriormente Alfabeto internacional N.º 5 o 1A5) – Tecnología de la información – Juego de caracteres codificado de siete bits para intercambio de información*.

Norma ISO/CEI 646:1991, *Information technology – ISO 7-bit coded character set for information exchange*.

[6] ISO/CEI 10646-1:1993, *Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane*.

[7] Recomendación UIT-T X.680 (1997) | ISO/CEI 8824-1:1998, *Tecnología de la información – Notación de sintaxis abstracta uno: Especificación de la notación básica*.

[8] Recomendación UIT-T X.681 (1997) | ISO/CEI 8824-2:1998, *Tecnología de la información – Notación de sintaxis abstracta uno: Especificación de objetos de información*.

[9] Recomendación UIT-T X.682 (1997) | ISO/CEI 8824-3:1998, *Tecnología de la información – Notación de sintaxis abstracta uno: Especificación de constricciones*.

[10] Recomendación UIT-T X.683 (1997) | ISO/CEI 8824-4:1998, *Tecnología de la información – Notación de sintaxis abstracta uno: Parametrización de especificaciones de notación de sintaxis abstracta uno*.

[11] Recomendación UIT-T X.690 (1997) | ISO/CEI 8825-1:1998, *Tecnología de la información – Reglas de codificación de notación de sintaxis abstracta uno: Especificación de las reglas de codificación básica, de las reglas de codificación canónica y de las reglas de codificación distinguida*.

[12] Recomendación UIT-T X.691 (1997) | ISO/CEI 8825-2:1998, *Tecnología de la información – Reglas de codificación de notación de sintaxis abstracta uno: Especificación de reglas de codificación compactada*.

3 Definiciones y abreviaturas

3.1 Definiciones

En esta Recomendación se definen los términos siguientes.

3.1.1 tipo compatible: TTCN-3 no está muy tipificada, pero el lenguaje requiere compatibilidad de tipos. Las variables, constantes, plantillas, etc., tienen tipos compatibles si se resuelven en el mismo tipo básico y, en el caso de asignaciones, concordancia, etc., no se viola la subtipificación (por ejemplo, gamas, restricciones de longitud).

3.1.2 puerto de comunicación: Mecanismo abstracto que facilita la comunicación entre componentes de prueba. Un puerto de comunicación es modelado como una cola FIFO (primero en entrar primero en salir) en el sentido receptor. Los puertos se pueden basar en mensaje, en procedimiento o en una mezcla de ambos.

3.1.3 excepción: En casos de comunicación síncrona, una excepción (si se define) es planteada por una entidad respondedora si no puede responder a una llamada de procedimiento distante con la respuesta prevista normal.

3.1.4 sucesión de pruebas: Módulo TTCN-3 que especifica completamente de manera explícita o implícita a través de enunciados de importación todas las definiciones y descripciones de comportamiento necesarias para definir un conjunto completo de casos de prueba.

3.1.5 interfaz de sistema de prueba: Componente de prueba que proporciona una correspondencia de los puertos disponibles en el sistema de prueba TTCN-3 (abstracto) con los ofrecidos por un sistema de prueba real.

3.1.6 parametrización de tipos: Capacidad de introducir un tipo como un parámetro real en un objeto parametrizado. Este parámetro de tipo real completa la especificación de tipo de ese objeto. Obsérvese que el parámetro no es un valor de un tipo, sino el propio tipo.

3.2 Definiciones de la Rec. UIT-T X.290 e ISO/CEI 9646-1

A los efectos de la presente Recomendación, se aplican los siguientes términos y definiciones de la Rec. UIT-T X.290 e ISO/CEI-9646-1 [3]:

- Declaración de conformidad de implementación (ICS, *implementation conformance statement*).
- Información suplementaria de implementación para pruebas (IXIT, *implementation eXtra information for testing*).
- Implementación sometida a prueba (IUT, *implementation under test*).
- Sistema sometido a prueba (SUT, *system under test*).
- Caso de prueba.
- Error de caso de prueba.
- Sistema de prueba.

3.3 Definiciones de la Rec. UIT-T X.292 e ISO/CEI 9646-3

A los efectos de la presente Recomendación, se aplican los siguientes términos y definiciones de la Rec. UIT-T X.292 e ISO/CEI 9646-3 [4]:

- Componente de prueba principal (MTC, *main test component*).
- Componente de prueba paralelo (PTC, *parallel test component*).
- Semántica instantánea.

3.4 Abreviaturas

En esta Recomendación se utilizan las siguientes siglas.

API Interfaz de programación de aplicación (*application programming interface*)

ASN.1 Notación de sintaxis abstracta uno (*abstract syntax notation one*)

ASP Primitiva de servicio abstracta (*abstract service primitive*)

ATS Sucesión de pruebas abstractas (*abstract test suite*)

BNF Forma Backus-Nauer (*backus-nauer form*)

CORBA Arquitectura de intermediario de petición de objeto común (*common object request broker architecture*)

ETS Sucesión de pruebas ejecutables (*executable test suite*)

FIFO Primero en entrar, primero en salir (*first in first out*)

IDL Lenguaje de descripción de interfaz (*interface description language*)

IUT Implementación sometida a prueba (*implementation under test*)

MTC Componente de prueba principal (*master test component*)

PDU Unidad de datos de protocolo (*protocol data unit*)

PICS Declaración de conformidad de implementación de protocolo (*protocol implementation conformance statement*)

PIXIT Información suplementaria de implementación de protocolo para pruebas (*protocol implementation eXtra information for testing*)

PTC Componente de prueba paralelo (*parallel test component*)

SUT Sistema sometido a prueba (*system under test*)

TTCN Notación combinada arborescente y tabular (*tree and tabular combined notation*)

4 Introducción

TTCN-3 es un lenguaje flexible y poderoso aplicable a la especificación de todos los tipos de pruebas de sistemas reactivas en una variedad de interfaces de comunicación. Las aplicaciones típicas son la prueba de protocolos (incluidos los protocolos móvil e Internet), prueba de servicios (incluidos los servicios suplementarios), prueba de módulos, prueba de plataformas basadas en CORBA, prueba de API, etc. TTCN-3 no está restringida a la prueba de conformidad y se puede utilizar para muchas otras clases de pruebas, a saber, prueba de interoperabilidad, robustez, regresión, sistemas e integración.

Desde el punto de vista sintáctico, TTCN-3 es muy diferente de las versiones anteriores del lenguaje definidas en la Rec. UIT-T X.292 e ISO/CEI 9646-3 [4]. Sin embargo, se ha mantenido, y en algunos casos se ha mejorado, gran parte de la funcionalidad básica probada de TTCN. Las características esenciales de TTCN-3 son:

- la capacidad de especificar configuraciones dinámicas de pruebas concurrentes;
- operaciones para comunicación síncrona y asíncrona;
- la capacidad de especificar información de codificación y otros atributos (incluida extensibilidad por el usuario);
- la capacidad de especificar plantillas de datos y de firmas con poderosos mecanismos de concordancia;
- la parametrización de tipos y valores;
- la asignación y el tratamiento de veredictos de prueba;
- mecanismos de parametrización de sucesión de pruebas y selección de casos de prueba;
- el uso combinado de TTCN-3 con ASN.1 (y posible utilización con otros lenguajes, tal como IDL);
- sintaxis, formato de intercambio y semántica estática bien definidos;
- diferentes formatos de presentación (por ejemplo, formatos de presentación tabular y gráfica);
- un algoritmo de ejecución preciso (semántica operacional).

4.1 El lenguaje núcleo y formatos de presentación

Históricamente, TTCN ha estado asociada siempre con la prueba de conformidad. Para abrir el lenguaje a una gama más amplia de aplicaciones de prueba, tanto en el dominio de normalización como en el dominio industrial, en la presente Recomendación se separa la especificación de TTCN-3 en varias partes. La primera parte, definida en la presente Recomendación es el lenguaje núcleo. La segunda parte, definida en la Rec. UIT-T Z.141 [1], es el formato de presentación

tabular, similar en apariencia y funcionalidad a las versiones anteriores de TTCN. La tercera parte, definida en la Rec. UIT-T Z.142 [2], es el formato de presentación gráfica.

El lenguaje núcleo satisface tres fines:

a) un lenguaje de prueba basado en texto generalizado por su propio derecho;

b) un formato de intercambio normalizado de sucesión de pruebas TTCN entre herramientas TTCN;

c) como la base semántica (y cuando procede, la base sintáctica) para diversos formatos de presentación.

El lenguaje núcleo se puede utilizar independientemente de los formatos de presentación. Sin embargo, el formato tabular y el formato gráfico no pueden ser utilizados sin el lenguaje núcleo. El uso y la implementación de estos formatos de presentación se basarán en el lenguaje núcleo.

El formato tabular y el formato gráfico son los primeros de un conjunto previsto de formatos de presentación diferentes. Estos otros formatos pueden ser los formatos de presentación normalizados o pueden ser formatos de presentación patentados definidos por los propios usuarios de TTCN-3. Estos formatos adicionales no se definen en la presente Recomendación.

TTCN-3 está totalmente armonizada con la ASN.1 que puede ser utilizada facultativamente con módulos TTCN-3 como una sintaxis alternativa de tipos y de valores de datos. La utilización de ASN.1 en módulos TTCN-3 se define en el anexo E. Se podrá aplicar el método utilizado para combinar ASN.1 y TTCN-3 con el fin de soportar el uso de otros sistemas de tipos y valores con TTCN-3, pero estos detalles no se definen en la presente Recomendación.

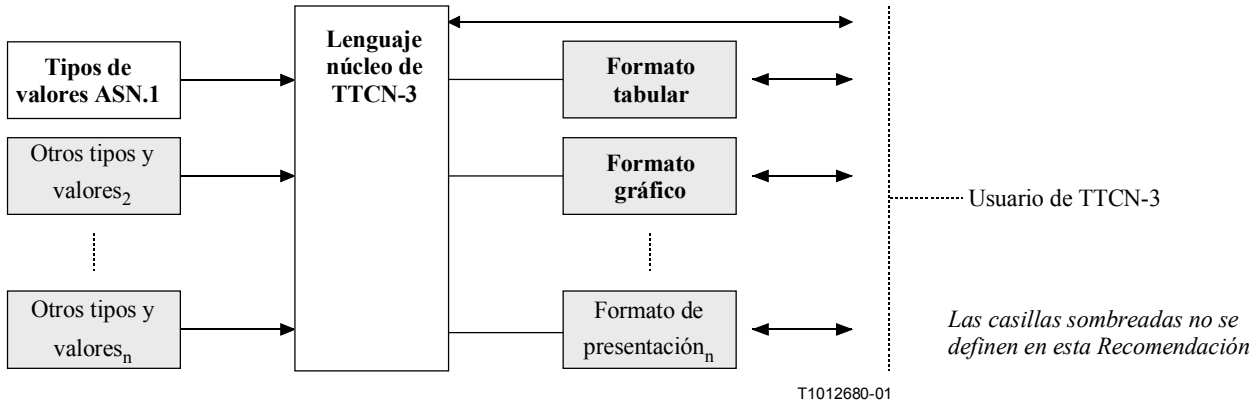


Figura 1/Z.140 – Visión del usuario del lenguaje núcleo y de los diversos formatos de presentación

El lenguaje núcleo se define mediante una sintaxis completa (véase el anexo A) y la semántica operacional (véase el anexo B). Contiene la semántica estática mínima (proporcionada en la parte principal de esta Recomendación y en el anexo A) que no restringe el uso del lenguaje debido a dominios de aplicación o metodologías subyacentes. La funcionalidad de las anteriores versiones de TTCN, tales como los índices de sucesión de pruebas que se pueden obtener mediante herramientas patentadas, no forma parte de TTCN-3.

5 Elementos de lenguaje básicos

La unidad de nivel máximo de TTCN-3 es un módulo. Un módulo no puede ser estructurado en submódulos. Un módulo puede importar definiciones de otros módulos. Los módulos pueden tener listas de parámetros para dar una forma de parametrización de sucesión de pruebas similar a los mecanismos de parametrización de PICS y PIXIT de TTCN-2.

Un módulo consiste en una parte de definiciones y una parte de control. La parte de definiciones de un módulo define los componentes de prueba, puertos de comunicación, tipos de datos, constantes, plantillas de datos de prueba, funciones, firmas para llamadas de procedimiento en puertos, casos de prueba, etc.

La parte de control de un módulo invoca los casos de prueba y controla su ejecución. La parte de control puede también declarar variables (locales), etc. Es posible utilizar enunciados de programa (tales como **`if-else`** y **`do-while`**) para especificar la selección y el orden de ejecución de los casos de prueba. TTCN-3 no soporta el concepto de variables globales.

TTCN-3 tiene un número de tipos de datos básicos predefinidos así como tipos estructurados, tales como registros (records), conjuntos (sets), uniones (unions), tipos enumerados (enumerated types) y matrices (arrays). Como una opción, los tipos y valores ASN.1 pueden ser utilizados con TTCN-3 mediante importación.

Una clase especial de valor de datos denominado una plantilla (template) proporciona mecanismos de parametrización y concordancia para especificar los datos de prueba que han de ser enviados o recibidos por los puertos de prueba. Las operaciones en estos puertos proporcionan capacidades de comunicación asíncrona y síncrona. Las llamadas de procedimiento pueden ser utilizadas para probar implementaciones que no se basan en mensajes.

El comportamiento de prueba dinámico se expresa como casos de prueba. Los enunciados de programa de TTCN-3 comprenden mecanismos poderosos de descripción de comportamientos, tales como recepción alternativa de comunicación y eventos de temporizador, entrelazado y comportamiento por defecto. Se soportan también los mecanismos de asignación y registro de veredictos de prueba.

Por último, es posible asignar atributos, tales como atributos de información de codificación y visualización, a la mayoría de los elementos de lenguaje de TTCN-3. Es posible también especificar atributos definidos por el usuario (no normalizados).

Cuadro 1/Z.140 – Visión general de los elementos de lenguaje de TTCN-3

Elemento de lenguaje	Palabra clave asociada	Especificado en definiciones de módulo	Especificado en control de módulo	Especificado en funciones/casos de prueba
Definición de módulos TTCN-3	**`module`**			
Importación de definiciones de otro módulo	**`import`**	Sí		
Agrupación de definiciones	**`group`**	Sí		
Definiciones de tipos de datos	**`type`**	Sí		
Definiciones de puertos de comunicación	**`port`**	Sí		
Definiciones de componentes de prueba	**`component`**	Sí		
Definiciones de firmas	**`signature`**	Sí		
Definiciones de funciones/constantes externas	**`external`**	Sí		
Definiciones de constantes	**`const`**	Sí	Sí	Sí
Definiciones de plantillas de datos/firmas	**`template`**	Sí		

Cuadro 1/Z.140 – Visión general de los elementos de lenguaje de TTCN-3

Elemento de lenguaje	Palabra clave asociada	Especificado en definiciones de módulo	Especificado en control de módulo	Especificado en funciones/casos de prueba
Definiciones de funciones	**function**	Sí		
Definiciones de alternativas denominadas	**named alt**	Sí		
Definiciones de casos de prueba	**testcase**	Sí		
Declaraciones de variables	**var**		Sí	Sí
Declaraciones de temporizadores	**timer**		Sí	Sí

5.1 Definiciones, ejemplificaciones y declaraciones

En la presente Recomendación el término "declaración" se utiliza de manera general para abarcar la formulación de una definición estática o la creación de alguna clase de ejemplificación dinámica cuando se da un nombre a un objeto TTCN-3. Por ejemplo, se definen tipos y constantes y un enunciado, tal como una invocación de una función o la declaración de una variable, es una ejemplificación. En ambos casos, cabe considerar que estas acciones equivalen a hacer una declaración.

5.2 Ordenación de elementos de lenguaje

En general, el orden en el cual se pueden hacer declaraciones y la mezcla de declaraciones con enunciados de programa es arbitrario. Sin embargo, dentro de un bloque de enunciados, por ejemplo, una rama de un enunciado **if-else**, todas las declaraciones (si hubiere alguna) se harán solamente al principio del bloque de enunciados.

Ejemplo:

```
// Ésta es una mezcla legal de declaraciones TTCN-3
:
var MyVarType MyVar2 := 3;
const integer MyConst:= 1;
if (x > 10)
{
   var integer MyVar1:= 1;
   :
   MyVar1:= MyVar1 + 10;
   :
}

   :
```

5.2.1 Referencias hacia adelante

Las definiciones en la parte de definiciones del módulo pueden ser hechas en cualquier orden y aunque se deben evitar las referencias hacia adelante (por motivos de legibilidad), esto no es obligatorio. Por ejemplo, los elementos recursivos, tales como funciones que llaman a otras funciones y parametrización de módulos, pueden originar referencias hacia adelante inevitables.

Las referencias hacia adelante sólo se permiten para declaraciones en la parte de definiciones del módulo, no se harán nunca dentro de la parte de control del módulo, definiciones de casos de prueba, funciones y alternativas denominadas. Esto significa que nunca se producirán referencias hacia adelante a variables locales, temporizadores locales y constantes locales.

5.3 Parametrización

TTCN-3 soporta parametrización de *tipos* y parametrización de *valores,* de acuerdo con las limitaciones siguientes:

a) los elementos de lenguaje que no pueden ser parametrizados son: **`const`** (constantes), **`var`** (variables), **`timer`** (temporizador), **`control`** (control), **`group`** (grupo) e **`import`** (importación);

b) el elemento de lenguaje **`module`** (módulo) permite la parametrización de valores *estática* para soportar parámetros de sucesión de pruebas, es decir, esta parametrización puede ser resuelta o no en el tiempo de compilación, pero será resuelta por el comienzo del tiempo de ejecución (es decir, *estática* en el tiempo de ejecución). Esto significa que, en el tiempo de ejecución, los valores de parámetros del módulo son globalmente visibles pero no pueden ser cambiados;

c) todas las definiciones de **`type`** (tipo) hechas por el usuario (incluidas las definiciones de tipos estructurados tales como **`record`** (registro), **`set`** (conjunto), etc.) y el tipo de configuración especial **`address`** (dirección) soportan parametrización de tipo *estática* y de valor *estática*, es decir, esta parametrización se resolverá en el tiempo de compilación;

d) los elementos de lenguaje **`signature`** (firma), **`testcase`** (caso de prueba) y **`function`** (función) soportan parametrización de valores *dinámica* (es decir, esta parametrización se resolverá en el tiempo de ejecución);

e) las alternativas denominadas soportan parametrización de valores *dinámica* (es decir, esta parametrización se resolverá en el tiempo de ejecución). Como las alternativas denominadas no son una unidad de ámbito, los parámetros formales definidos son sustituidos sencillamente por los parámetros reales dados cuando se ejecuta la expansión (macro) de **`named alt`** (alternativa denominada).

En el cuadro 2 figura un resumen de los elementos de lenguaje que pueden ser parametrizados y lo que puede ser transferido como parámetros.

Cuadro 2/Z.140 – Visión general de elementos de lenguaje de TTCN-3 parametrizables

Palabra clave	Parametrización de tipo	Parametrización de valor	Tipos de valores que pueden aparecer en listas de parámetros formales/reales
`module`		Estática al comienzo del tiempo de ejecución	*Valores de:* todos los tipos básicos, todos los tipos definidos por el usuario y tipo **`address`**
`type`	Estática en el tiempo de compilación	Estática en el tiempo de compilación	*Valores de:* todos los tipos básicos, todos los tipos definidos por el usuario y tipo **`address`**. NOTA – Las definiciones de **`record of`**, **`set of`**, **`enumerated`**, **`port`**, **`component`** y **`subtype`** no soportan parametrización.
`template`		Dinámica en el tiempo de ejecución	*Valores de:* todos los tipos básicos, todos los tipos definidos por el usuario, tipo **`address`**, tipo **`component`** y **`template`.**
`function`		Dinámica en el tiempo de ejecución	*Valores de:* todos los tipos básicos, todos los tipos definidos por el usuario, tipo **`address`**, tipo **`component`**, tipo **`port`**, **`template`** y **`timer`**.

Cuadro 2/Z.140 – Visión general de elementos de lenguaje de TTCN-3 parametrizables

Palabra clave	Parametrización de tipo	Parametrización de valor	Tipos de valores que pueden aparecer en listas de parámetros formales/reales
testcase		Dinámica en el tiempo de ejecución	*Valores de:* todos los tipos básicos, todos los tipos definidos por el usuario, tipo **address** y **template**.
signature		Dinámica en el tiempo de ejecución	*Valores de:* todos los tipos básicos, todos los tipos definidos por el usuario, tipo **address** y tipo **component**.
named alt		Expansión de macro estática	*Valores de:* todos los tipos básicos, todos los tipos definidos por el usuario, tipo **address**, tipo **component**, tipo **port**, **template** y **timer**.
NOTA – En las cláusulas pertinentes de la presente Recomendación figuran ejemplos de sintaxis y uso específico de la parametrización con los diferentes elementos de lenguaje.			

5.3.1 Introducción de parámetros por referencia y por valor

Por defecto, todos los parámetros de tipos básicos, tipos de cadena básicos, tipos estructurados definidos por el usuario, tipo de dirección y tipo de componente son introducidos por valor. Esto puede ser indicado facultativamente por la palabra clave **in** (entrada). Para introducir parámetros de los tipos mencionados por referencia se utilizarán las palabras clave **out** (salida) o **inout** (entrada/salida).

Los temporizadores y puertos son siempre introducidos por referencia y son identificados por las palabras clave **timer** (temporizador) y **port** (puerto). La palabra clave **inout** puede ser utilizada facultativamente para indicar introducción por referencia.

5.3.1.1 Parámetros introducidos por referencia

La introducción de parámetro por referencia tiene las siguientes limitaciones:

a) Sólo las listas de parámetros formales de **function**, **signature** y **testcase** pueden contener parámetros introducidos por referencia.

   NOTA – Hay otras restricciones sobre cómo utilizar parámetros introducidos por referencia en firmas (véase la cláusula 22).

b) Los parámetros reales sólo serán variables (por ejemplo, no serán constantes ni plantillas).

c) Sólo los parámetros de valor (es decir, no los parámetros de tipo) serán introducidos por referencia.

Ejemplo:

```
function MyFunction(inout boolean MyReferenceParameter){ ... };
// MyReferenceParameter es introducido por referencia. El parámetro real puede
// ser leído y fijado desde el interior de la función

function MyFunction(out boolean MyReferenceParameter){ ... };
// MyReferenceParameter es introducido por referencia. El parámetro real sólo
// puede ser fijado desde el interior de la función
```

5.3.1.2 Parámetros introducidos por valor

Los parámetros reales que son introducidos por valor pueden ser variables, así como constantes, plantillas, etc.

```
function MyFunction(in template MyTemplateType MyValueParameter){ ... };
// MyValueParameter es pasado por valor, la palabra clave in es facultativa
```

5.3.2 Listas de parámetros formales y reales

El número de elementos y el orden en el que aparecen en una lista de parámetros reales será igual que el número de elementos y el orden en el que aparecen en la correspondiente lista de parámetros formales. Además, el tipo de cada parámetro real será compatible con el tipo de cada parámetro formal correspondiente:

Ejemplo:

```
// Una definición de función con una lista de parámetros formales
function MyFunction(integer FormalPar1, boolean FormalPar2, bitstring
FormalPar3)
   { ... }

// Una llamada de función con una lista de parámetros reales
MyFunction(123, true,'1100'B);
```

5.3.3 Lista de parámetros formales vacía

Si la lista de parámetros formales de un elemento de lenguaje TTCN-3 parametrizable que parece una función (es decir, **function**, **testcase**, **signature**, **named alt** o **external**), está vacía, los paréntesis vacíos se incluirán en la declaración y en la invocación de ese elemento. En los demás casos, se omitirán los paréntesis vacíos.

Ejemplo:

```
// Una definición de función con una lista de parámetros vacía se escribirá como
function MyFunction(){ … }

// Una definición de registro con una lista de parámetros vacía se escribirá
como type record MyRecord { … }
```

5.3.4 Listas de parámetros jerarquizados

En general, todas las entidades parametrizadas especificadas como un parámetro real tendrán sus propios parámetros resueltos en la lista de parámetros reales.

Ejemplo:

```
// Dada la definición de mensaje
type record MyMessageType
{
   integer    field1,
   charstring field2,
   boolean    field3
}
// Una plantilla de mensaje pudiera ser
template MyMessageType MyTemplate(integer MyValue) :=
{
   field1 := MyValue,
   field2 := pattern "abc*xyz",
   field3 := true
}
```

```
// Un caso de prueba parametrizado con una plantilla pudiera ser
testcase TC001(template MyMessageType RxMsg) runs on PTC1 system TS1
{  :
   MyPCO.receive(RxMsg);
}

// Cuando el caso de prueba es llamado en la parte de control y la plantilla
// parametrizada se usa como un parámetro real, se debe proporcionar control a
// los parámetros reales para la plantilla
control
{  :
   TC001(MyTemplate(7));
   :
}
```

5.4 Reglas de alcance

TTCN-3 proporciona cinco unidades de ámbito básicas:

a) módulos;

NOTA – Hay reglas de alcance adicionales para grupos (véase 7.3.1).

b) parte de control de un módulo;

c) funciones;

d) casos de prueba;

e) bloques de enunciados dentro de control, funciones, y casos de prueba.

Cada unidad de ámbito consiste en declaraciones (facultativas) más cierta forma de descripción funcional (facultativa). Todas las unidades de ámbito, salvo los módulos, son jerárquicas, y cada nivel de la jerarquía define su propio ámbito local. Las declaraciones en un nivel de alcance más alto son visibles a los niveles más bajos (dentro de la misma jerarquía de alcance). Las declaraciones en un nivel de alcance más bajo no son visibles al nivel de alcance más alto.

Ejemplo:

```
module MyModule
{  :
   const integer MyConst := 0;  // MyConst es visible a MyBehaviourA y
                                // MyBehaviourB
   :
   function MyBehaviourA()
   {  :
      const integer A := 1;     // La constante A sólo es visible a
                                // MyBehaviourA
      :
   }

   function MyBehaviourB()
   {  :
     const integer B := 1;   // La constante B sólo es visible a
                             // MyBehaviourB
      :

   }
}
```

5.4.1 Alcance y reutilización de identificadores

TTCN-3 no soporta reutilización de identificadores, es decir, todos los identificadores en la misma jerarquía de alcance serán únicos. Esto significa que una declaración en un nivel de alcance más bajo no reutilizará el mismo identificador que una declaración en un nivel de alcance más alto (y en la misma jerarquía de alcance).

Ejemplo:

```
module MyModule
{  :
   const integer A := 1;
   :
   function MyBehaviourA()
   {  :
      const integer A := 1; // NO se permite
      :
      if(...)
      {  :
         const boolean A := true; // NO se permite
         :
      }
   }
}

// Lo siguiente ESTÁ permitido pues las constantes no se declaran en la misma
// jerarquía de ámbito (suponiendo que no hay declaración de A en un
// encabezamiento de módulo)
function MyBehaviourA()
{  :
   const integer A := 1;
   :
}

function MyBehaviourB()
{  :
   const integer A := 1;
   :
}
```

5.4.2 Alcance de parámetros formales

El alcance de los parámetros formales en un elemento de lenguaje parametrizado (por ejemplo, en una llamada de función) estará restringido a la definición en la cual los parámetros aparecen y a los niveles de alcance más bajos en la misma jerarquía de alcance. Es decir, seguirán las reglas de alcance normales (véase 5.4). Las reglas de reutilización de identificadores (véase 5.4.1) se aplicarán también a parámetros formales.

5.5 Identificadores y palabras clave

Los identificadores TTCN-3 son sensibles a mayúsculas y minúsculas y las palabras clave de TTCN-3 se escribirán enteramente con letras minúsculas (véase el anexo A).

6 Tipos y valores

TTCN-3 soporta un número de tipos básico predefinidos. Estos tipos básicos comprenden los normalmente asociados con un lenguaje de programación, tales como **`integer`**, **`boolean`** y tipos de cadena, así como algunos específicos de TTCN-3, tales como **`objid`** y **`verdicttype`**. Es posible construir tipos estructurados, tales como **`record`**, **`set`** y **`enumerated`**, a partir de estos tipos básicos.

Los tipos especiales asociados con configuraciones, tales como **`address`**, **`port`** y **`component`**, pueden ser utilizados para definir la arquitectura del sistema de prueba (véase la cláusula 21).

En el cuadro 3 se resumen los tipos TTCN-3.

Cuadro 3/Z.140 – Visión general de tipos TTCN-3

Clase de tipo	Palabra clave	Subtipo
Tipos básicos	**integer**	gama, lista
	char	gama, lista
	universal char	gama, lista
	float	lista
	boolean	lista
	objid	lista
	verdicttype	lista
Tipos de cadena básicos	**bitstring**	lista, longitud
	hexstring	lista, longitud
	octetstring	lista, longitud
	charstring	lista, longitud
	universal charstring	lista, longitud
Tipos estructurados definidos por el usuario	**record**	lista
	record of	lista
	set	lista
	set of	lista
	enumerated	lista
	union	lista
Tipos de configuraciones especiales	**address**	
	port	
	component	

6.1 Tipos y valores básicos

TTCN-3 soporta los siguientes tipos básicos:

a) **integer**: un tipo con valores distinguidos que son números completos positivos y negativos, incluido cero.

Los valores de tipo **integer** serán indicados por una o más cifras, la primera cifra no será cero, a menos que el valor sea 0; el valor cero será representado por un solo cero.

b) **char**: un tipo cuyos valores distinguidos son caracteres de la Rec. UIT-T T.50 e ISO/CEI 646 [5].

Los valores de tipo **char** pueden aparecer colocados entre comillas (") o calculados utilizando una función de conversión predefinida con el valor entero positivo de su codificación como argumento.

El orden entre los valores de tipo **char** es definido por el valor entero de su codificación, es decir, los operadores relacionales ==, <, >, !=, >= y <= pueden ser utilizados para comparar valores de tipo **char**.

c) **universal char**: un tipo cuyos valores distinguidos son caracteres de ISO/CEI 10646-1 [6].

Los valores de tipo **universal char** pueden aparecer colocados entre comillas (") o calculados utilizando una función de conversión predefinida con el valor entero positivo de su codificación como argumento.

El orden entre los valores de tipo **char** es definido por el valor entero de su codificación, es decir, los operadores relacionales ==, <, >, !=, >= y <= pueden ser utilizados para comparar valores de tipo **universal char**.

d) **float**: un tipo para describir números con coma flotante.

Los números con coma flotante se representan como: $<mantissa>*<base>^{<exponent>}$

Donde <*mantissa*> es un entero positivo o negativo, <*base*> un entero positivo (en la mayoría de los casos, 2, 10 ó 16) y <*exponent*> un entero positivo o negativo.

La representación de número con coma flotante está restringida a una base con el valor de 10. Los valores de coma flotante pueden ser expresados:

- utilizando la notación normal con una coma en una secuencia de números tal como 1,23 (que representa $123*10^{-2}$), 2,783 (por ejemplo, $2783*10^{-3}$) o –123,456789 (que representa $-123456789*10^{-6}$); o
- por dos números separados por E, donde el primer número especifica la mantisa y el segundo especifica el exponente, por ejemplo 12,3E4 (que representa $12,3*10^{4}$) o –12,3E-4 (que representa $-12,3*10^{-4}$).

e) **boolean**: un tipo que consiste en dos valores distinguidos.

Los valores de tipo booleano serán indicados por **true** (verdadero) y **false** (falso).

f) **objid**: un tipo cuyos valores distinguidos son el conjunto de todos los identificadores de objeto asignados de acuerdo con las reglas indicadas en [7], [8], [9] y [10]. Por ejemplo:

{itu-t(0) identified-organization(4) etsi(0)}

o alternativamente: {itu-t identified-organization etsi}

o alternativamente: { 0 4 0}

g) **verdicttype**: un tipo que se ha de utilizar con veredictos de prueba que consisten en cuatro valores distinguidos.

Los valores de **verdicttype** serán indicados por **pass**, **fail**, **inconc**, **none** y **error**.

6.1.1 Tipos y valores de cadena básicos

TTCN-3 soporta los siguientes tipos de cadena básicos:

NOTA – El término general cadena o tipo de cadena en TTCN-3 indica **bitstring**, **hexstring**, **octetstring**, **charstring** y **universal charstring**.

a) **bitstring**: un tipo cuyos valores distinguidos son las secuencias ordenadas de ninguno, uno o más bits.

Los valores de tipo **bitstring** serán indicados por un número arbitrario (posiblemente cero) de ceros y unos, precedidos por una sola comilla (') y seguidos por el par de caracteres 'B. Por ejemplo:

```
'01101'B
```

b) **hexstring**: un tipo cuyos valores distinguidos son las secuencias ordenadas de ninguno, uno o más dígitos hexadecimales, cada uno correspondiente a una secuencia ordenada de cuatro bits.

Los valores de tipo **hexstring** serán indicados por un número arbitrario (posiblemente cero) de los dígitos hexadecimales:

```
1 2 3 4 5 6 7 8 9 A B C D E F
```

precedidos por una sola comilla (') y seguidos por el par de caracteres 'H; Cada dígito hexadecimal se utiliza para indicar el valor de un semiocteto utilizando una representación hexadecimal. Por ejemplo:

```
'AB01D'H
```

c) **octetstring**: un tipo cuyos valores distinguidos son las secuencias ordenadas de ninguno o un número par positivo de dígitos hexadecimales (cada par de dígitos correspondiente a una secuencia ordenada de ocho bits).

Los valores de tipo **octetstring** serán indicados por un número arbitrario, pero par (posiblemente cero) de los dígitos hexadecimales.

```
1 2 3 4 5 6 7 8 9 A B C D E F
```

precedidos por una sola comilla (') y seguidos por el par de caracteres 'O; cada dígito hexadecimal se utiliza para indicar el valor de un semiocteto con una representación hexadecimal. Por ejemplo:

```
'FF96'O
```

d) **charstring**: son tipos cuyos valores distinguidos son ninguno, uno o más caracteres de UIT-T T.50 e ISO/CEI 646 [5]. El tipo de cadena de caracteres precedido por la palabra clave **universal** indica tipos cuyos valores distinguidos son ninguno, uno o más caracteres de ISO/CEI 10646-1 [6].

Los valores de tipo **charstring** y de tipo **universal charstring** serán indicados por un número arbitrario (posiblemente cero) de caracteres del juego de caracteres pertinente, precedidos y seguidos por comillas dobles (").

Cuando es necesario definir cadenas que incluyen el carácter comillas dobles ("), el carácter es representado por un par de comillas dobles en la misma línea sin caracteres de espacio. Por ejemplo:, ""abcd"" representa la cadena literal "abcd".

6.1.2 Acceso a elementos de cadena

Es posible acceder a cada elemento en un tipo cadena utilizando una sintaxis similar a matrices. Sólo se puede acceder a los elementos individuales de la cadena.

Las unidades de longitud de diferentes elementos de tipo cadena se indican en el cuadro 4.

Cuadro 4/Z.140 – Unidades de longitud utilizadas en especificaciones de longitud de campo

Tipo	Unidades de longitud
bitstring	bits
hexstring	dígitos hexadecimales
octetstring	octetos
character strings	caracteres

Los índices comenzarán con el valor cero (0). Por ejemplo:

```
// Dado
MyBitString := '11110111'B;
// Haciendo
MyBitString[4] := '1'B;
// Resulta en la cadena de bits '11111111'B
```

6.2 Subtipos y valores definidos por el usuario

Los tipos definidos por el usuario serán indicados por la palabra clave **type**. Con tipos definidos por el usuario es posible hacer subtipos (tales como listas, gamas y restricciones de longitud) en **integer** y los diversos tipos de cadena.

6.2.1 Lista de valores

TTCN-3 permite la especificación de una lista de valores distinguidos de cualquier tipo dado enumerado en el cuadro 3. Los valores en la lista serán del tipo básico y serán un subconjunto verdadero de los valores definidos por el tipo básico. El subtipo definido por esta lista restringe los valores permitidos del subtipo a los valores de la lista. Por ejemplo:

```
type bitstring MyListOfBitStrings ('01'B, '10'B, '11'B);
```

6.2.2 Gamas

TTCN-3 permite la especificación de una gama de valores de tipo **integer**, **char** y **universal char** (o derivaciones de estos tipos). El subtipo definido por esta gama comprende restricciones de los valores permitidos del subtipo a los valores en la gama que incluye la frontera inferior y la frontera superior. Por ejemplo:

```
type integer MyIntegerRange (0 .. 255);
```

6.2.2.1 Gamas infinitas

Para especificar una gama de enteros infinita, se puede utilizar la palabra clave **infinity** en vez de un valor que indique que no hay frontera inferior o superior. La frontera superior será mayor o igual que la frontera inferior. Por ejemplo:

```
type integer MyIntegerRange (-infinity .. –1);   // Todos números enteros
                                                 // negativos
```

NOTA – El 'valor' para **infinity** depende de la implementación. El uso de esta característica puede ocasionar problemas de portabilidad.

6.2.2.2 Mezcla de listas y gamas

Para los valores de tipo **integer**, **char** y **universal char** (o derivados de estos tipos) es posible mezclar listas y gamas. Por ejemplo:

```
type integer MyIntegerRange (1, 2, 3, 10 .. 20, 99, 100);
```

6.2.3 Restricciones de longitud de cadena

TTCN-3 permite especificar restricciones de longitud en tipos de cadenas. Las fronteras de longitud son de complejidad diferente dependiendo del tipo de cadena con el cual se utilizan. En todos los casos, estas fronteras evaluarán a valores **integer** no negativos (o valores **integer** derivados). Por ejemplo:

```
type bitstring MyByte length(8);              // Exactamente longitud 8
type bitstring MyByte length(8 .. 8);         // Exactamente longitud 8
type bitstring MyNibbleOrByte length(4 .. 8); // Longitud mínima 4,
                                              // Longitud máxima 8
```

El cuadro 4 especifica las unidades de longitud para diferentes tipos de cadena.

Para el límite superior, se puede utilizar también la palabra clave **infinity**, para indicar que no hay límite superior para la longitud. La frontera superior será mayor o igual que la frontera inferior.

6.3 Tipos y valores estructurados

La palabra clave **type** se utiliza también para especificar tipos estructurados, tales como tipos **record**, tipos **record of**, tipos **set**, tipos **set of**, tipos **enumerated** y tipos **union**.

Los valores de estos tipos pueden ser dados utilizando una notación de asignación explícita o un inicializador abreviado. Por ejemplo:

```
const MyRecordType MyRecordValue:=
{
   field1 := '11001'B,
   field2 := true,
   field3 := "A string"
}

// Or
const MyRecordType MyRecordValue:= {'11001'B, true, "A string"}
```

No se permite mezclar las dos notaciones de valor en el mismo contexto (inmediato). Por ejemplo:

```
// Esto no está admitido
const MyRecordType MyRecordValue:= {MyIntegerValue, field2 := true, "A string"}
```

6.3.1 Tipos y valores de registro

TTCN-3 soporta tipos estructurados ordenados conocidos como **record**. Los elementos de un tipo **record** pueden ser cualquiera de los tipos básicos o tipos definidos por el usuario, tales como otros registros, conjuntos o matrices. Los valores de un registro serán compatibles con los tipos de campos de registro. Los identificadores de elementos son locales del registro y serán únicos dentro de éste. Una constante que es de tipo **record** no contendrá variables (incluidos parámetros de módulo) como valores de campo, directa o indirectamente.

```
type record MyRecordType
{
   integer       field1,
   MyOtherStruct field2 optional,
   charstring    field3
}

type record MyOtherstructType
{
   bitstring  field1,
   boolean    field2
}
```

Los registros pueden ser definidos sin campos (es decir, como registro vacíos). Por ejemplo:

```
type record MyEmptyRecord {  }
```

Un valor **record** es asignado elemento por elemento. Por ejemplo:

```
var integer MyIntegerValue:= 1;

var MyRecordType MyRecordValue:=
{
   field1 := MyIntegerValue,
   field2 := MyOtherRecordValue,
   field3 := "A string"
}
```

```
const MyOtherRecordType MyOtherRecordValue:=
{
   field1 := '11001'B,
   field2 := true
}
```

o utilizando un inicializador. Por ejemplo:

```
MyRecordValue:= {MyIntegerValue, {'11001'B, true}, "A string"};
```

Para campos facultativos se permite omitir el valor utilizando el símbolo de omitir parámetro. Por ejemplo:

```
MyRecordValue:= {MyIntegerValue, - , "A string"};

// Obsérvese que esto es igual que escribir, es decir, el valor de field2 es
// indefinido
MyRecordValue.field1 := MyIntegerValue;
MyRecordValue.field3 := "A string"
```

6.3.1.1 Referencia de campos de registro jerarquizados

Los elementos de registro jerarquizados son referenciados por pares de *RecordId.ElementId*. Por ejemplo:

```
MyVar1 := MyRecord1.MyElement1;
// Si un registro está jerarquizado, la referencia puede parecer como esto
MyVar2 := MyRecord1.MyElement1.MyRecord2.MyElement2;
```

6.3.1.2 Elementos facultativos en un registro

Los elementos facultativos en un **record** serán especificados utilizando la palabra **optional**. Por ejemplo:

```
type record MyMessageType
{
   FieldType1 field1,
   FieldType2 field2  optional,
   :
   FieldTypeN fieldN
}
```

6.3.2 Tipos y valores de conjuntos

TTCN-3 soporta tipos estructurados no ordenados conocidos como **set**. Los tipos y valores de conjunto son similares a registros, salvo que la ordenación de los campos no es significativa. Por ejemplo:

```
type set MySetType
{
   integer    field1,
   charstring field2
}
```

La notación de inicializador para fijar valores no se utilizará para valores de tipos **set**.

6.3.2.1 Elementos facultativos en un conjunto

Los elementos facultativos en un **set** serán especificados utilizando la palabra clave **optional**.

6.3.3 Registros y conjuntos de tipos

TTCN-3 soporta la especificación de registros y conjuntos cuyos elementos son todos del mismo tipo. Éstos se indican utilizando la palabra clave **of**. Estos registros y conjuntos no tienen identificadores de elementos y pueden ser considerados similares a una matriz ordenada y una matriz no ordenada, respectivamente.

La palabra clave **length** se utiliza para restringir longitudes de **record of** y **set of**. Por ejemplo:

```
type record of length(10) integer MyRecordOfType; // Es un registro con un
                                                   // máximo de 10 enteros
type set of boolean MySetOfType;                   // Es un conjunto ilimitado de
                                                   // valores booleanos
type record of length(10) charstring StringArray length(10);
                                  // Es un registro con un máximo de 10
                                  // cadenas, cada una con una longitud
                                  // máxima de 10 caracteres
```

La notación de valor para **record of** y **set of** es igual que la notación de valor para matrices (véase 6.4).

6.3.4 Tipos y valores enumerados

TTCN-3 soporta tipos enumerados. Los tipos **enumerated** se utilizan para modelar tipos que sólo toman un conjunto de valores denominados distintos. Las operaciones en tipos enumerados sólo utilizarán los identificadores denominados y están restringidas a operadores de asignación, equivalencia y ordenación.

Cada valor denominado puede tener facultativamente un valor entero asociado, que se define después del nombre entre paréntesis. Estos valores sólo son utilizados por el sistema para permitir el uso de operadores relacionales. Si no se indican enteros explícitos, se supone que la ordenación comienza con cero. Por ejemplo:

```
type enumerated MyEnumType
{
   Monday, Tuesday, Wednesday, Thursday, Friday
}

// Un caso válido de ejemplificación de MyEnumType sería
var MyEnumType Today     := Monday;
var MyEnumType Tomorrow  := Tuesday;
// y el enunciado Today < Tomorrow es verdadero
```

6.3.5 Uniones

TTCN-3 soporta tipos **union**. Los tipos **union** son similares a registro, salvo que sólo uno de los campos especificados estará presente en un valor de unión real. Los tipos **union** son útiles para modelar una estructura que puede tener uno de un número finito de tipos conocidos. Por ejemplo:

```
type union MyUnionType
{
   integer       number,
   charstring    string
}
// Un caso válido de ejemplificación de MyUnionType sería
var MyUnionType age;
age.number := 34;
```

La notación de inicializador para fijar valores no se utilizará para valores de tipos **union**.

La palabra clave **optional** no se utilizará con tipos **union**.

6.4 Matrices

En común con muchos lenguajes de programación, no se considera que las matrices son tipos TTCN-3. En cambio, pueden ser especificadas en el punto de una declaración de variables. Por ejemplo:

```
var integer MyArray[3];   // Ejemplifica una matriz de enteros de 3 elementos
                          // con el índice 0 a 2
```

Los valores de elementos de matrices serán compatibles con la correspondiente declaración de variables. Los valores pueden ser asignados individualmente o todos a la vez. Por ejemplo:

```
MyArray[0]:= 10;
MyArray[1]:= 20;
MyArray[2]:= 30;

// o usando un inicializador
MyArray:= {10, 20, 30};
```

Los índices de matrices son expresiones que darán un valor **integer** positivo, incluido el valor cero. Por defecto, el índice de matrices de TTCN-3 comenzará con la cifra 0 (cero).

Las dimensiones de la matriz se especificarán utilizando expresiones constantes que evaluarán a un valor **integer** positivo. Las dimensiones de la matriz pueden ser especificadas también utilizando gamas. En estos casos, los valores inferior y superior de la gama definen los valores de índices inferior y superior. Por ejemplo:

```
var integer MyArray[1 .. 5];   // Ejemplifica una matriz de enteros de
                               // 5 elementos con el índice 1 a 5
MyArray[1] := 10;// Índice más bajo
MyArray[5] := 50;// Índice más alto
```

Las matrices de registro de tipos ofrecen la posibilidad de especificar matrices multidimensionales. Por ejemplo:

```
// Dado
type record MyRecordType
{
   integer       field1,
   MyOtherStruct field2,
   charstring    field3
}
// Una matriz de MyRecordType podría ser
var MyRecordType MyRecordArray[10];
// Una referencia a un elemento particular parecería como esto
MyRecordArray[1].field1 := 1;
```

6.5 Tipos recursivos

Cuando sea aplicable, las definiciones de tipos de TTCN-3 pueden ser recursivas. Sin embargo, el usuario asegurará que toda recursión de tipos puede resolverse y que no se produce recursión infinita.

6.6 Parametrización de tipos

La parametrización de tipos permite identificadores de tipo ficticio que actúan como sustituidores para cualquier tipo. Esto significa que el especificador de TTCN-3 puede dejar abierto un tipo mientras pueda resolverse en el tiempo de compilación.

NOTA – Ésta es una generalización del concepto de metatipos de unidades de datos de protocolo de TTCN-2.

El tipo real sólo se conoce cuando se utiliza realmente el parámetro de tipo. Por ejemplo:

```
type record MyRecordType(MyMetaType)
{
   boolean  field1,
   MyMetaType  field2   // MyMetaType no es un tipo particular
}

var MyRecordType(integer) MyRecordValue :=
{
   field1 := true,
   field2 := 123       // MyMetaType es ahora de tipo integer
}
```

6.7 Compatibilidad de tipos

TTCN-3 no está muy tipificada pero el lenguaje requiere compatibilidad de tipos. Las variables, constantes, plantillas, etc., tienen tipos compatibles si éstos se resuelven en el mismo tipo básico y, en el caso de asignaciones, concordancia, etc., no se viola la subtipificación (por ejemplo, gamas, restricciones de longitud).

Por ejemplo:

```
// Dado
type integer MyInteger(1 .. 10);
:
var integer x;
var MyInteger y;

// Entonces
x := 20; // Es una asignación válida
y := 20; // NO es una asignación válida porque 20 no está en la gama de y

y := 5;  // Es una asignación válida

x := y;  // Es una asignación válida, porque el valor de y está en la gama de x
y := x;  // NO es una asignación válida, porque el valor de x no está en la
         // gama de y

x := 5;  // Es una asignación válida
y := x;  // Es una asignación válida, porque el valor de x está ahora en la
         // gama de y
```

6.7.1 Conversión de tipos

Si es necesario convertir valores de un tipo a valores de otro tipo, cuando los tipos no se derivan del mismo tipo básico, se utilizará una de las funciones de conversión predefinidas del anexo D o una función definida por el usuario. Por ejemplo:

```
// Para convertir un valor integer en un valor hexstring se usa la función
predefinida int2hex
   MyHstring := int2hex(123, 4);
```

7 Módulos

Los principales bloques de construcción de TTCN-3 son módulos. Por ejemplo, un módulo puede definir una serie de pruebas totalmente ejecutables o sólo una biblioteca. Un módulo consiste en una parte de definiciones (facultativa) y una parte de control de módulo (facultativa).

NOTA – El término "sucesión de pruebas" es sinónimo de un módulo TTCN-3 completo que contiene casos de prueba y una parte de control.

7.1 Denominación de módulos

Los nombres de módulo tienen la forma de un identificador TTCN-3 seguido por un identificador de objeto facultativo.

NOTA – El identificador de módulo es el nombre textual informal del módulo.

7.2 Parametrización de módulos

La lista de parámetros **module** define un conjunto de valores que son suministrados por el entorno de prueba en el tiempo de ejecución. Durante la ejecución de la prueba, estos valores serán tratados como constantes. Por ejemplo:

```
module MyParameterizedModule(integer TS_Par1, boolean TS_Par2, hexstring
                                TS_Par3)
{ ... }
```

NOTA – Esto proporciona una funcionalidad similar a la de los parámetros de series de prueba de TTCN-2 que suministra valores de PICS y PIXIT de la serie de pruebas.

7.2.1 Valores por defecto para parámetros de módulo

Para los casos en que los valores de parámetros de módulo reales no son proporcionados por el entorno de prueba en el tiempo de ejecución, se permite especificar valores por defecto para parámetros de módulo. Esto será efectuado por una asignación en la lista de parámetros de módulo. Por ejemplo:

```
module MyModuleDefaultParameter(integer Par1 := 1234, boolean Par2 := false)
{ ... }
```

7.3 Parte de definiciones del módulo

La parte de definiciones del módulo especifica las definiciones de nivel máximo del módulo. Estas definiciones pueden ser utilizadas en cualquier parte en el módulo, incluida la parte de control. Los elementos de lenguaje que pueden ser definidos en un módulo TTCN-3 se enumeran en el cuadro 1. Las definiciones del módulo pueden ser importadas por otros módulos.

Ejemplo:

```
module MyModule
{  // Este módulo contiene definiciones solamente
   :
   const integer MyConstant := 1;
   type record MyMessageType { ... }
   :
   function TestStep(){ ... }
   :
}
```

Las declaraciones de elementos de lenguaje dinámicos, tales como **var** o **timer**, sólo se harán en la parte de control, casos de prueba o funciones.

NOTA – TTCN-3 no soporta la declaración de variables en la parte de definiciones del módulo, sólo en la parte de control. Esto significa que en TTCN-3 no se puede definir variables globales.

7.3.1 Grupos de definiciones

En la parte de definiciones del módulo, las definiciones pueden ser recopiladas en grupos denominados. Es posible especificar un grupo de declaraciones cuando se permite una sola declaración. Los grupos pueden ser jerarquizados, es decir, los grupos pueden contener otros grupos. Esto permite al especificador de series de pruebas estructurar, entre otras cosas, colecciones de datos de prueba o funciones que describen el comportamiento de la prueba.

La agrupación se hace para facilitar la legibilidad y añadir estructura lógica a la serie de pruebas, si es necesario. Esto significa que todos los identificadores de las declaraciones en el conjunto de grupos (incluidos cualesquiera grupos jerarquizados) en cualquier nivel dado de agrupación serán únicos. En otras palabras, los grupos y los grupos jerarquizados no tienen alcance *salvo* en el contexto de cualesquiera atributos dados al grupo por un enunciado **`with`** asociado. En estos casos, un enunciado **`with`** en un grupo externo es sustituido por un enunciado **`with`** en un grupo interno.

Ejemplo:

```
// Una colección de definiciones
group MyGroup
{
   const integer MyConst:= 1;
   :
   type record MyMessageType { ... }
   }

// Un grupo de pasos de prueba
   group MyTestStepLibrary
   {
      group MyGroup1
      {
         function MyTestStep11() { ... }
         function MyTestStep12() { ... }
         :
         function MyTestStep1n() { ... }
      }
      group MyGroup2
      {
         function MyTestStep21() { ... }
         function MyTestStep22() { ... }
         :
         function MyTestStep2n() { ... }
      }
   }
```

7.4 Parte de control del módulo

La parte de control del módulo describe el orden de ejecución (posiblemente repetitivo) de los casos de prueba reales. Un caso de prueba será definido en la parte de definiciones del módulo e invocado en la parte de control.

Ejemplo:

```
module MyTestSuite
{  // Este módulo contiene definiciones ...
   :
   const integer MyConstant := 1;
   type record MyMessageType { ... }
   template MyMessageType MyMessage := { ... }
   :
   function MyFunction1() { ... }
   function MyFunction2() { ... }
   :
   testcase MyTestcase1() runs on MyMTCType { ... }
   testcase MyTestcase2() runs on MyMTCType { ... }
   :
   // ... y una parte de control, de modo que el control es ejecutable
   control
   {
      var boolean MyVariable; // Variable de control local
      :
```

```
      MyTestCase1(); // Ejecución secuencial de casos de prueba
      MyTestCase2();
      :
   }
}
```

7.5 Importación de módulos

Es posible reutilizar definiciones especificadas en diferentes módulos mediante el enunciado **`import`**. TTCN-3 no tiene construcciones de exportación explícitas, de modo que, por defecto, todas las definiciones de módulo en la parte de definiciones del módulo pueden ser importadas. Un enunciado **`import`** puede ser utilizado en cualquier lugar en la parte de definiciones del módulo. No se utilizará en la parte de control.

Si una definición importada tiene atributos (definidos por medio de un enunciado **`with`**), se importarán también los atributos.

NOTA – Si el módulo tiene atributos globales, éstos son asociados a definiciones sin estos atributos.

Ejemplo:

```
module MyModuleA
{  // Este módulo contiene definiciones y definiciones importadas
   :
   const integer MyConstant := 1;
   import all from MyModuleB;  // El ámbito de las definiciones importadas es
                               // global para MyModuleA
   type record MyMessageType { ... }
   :
   function MyBehaviourC()
   {
      const integer MyConstant := 2;
      // en este caso no se puede usar importación
      :
   }
   :
   control
{  // en este caso no se puede usar importación
   :
   }
}
```

7.5.1 Reglas sobre la utilización de importación

Al utilizar la importación, se aplicarán las siguientes reglas:

a) Sólo las definiciones de nivel máximo en el módulo pueden ser importadas explícitamente. Las definiciones que se producen en un ámbito más bajo (por ejemplo, constantes locales definidas en una función), no serán importadas.

b) Por defecto, todas las definiciones dependientes de otras definiciones (por ejemplo, tipos **`record`**) son importadas junto con todas las definiciones de las cuales dependen. Si no se desea importar estas dependencias, se puede utilizar la directiva **`nonrecursive`**.

c) Es posible también importar grupos de definiciones. Sin embargo, los grupos sólo son utilizados para fines de estructuración y no tienen unidades de ámbito. Por tanto, se permite importar subgrupos, es decir, un grupo que es definido dentro de otro grupo.

7.5.2 Importación de definiciones individuales

Es posible importar definiciones individuales. Por ejemplo:

```
import type MyType from MyModuleC;
```

7.5.3 Importación de todas las definiciones de un módulo

Es posible importar todo el contenido de una parte de definiciones del módulo (pero no el propio módulo real), por ejemplo:

```
import all from MyModule;
```

7.5.4 Importación de grupos

Es posible importar grupos, por ejemplo:

```
import group MyGroup from MyModule;
```

Los subgrupos, es decir, los grupos que son definidos dentro de otro grupo, son también importados por este enunciado.

7.5.5 Importación de definiciones de la misma clase

Es posible importar bloques de la misma clase de definiciones. Por ejemplo:

```
import all template from MyModule;
```

7.5.6 Importación recursiva de definiciones complejas

Por defecto, las definiciones recursivas, es decir, definiciones que hacen referencia a otras definiciones, son importadas implícitamente por el enunciado **import**. Ejemplos de definiciones recursivas son tipos **record** junto con sus tipos de componentes o funciones que llaman a otras funciones. Por ejemplo:

```
import type MyType from MyModuleC;
```

Todas las definiciones implícitamente importadas son visibles en el nivel máximo de alcance y pueden ser utilizadas a continuación del enunciado **import**.

Obsérvese que las definiciones locales con definiciones circundantes, por ejemplo, declaraciones de constantes locales dentro de una definición, nunca serán visibles.

Ejemplo:

```
// Dado
module MyModuleA
{  :
   function MyBehaviourB() { ... }
   function MyBehaviourA()
   {  :
      MyBehaviourB();
      :
      const integer LocalConst:= 1000;
      :
   }
}

// Entonces
module MyModuleB
{  :
   import function MyBehaviourA from MyModuleA;
   :
}
```

```
// Importará también y hará visible MyBehaviourB. La constante LocalConst estará
// insertada aún en MyBehaviourA y no será visible (fuera de MyBehaviourA).
```

Si las definiciones importadas de un módulo dependen de definiciones en otro módulo, las definiciones del otro módulo son importadas también, es decir, las importaciones serán definiciones dependientes implícitamente de la importación del módulo de terceros. Esto se debe a la regla de que una definición importada es tratada de la misma manera que una definición definida en el propio módulo.

Si se desea inhibir importaciones recursivas, se utilizará la directiva **nonrecursive**. Por ejemplo:

```
import type MyType from MyModuleC nonrecursive;
```

7.5.7 Tratamiento de conflictos de nombres en importación

Todos los módulos TTCN-3 tendrán su propio espacio de nombre en el cual todas las definiciones serán identificadas de manera única. Se pueden producir conflictos de nombres debido a importación, por ejemplo, importación de diferentes módulos, importación de grupos o importación de definiciones recursivas. Los conflictos de nombres serán resueltos poniendo un prefijo a la definición importada (que origina el conflicto de nombres) por el identificador del módulo del cual es importada. El prefijo y el identificador serán separados por un punto (.).

En los casos en que no hay ambigüedades, el prefijo no tiene que estar siempre presente cuando se utilizan definiciones importadas.

Ejemplo:

```
module MyModuleA
{  :
   type bitstring MyTypeA;
   import type MyTypeA from SomeModuleC;   // Donde MyTypeA es de tipo cadena de
                                           // caracteres
   import type MyTypeB from SomeModuleC;   // Donde MyTypeB es de tipo cadena de
                                           // caracteres
   :
   control
   {  :
      var   SomeModuleC.MyTypeA MyVar1 := "Test String"; // Se debe usar prefijo
      var MyTypeA MyVar2 := '10110011'B;// Éste es el MyTypeA original
      :
      var MyTypeB MyVar3 := "Test String"; // No hay que usar prefijo …
      var SomeModuleC.MyTypeB MyVar3 := "Test String";
                                           // ... pero puede, si se desea
      :
   }
}
```

NOTA – Se supone siempre que las definiciones con el mismo nombre definidas en distintos módulos sean diferentes, incluso si las definiciones reales en los diferentes módulos son idénticas. Por ejemplo, la importación de un tipo que ya está definido localmente, incluso con el mismo nombre, conduciría a disponer de dos tipos diferentes en el módulo.

7.5.8 Tratamiento de múltiples referencias a la misma definición

La utilización de **import** en definiciones individuales, grupos de definiciones, definiciones de la misma clase, etc., puede originar situaciones en las que se hace referencia a la misma definición más de una vez. En estos casos, la definición será importada sólo una vez.

NOTA – Los mecanismos para resolver estas ambigüedades, por ejemplo, sobrescritura y envío de avisos al usuario, están fuera del ámbito de la presente Recomendación y deben ser proporcionados por las herramientas de TTCN-3.

7.5.9 Importación de parámetros de módulo

Si una definición importada utiliza un parámetro de módulo, este parámetro se incluirá también en la lista de parámetros del módulo *importador*.

7.5.10 Definiciones de importación de módulos no TTCN

La palabra clave **`language`** se utiliza para indicar los casos cuando se importan definiciones de tipos de módulos no TTCN. Por ejemplo:

```
Import type MyASN1Type from MyASN1Module language "ASN.1:1997";
```

Por defecto, el lenguaje es TTCN-3. Por ejemplo:

```
import type MyType from MyModule;
// es igual que importar el tipo
import type MyType from MyModule language "TTCN-3";
```

8 Configuraciones de prueba

TTCN-3 permite la especificación (dinámica) de configuraciones de prueba concurrentes (o configuración, para abreviar). Una configuración consiste en un conjunto de componentes de prueba interconectados con puertos de comunicación bien definidos y una interfaz de sistema de prueba explícita que define las fronteras del sistema de prueba. (Véase la figura 2.)

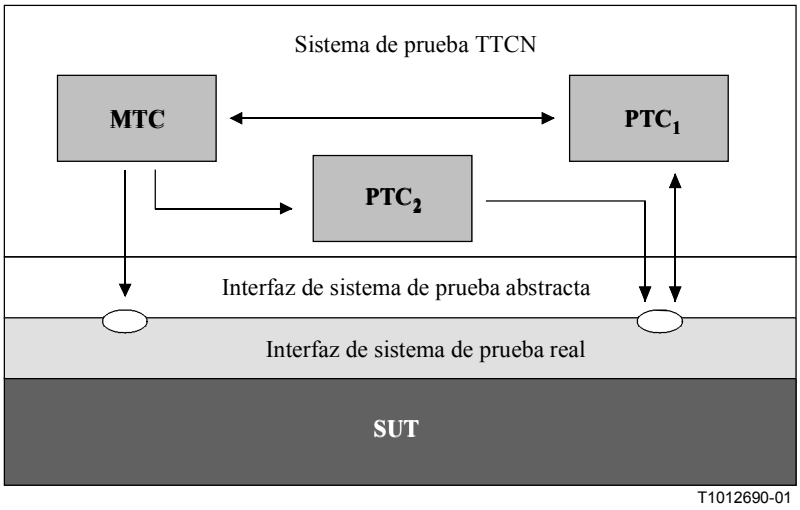


Figura 2/Z.140 – Visión conceptual de una configuración de prueba TTCN-3 típica

Dentro de cada configuración habrá solamente un componente de prueba principal (MTC). Los componentes de prueba que no son MTC se denominan componentes de prueba paralelos o PTC. Los MTC serán creados automáticamente al comienzo de la ejecución de cada caso de prueba. El comportamiento definido en el cuerpo del caso de prueba se ejecutará en este componente. Durante la ejecución de un caso de prueba, otros componentes pueden ser creados dinámicamente mediante el uso explícito de la operación **`create`**.

La ejecución del caso de prueba terminará cuando termina el MTC. Todos los demás PTC son creados igualmente, es decir, no hay relación jerárquica explícita entre ellos y la terminación de un solo PTC no termina otros componentes ni el MTC.

La comunicación se efectúa entre los componentes dentro del sistema de prueba y entre los componentes y la interfaz del sistema de prueba a través de puertos de comunicación.

Los tipos de componentes y los tipos de puertos de prueba, indicados por las palabras clave **`component`** y **`port`**, serán definidos en la parte de definición del módulo. La configuración real de

componentes y las conexiones entre ellos se logra ejecutando las operaciones **`create`** y **`connect`** dentro del comportamiento del caso de prueba. Los puertos de componentes están conectados a los puertos de la interfaz del sistema de prueba por medio de la operación **`map`** (véase 21.2).

8.1 Modelo de comunicación de puertos

Los componentes de prueba pueden estar conectados con otros componentes y con la interfaz del sistema de prueba. No hay restricciones del número de conexiones que un componente puede tener, pero un componente no estará conectado consigo mismo. Se permiten conexiones de uno a muchos.

Los componentes de prueba son conectados a través de sus puertos, es decir, las conexiones entre componentes y entre un componente y la interfaz del sistema de prueba se efectúa por puertos. Cada puerto está modelado como una cola FIFO infinita que almacena los mensajes entrantes o llamadas de procedimiento hasta que son procesados por el componente que posee ese puerto.

NOTA – Aunque en principio los puertos TTCN-3 son infinitos, en un sistema de prueba real pueden desbordar. Esto se debe tratar como un error de caso de prueba (véase 24.2.1).

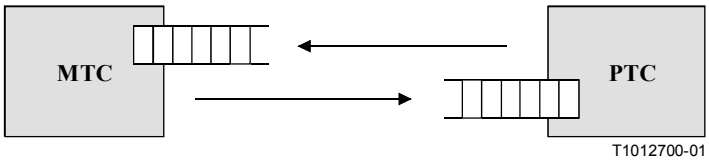


Figura 3/Z.140 – Modelos de puertos de comunicación de TTCN-3

8.2 Interfaz de sistema de prueba abstracta

TTCN-3 se utiliza para probar implementaciones. El objeto que se prueba se conoce como la implementación sometida a prueba o IUT. La IUT puede ofrecer interfaces directas para la prueba o puede formar parte del sistema, en cuyo caso el objeto probado se conoce como el sistema sometido a prueba o SUT. En el caso mínimo, la IUT y el SUT son equivalentes. En la presente Recomendación, el término "SUT" se utiliza de manera general para significar el SUT o la IUT.

En un entorno de prueba real, los casos de prueba tienen que comunicar con el SUT. Sin embargo, la especificación de la conexión física real está fuera del ámbito de TTCN-3. En cambio, una interfaz de sistema de prueba bien definida (pero abstracta) está asociada con cada caso de prueba. Una definición de interfaz de sistema de prueba es idéntica a una definición de componente, es decir, es una lista de todos los puertos de comunicación posibles a través de los cuales el caso de prueba está conectado con el SUT.

8.3 Definición de tipos de puertos de comunicación

Los puertos facilitan la comunicación entre componentes de prueba y entre componentes de prueba y la interfaz del sistema de prueba.

TTCN-3 soporta puertos basados en mensaje y basados en procedimiento. Cada puerto será definido como basado en mensaje o basado en procedimiento o mixto, lo que será indicado por la palabra clave **`message`** o palabra clave **`procedure`** dentro de la definición del tipo de puerto asociada.

Los puertos son direccionales. Los sentidos son especificados por las palabras clave **`in`** (para el sentido entrada), **`out`** (para el sentido salida) e **`inout`** (para ambos sentidos). Cada definición de tipo de puerto tendrá una o más listas que indican la colección permitida de tipos (mensajes) y/o procedimientos junto con el sentido de comunicación autorizado. Por ejemplo:

```
// Puerto basado en mensaje por el que se puede recibir tipos MsgType1 y
// MsgType2, enviar MsgType3 y enviar y recibir cualquier valor entero
```

```
type port MyMessagePortType message
{
   in       MsgType1, MsgType2;
   out      MsgType3;
   inout    integer
   }

// Puerto basado en procedimiento que permite la llamada distante de los
// procedimientos Proc1, Proc2 y Proc3. Obsérvese que Proc1, Proc2 y Proc3 se
// definen como firmas
type port MyProcedurePortType procedure
{
   out      Proc1, Proc2, Proc3
}
```

NOTA – El término "mensaje" se utiliza para significar mensajes definidos por plantillas y valores reales resultantes de expresiones. De este modo, la lista que restringe lo que se puede utilizar en un puerto basado en mensajes es simplemente una lista de nombres de tipos.

La utilización de la palabra clave **all** en una de las listas asociadas con un tipo de puerto permite pasar todos los tipos de puerto y/o firmas de procedimiento definidos en el módulo por ese puerto de comunicación. Por ejemplo:

```
// Puerto basado en mensaje por el que se puede transferir en ambos sentidos
// cualquier valor de todos tipos incorporados y tipos definidos por el usuario
type port MyAllMesssagesPortType message
{
   inout all
}
```

8.3.1 Puertos mixtos

Es posible definir un puerto que permite ambas clases de comunicación, lo que se indica mediante la palabra clave **mixed**. Esto significa que las listas para puertos mixtos serán siempre mixtas y que incluyen firmas y tipos. No se efectúa ninguna separación en la definición.

```
// Puerto mixto, que define un puerto basado en mensaje y un puerto basado
// en procedimiento con el mismo nombre. Las listas in, out e inout son
// también mixtas: MsgType1, MsgType2, MsgType3 e integer se refieren a la
// parte del puerto mixto basado en mensaje y Proc1, Proc2, Proc3, Proc4 y
// Proc5 se refieren al puerto basado en procedimiento.
type port MyMixedPortType mixed
{
   in       MsgType1, MsgType2, Proc1, Proc2;
   out      MsgType3, Proc3, Proc4;
   inout    integer, Proc5;
}}

// Puerto mixto, todos los tipos y firmas definidos en el módulo se pueden usar
// en este puerto para comunicar con el SUT u otros componentes de prueba

type port MyAllMixedPortType mixed
{
   inout    all
}
```

Un puerto mixto en TTCN-3 se define como una notación abreviada para dos puertos, es decir, un puerto basado en mensaje y un puerto basado en procedimiento con el mismo nombre. En el tiempo de ejecución, la distinción entre ambos puertos es efectuada por las operaciones de comunicación.

Las operaciones utilizadas para controlar puertos (véase la cláusula 21), es decir, **start**, **stop** y **clear**, se ejecutarán en ambas colas (en orden arbitrario) si son llamadas con un identificador de un puerto mixto.

8.4 Definición de tipos de componentes

El tipo **component** define los puertos que están asociados con un componente. Estas definiciones se efectuarán en la parte de definiciones del módulo. Los nombres de puertos en una definición de componente son locales de ese componente, es decir, otro componente puede tener puertos con los mismos nombres. Todos los puertos del mismo componente tendrán nombres únicos. Sin embargo, esto no significará que hay alguna conexión entre los componentes por estos puertos.

Ejemplo:

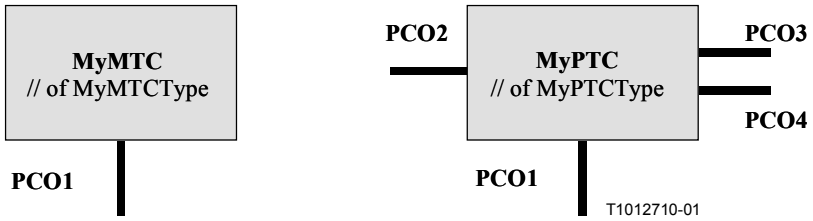


Figura 4/Z.140 – Componentes típicos

```
type component MyMTCType
{
   port MyMessagePortType       PCO1
}

type component MyPTCType
{
   port MyMessagePortType       PCO1, PCO4;
   port MyProcedurePortType     PCO2;
   port MyAllMesssagesPortType  PCO3
}
```

8.4.1 Declaración de variables y temporizadores locales en un componente

Es posible declarar variables y temporizadores locales de un componente determinado. Por ejemplo:

```
type component MyMTCType
{
   var integer MyLocalInteger;
   timer MyLocalTimer;
   port MyMessagePortType PCO1
}
```

Estas declaraciones son visibles a todas las funciones que se ejecutan en el componente. Esto será indicado explícitamente utilizando la palabra clave **runs on** (véase la cláusula 16).

Las variables y temporizadores de componentes están asociados con el componente y siguen las reglas de alcance definidas en 5.1. Cada nuevo componente tendrá su propio conjunto de variables y temporizadores especificado en la definición de componente (incluidos cualesquiera valores iniciales, si se indican).

8.4.2 Definición de componentes con matrices de puertos

Es posible definir matrices de puerto en definiciones de tipos de componente (véase también 21.9). Por ejemplo:

```
type component My3pcoCompType
{
   port MyMessageInterfaceType PCO[3]
   // Define un tipo de componente que tiene una matriz de 3 puertos.
}
```

8.5 Direccionamiento de entidades dentro del SUT

Un SUT puede estar formado por varias entidades que tienen que ser direccionadas individualmente. El tipo de datos de dirección es un tipo que se ha de utilizar con operaciones de puertos para direccionar a entidades SUT. La representación de datos real de **address** es resuelta por una definición de tipo explícita dentro de la serie de pruebas o externamente por el sistema de prueba (es decir, el tipo **address** se deja como un tipo abierto dentro de la especificación de TTCN-3), lo que permite especificar casos de prueba abstractos independientemente de cualquier mecanismo de dirección real específico del SUT.

Las direcciones SUT explícitas sólo serán generadas dentro de un módulo TTCN-3 si el tipo es definido dentro del módulo. Si el tipo no es definido dentro del módulo, las direcciones SUT explícitas sólo serán introducidas como parámetros o recibidas en campos de mensaje o como parámetros de llamadas de procedimiento distante.

Además, se dispone del valor especial **null** para indicar una dirección no definida, por ejemplo, para la inicialización de variables del tipo de dirección.

Ejemplo:

```
// Asocia el tipo integer con la dirección de tipo abierto
type integer address;
 :
// Nueva variable de dirección inicializada con
var address MySUTentity := null;
:
// Que recibe un valor de dirección y lo asigna a la variable MySUTentity
PCO.receive(address:*) -> value MySUTentity;
 :
// Uso de la dirección recibida para enviar la plantilla MyResult
PCO.send(MyResult) to MySUTentity;
 :
// Uso de la dirección recibida para recibir una plantilla de confirmación
PCO.receive(MyConfirmation) from MySUTentity;
```

8.6 Referencias de componentes

Las referencias de componente son referencias únicas a los componentes de prueba creados durante la ejecución de un caso de prueba. Esta referencia de componente única es generada por el sistema de prueba cuando se crea un componente, es decir, una referencia de componente es el resultado de una operación **create** (véase 21.1). Además, las referencias de componente son devueltas por las funciones predefinidas **system** (que devuelve la referencia de componente para identificar los puertos de la interfaz del sistema de prueba), **mtc** (que devuelve la referencia de componente del MTC) y **self** (que devuelve la referencia del componente en el cual se invoca **self**).

Las referencias de componentes se utilizan en las operaciones de configuración **connect**, **map** y **start** (véase la cláusula 21) para establecer configuraciones de prueba y en las partes **from**, **to** y **sender** de operaciones de comunicación para fines de direccionamiento (véase la cláusula 22).

Además, se dispone del valor especial **null** para indicar una referencia de componente no definida (por ejemplo, para inicializar variables con el fin de tratar referencias de componentes).

La representación de datos reales de referencias de componentes será resuelta externamente por el sistema. Esto permite especificar casos de prueba abstractos independientemente de cualquier entorno de tiempo de ejecución TTCN-3 real, en otras palabras, TTCN-3 no restringe la implementación de un sistema de prueba con respecto al tratamiento e identificación de componentes de prueba.

NOTA – Una referencia de componente incluye información de tipo de componente. Esto significa, por ejemplo, que una variable para tratar referencias de componente debe utilizar el correspondiente nombre de tipo de componente en su declaración.

Ejemplo:

```
// Una definición de tipo de componente
type component MyCompType {
   port PortTypeOne PCO1;
   port PortTypeTwo PCO2
}
// Declaración de dos variables para tratar referencias a componentes de tipo
// MyCompType
// y crear un componente de este tipo
var MyCompType MyCompInst := MyCompType.create;

// Uso de referencias de componentes en operaciones de configuración siempre
// refiriéndose al componente creado anteriormente
connect(self:MyPCO1, MyCompInst:PCO1);
map(MyCompInst:PCO2, system:ExtPCO1);
MyCompInst.start(MyBehavior(self));  // self es pasado como un parámetro a
                                     // MyBehavior

// Uso de referencias de componentes en cláusulas from y to
MyPCO1.receive from MyCompInst;
 :
MyPCO2.receive(integer:*) -> sender MyCompInst;
 :
MyPCO1.receive(MyTemplate) from MyCompInst;
 :
MPCO2.send(integer:5) to MyCompInst;

// El siguiente ejemplo explica el caso de una conexión de uno a muchos en un
// puerto PCO1 donde se pueden recibir valores de tipo M1 de varios componentes
// de los diferentes tipos CompType1, CompType2 y CompType3 y donde hay que
// extraer el emisor. En este caso se puede usar el siguiente esquema:
:
var M1 MyMessage, MyResult;
var MyCompType1 MyInst1 := null;
var MyCompType2 MyInst2 := null;
var MyCompType3 MyInst3 := null;
 :
alt {
   [] PCO1.receive(M1:*) from MyCompType1 -> value MyMessage sender MyInst1 {}
   [] PCO1.receive(M1:*) from MyCompType2 -> value MyMessage sender MyInst2 {}
   [] PCO1.receive(M1:*) from MyCompType3 -> value MyMessage sender MyInst3 {}
}
 :

MyResult := MyMessageHandling(MyMessage);  // se extrae algún resultado de
                                           // una función

if (MyInst1 != null) {PCO1.send(MyResult) to MyInst1};
if (MyInst2 != null) {PCO1.send(MyResult) to MyInst2};
if (MyInst3 != null) {PCO1.send(MyResult) to MyInst3};
 :
```

8.7 Definición de la interfaz del sistema de prueba

La definición de tipo de componente se utiliza para definir la interfaz del sistema de prueba porque, teóricamente, las definiciones de tipo de componente y las definiciones de la interfaz del sistema de prueba tienen la misma forma (ambas son colecciones de puertos que definen posibles puntos de conexión).

```
type component MyISDNTestSystemInterface
{
   port MyBchannelInterfaceType   B1;
   port MyBchannelInterfaceType   B2;
   port MyDchannelInterfaceType   D1
}
```

En general, una referencia de tipo de componente que define la interfaz del sistema de prueba está asociada con cada caso de prueba. Los puertos de la interfaz del sistema de prueba son ejemplificados automáticamente junto con el MTC cuando comienza la ejecución del caso de prueba, es decir, cuando el caso de prueba es llamado desde la parte de control del módulo.

La operación que devuelve la dirección de la interfaz del sistema de prueba es **system**, y puede ser utilizada para direccionar a los puertos del sistema de prueba. Por ejemplo:

```
map(MyNewComponent:Port2, system:PCO1);
```

Cuando el MTC es el único componente que es ejemplificado durante la ejecución de la prueba, no hay que asociar una interfaz del sistema de prueba al caso de prueba. En este caso, la definición de tipo de componente asociada con el MTC define implícitamente la correspondiente interfaz del sistema de prueba.

9 Declaración de constantes

Las constantes pueden ser declaradas y utilizadas en encabezamientos de módulo, control de módulo, casos de prueba y funciones, y son indicadas por la palabra clave **const**. El valor de la constante será asignado en el punto de declaración. Por ejemplo:

```
const integer MyConst1 := 1;
const boolean MyConst2 := true, MyConst3 := false;
```

La asignación del valor a la constante se puede hacer dentro del módulo o externamente. En el segundo caso es una declaración de constante externa indicada por la palabra clave **external**. Las constantes externas se resolverán a un valor en el tiempo de compilación. Por ejemplo:

```
external const integer MyExternalConst; // declaración de constante externa
```

Una constante externa puede tener un tipo arbitrario pero el tipo tiene que ser conocido en el módulo, es decir, un tipo básico, definido en el módulo o importado de algún otro módulo. La correspondencia del tipo con la representación externa de una constante externa está también fuera del ámbito de la presente Recomendación, al igual que el mecanismo de cómo se introduce el valor de una constante externa en un módulo.

10 Declaración de variables

Las variables son indicadas por la palabra clave **var**. Las variables pueden ser declaradas y utilizadas en control de módulo, casos de prueba y funciones. No serán declaradas ni utilizadas en un encabezamiento de módulo (es decir, TTCN-3 no soporta variables globales). Una declaración de variable puede tener asignado un valor inicial facultativo. Por ejemplo:

```
var integer MyVar1 := 1;
var boolean MyVar2 := true, MyVar3 := false;
```

La utilización de variables no inicializadas en el tiempo de ejecución originarán un error de caso de prueba.

11 Declaración de temporizadores

Los temporizadores pueden ser declarados y utilizados en control de módulo, casos de prueba y funciones y no serán declarados ni utilizados en la parte de definiciones del módulo. Una declaración de temporizador puede tener asignado un valor de duración por defecto facultativo. El temporizador será arrancado con este valor si no se especifica ningún otro. Este valor será del tipo **`float`** cuando la unidad básica es segundos. Por ejemplo:

```
timer MyTimer1 := 5E-3;   // Declaración del temporizador MyTimer1 con el valor
                          // por defecto de 5 ms

timer MyTimer2;           // Declaración de MyTimer2 sin un valor por defecto,
                          // es decir, hay que asignar un valor cuando se
                          // arranca el temporizador
```

Las operaciones de temporizador **`start`**, **`stop`**, **`read`** y **`timeout`** pueden ser utilizadas para controlar temporizadores (véase la cláusula 23). Por ejemplo:

```
// Los usos de MyTimer2 pudieran ser
MyTimer2.start(10);    // 10 segundos
MyTimer2.start(180);   // 3 minutos
```

11.1 Temporizadores como parámetros

Los temporizadores sólo pueden ser introducidos por referencias a funciones y alternativas denominadas. Los temporizadores introducidos en una función o alternativa denominada son conocidos dentro de la definición de comportamiento de la función o de la alternativa denominada.

Un temporizador introducido como parámetro de referencia puede ser utilizado como cualquier otro temporizador, es decir, no tiene que ser declarado. Un temporizador arrancado puede ser introducido también en una función o alternativa denominada. El temporizador continúa su ejecución, es decir, no se detiene implícitamente. En consecuencia, los posibles eventos de temporización pueden ser tratados dentro de la función o alternativa denominada a la cual se ha pasado el temporizador.

Ejemplo:

```
// Definición de función con un temporizador en la lista de parámetros formales
function MyBehaviour (timer MyTimer)
{  :
   MyTimer.start;
   :
}
```

12 Declaración de mensajes

Uno de los elementos clave de TTCN-3 es la capacidad de enviar y recibir mensajes complejos por los puertos de comunicación definidos por la configuración de prueba. Estos mensajes pueden ser los que se relacionan explícitamente con la prueba del SUT o con los mensajes de coordinación interna y de control específicos de la configuración de prueba pertinente.

NOTA – En TTCN-2 estos mensajes son las primitivas de servicio abstractas (ASP, *abstract service primitives*), las unidades de datos de protocolo (PDU) y los mensajes de coordinación. El lenguaje núcleo de TTCN-3 es genérico en el sentido de que no hace distinciones sintácticas o semánticas de esta clase.

Los mensajes complejos pueden ser definidos como tipos record (véase 6.3.1). Por ejemplo:

```
type record MyMessageType
{
   FieldType1  field1,
```

```
   FieldType2  field2,
   :
   FieldTypeN  fieldN
}
```

Naturalmente, los mensajes pueden estar subestructurados, por ejemplo:

```
// Elemento de información tipo 1 (IEType1). Declaraciones similares para
IEType2 a IETypeN
type record IEType1
{
   IEFieldType1  iefield1,
   IEFieldType2  iefield2,
   :
   IEFieldTypeN  iefieldN
}

// Un mensaje que contiene elementos de información
type record MyMessageType
{
   IEType1  field1,
   IEType2  field2,
   :
   IETypeN field3
}
```

12.1 Campos de mensaje facultativos

Por defecto, todos los campos en un mensaje serán obligatorios. Se especificarán campos de mensaje facultativos utilizando la palabra clave **`optional`**. Por ejemplo:

```
type record MyMessageType
{
   FieldType1  field1,
   FieldType2  field2   optional,
   :
   FieldTypeN  fieldN
}
```

13 Declaración de firmas de procedimiento

Se necesitan firmas de procedimiento (o firmas, para abreviar) para comunicación síncrona. Un procedimiento puede ser invocado en el SUT (es decir, el sistema de prueba ejecuta la llamada) o invocado en el sistema de prueba (es decir, el SUT ejecuta la llamada).

Para los procedimientos llamados desde el SUT y los llamados desde el sistema de prueba, la **`signature`** del procedimiento completo se definirá en el módulo TTCN-3.

Dentro de la definición de **`signature`**, la lista de parámetros puede incluir identificadores de parámetros, tipos de parámetros y su sentido, es decir, **`in`**, **`out`**, o **`inout`**). Obsérvese que el sentido de los parámetros es visto por la parte *llamada* más bien que por la parte *llamante*. Por ejemplo:

```
signature MyRemoteProc (in integer Par1, out float Par2, inout integer Par3)
   return integer;
// Esto define el procedimiento distante MyRemoteProc. MyRemoteProc devuelve un
// valor entero y tiene tres parámetros: un parámetro in de tipo integer, un
// parámetro out de tipo float y un parámetro inout de tipo integer
```

El procedimiento **`call`** resultará en que la parte llamada ejecuta una **`reply`** (el caso normal) o plantea una excepción. Las acciones resultantes de una llamada de procedimiento aceptada son definidas por la parte receptora (véase la cláusula 22).

13.1 Omisión de parámetros reales

Se permite omitir parámetros reales de una lista de parámetros reales de firma. Esto se indica representando el parámetro real omitido en su posición correcta utilizando la palabra clase **omit**. Por ejemplo:

```
ParameterList(Par1, omit, Par3) // Par2 se omite
```

NOTA – Esto es necesario a menudo cuando se utilizan firmas de procedimiento en comunicación síncrona.

13.2 Especificación de excepciones

Las excepciones se representan en un TTCN-3 como valores de un tipo específico, incluso es posible utilizar plantillas y mecanismos de concordancia.

NOTA – La conversión de excepciones generada por el SUT en el tipo correspondiente es específico de la herramienta y del sistema, por lo que está fuera del ámbito de la presente Recomendación.

Las excepciones se definen en forma de una lista de excepciones incluida en la definición de firma. Esta lista define todos los posibles tipos diferentes asociados con el conjunto de posibles excepciones (el significado de las propias excepciones sólo se suele distinguir representándolas mediante valores específicos de estos tipos.

Ejemplo:

```
signature MyRemoteProc (in integer Par1, out float Par2, inout integer Par3)
   return integer exception(ExceptionType1, ExceptionType2);

// Una llamada de MyRemoteProc puede plantear excepciones de tipo ExceptionType1
// o excepciones ExceptionType2
```

14 Declaración de plantillas

Se utilizan plantillas para transmitir un conjunto de valores distintos o para probar si un conjunto de valores recibidos concuerda con la especificación de plantillas.

Las plantillas proporcionan las siguientes posibilidades:

a) son una manera de organizar y reutilizar datos de prueba, incluida una forma sencilla de herencia;

b) pueden ser parametrizadas;

c) permiten mecanismos de concordancia;

d) pueden ser utilizadas con otras comunicaciones basadas en mensajes o basadas en procedimiento.

Dentro de una plantilla, se pueden especificar valores, gamas y atributos de concordancia y utilizarlos en comunicaciones basadas en mensaje y basadas en procedimiento. Se pueden especificar plantillas para cualquier tipo o firma de procedimiento de TTCN-3. Las plantillas basadas en tipos se utilizan para comunicaciones basadas en mensaje y las plantillas de firma se utilizan en comunicaciones basadas en procedimiento.

14.1 Declaración de plantillas de mensaje

Los mensajes con valores reales pueden ser especificados utilizando plantillas. Cabe considerar que una plantilla es un conjunto de instrucciones para construir un mensaje que se ha de enviar o para concordar un mensaje recibido.

Las plantillas pueden ser especificadas para cualquier tipo TTCN-3 definido en el cuadro 3, salvo para los tipos especiales (**port**, **component**, **address**).

```
// Cuando se usa en una operación receptora, esta plantilla concordará con
// cualquier valor integer
template integer Mytemplate := *;
// Esta plantilla sólo concordará con los valores enteros 1, 2 ó 3
template integer Mytemplate := (1, 2, 3);
```

No obstante, se prevé que el uso más generalizado será con registros, como se muestra en los ejemplos de las cláusulas siguientes.

14.1.1 Plantillas para enviar mensajes

Una plantilla utilizada en una operación **send** define un conjunto completo de valores de campo que comprenden el mensaje que se ha de transmitir por un puerto de prueba. Al ejecutar la operación **send**, la plantilla estará totalmente definida, es decir, todos los campos se resolverán a valores reales y no se utilizará ningún mecanismo de concordancia en los campos de plantilla, directa ni indirectamente.

Ejemplo:

```
// Dada la definición de mensaje
type record MyMessageType
{
   integer  field1,
   charstring field2,
   boolean  field3
}

// una plantilla de mensaje podría ser
template MyMessageType MyTemplate:=
{
   field1 := 1,
   field2 := "My string",
   field3 := true
}
// y una operación send correspondiente podría ser
MyPCO.send(MyTemplate);
```

NOTA – Se pueden utilizar también plantillas para excepciones si se ha definido el tipo correspondiente.

14.1.2 Plantillas para recibir mensajes

Una plantilla utilizada en una operación **receive** define una plantilla de datos con la cual ha de concordar un mensaje entrante. Los mecanismos de concordancia, definidos en el anexo C, pueden ser utilizados en plantillas de recepción. No se producirá ninguna vinculación de los valores entrantes con la plantilla.

Ejemplo:

```
// Dada la definición de mensaje
type record MyMessageType
{
   integer    field1,
   charstring field2,
   boolean    field3
}

// Una plantilla de mensaje pudiera ser
template MyMessageType MyTemplate:=
{
   field1 := 1,
   field2 := pattern "abc*xyz",
   field3 := true
}
```

```
// y una operación receive correspondiente podría ser
MyPCO.receive(MyTemplate);
```

14.2 Declaración de plantillas de firma

Es posible especificar listas de parámetros de procedimiento utilizando plantillas. Éstas serán definidas por cualquier procedimiento mediante referencia a la definición de firma asociada.

Ejemplo:

```
// definición de firma para un procedimiento distante
signature RemoteProc(in integer Par1, out integer Par2, inout integer Par3)
   return integer;
// plantillas de ejemplo asociadas con firma de procedimiento definida
template RemoteProc Template1:=
{
   Par1 :=  1,
   Par2 :=  2,
   Par3 :=  3
}

template RemoteProc Template2:=
{
   Par1 := 1,
   Par2 := *,
   Par3 := 3
}

template RemoteProc Template3:=
{
   Par1 := 1,
   Par2 := *,
   Par3 := *
}
```

14.2.1 Plantillas para llamada de procedimientos

Una plantilla utilizada en una operación **call** o **reply** define un conjunto completo de valores de campo para todos los parámetros **in** e **inout**. Cuando se ejecuta la operación **call**, todos los parámetros **in** e **inout** en la plantilla se resolverán a valores reales, no se utilizarán mecanismos de concordancia en estos campos, directa ni indirectamente. Se pasa sencillamente por alto toda especificación de plantilla para parámetros **out**, por lo que se permite especificar mecanismos de concordancia para estos campos u omitirlos (véase el anexo C).

Ejemplo:

```
// Llamada válida pues todos los parámetros in e inout tienen un valor distinto
MyPCO.call(RemoteProc:Template1);

// Llamada válida pues todos los parámetros in e inout tienen un valor distinto
MyPCO.call(RemoteProc:Template2);

// Llamada no válida pues los parámetros Par3 tienen un atributo concordante no
// un valor
MyPCO.call(RemoteProc:Template3);

// Las plantillas nunca devuelven valores. En el caso de Par2 y Par3 los valores
// devueltos por la llamada deben ser extraídos usando una cláusula de
// asignación al final del enunciado de llamada
```

14.2.2 Plantillas para aceptar llamadas de procedimiento

Una plantilla utilizada en una operación **getcall** define una plantilla de datos con la cual se comparan los campos de parámetros entrantes. Los mecanismos de concordancia, definidos en el

anexo C pueden ser utilizados en cualquier plantilla empleada por esta operación. No se producirá ninguna vinculación de valores entrantes con la plantilla. Cualquier parámetro **in** será pasado por alto en el proceso de concordancia.

Ejemplo:

```
// getcall, válida, concordará si Par2 == 2 y Par3 == 3
MyPCO.getcall(RemoteProc:Template1);

// getcall, válida, concordará si Par3 == 3 y Any value de Par2
MyPCO.getcall(RemoteProc:Template2);

// getcall valida, concordará en Any value de Par3 y Par2
MyPCO.getcall(RemoteProc:Template3);
```

14.3 Mecanismos de concordancia de plantillas

En general, se utilizarán mecanismos de concordancia para sustituir valores de campos de plantillas o para sustituir incluso todo el contenido de una plantilla. Es posible utilizar algunos de estos mecanismos combinados.

Cabe también utilizar mecanismos de concordancia y comodines en línea en sólo en eventos recibidos (es decir, operaciones **receive**, **getcall**, **getreply** y **catch**). Pueden aparecer en valores explícitos, por ejemplo:

```
MyPCO.receive(charstring:"abcxyz");
MyPCO.receive (integer:complement(1, 2, 3));
```

El tipo de identificador es facultativo, por ejemplo:

```
MyPCO.receive("abcxyz");
```

Sin embargo, el tipo de plantilla en línea estará en la lista de puertos por los cuales se recibe la plantilla. Cuando hay una ambigüedad (es decir, mediante subtipificación), se incluirá el nombre de tipo en el enunciado de recepción.

Los mecanismos de concordancia se clasifican en cuatro grupos:

a) valores específicos (es decir, una expresión que da un valor específico);

b) símbolos especiales que pueden ser utilizados *en vez* de valores:
   - **(...)**: una lista de valores;
   - **complement (...)**: complemento de una lista de valores;
   - **omit**: se omite el valor;
   - **?**: comodín para cualquier valor;
   - *****: comodín para cualquier valor o ningún valor (es decir, un valor omitido)
   - **(**inferior **a** superior**)**: una gama de valores enteros entre las fronteras inferior y superior, incluidas éstas;

c) símbolos especiales que pueden ser utilizados *dentro* de valores:
   - **?**: comodín para cualquier elemento en una cadena, matriz, **record of** o **set of**;
   - *****: comodín para cualquier número de elementos consecutivos en una cadena, matriz, **record of** o **set of**, o ningún elemento (es decir, un elemento omitido);

d) símbolos especiales que describen *atributos* de valores:
   - **length**: restricción para cadenas y matrices;
   - **ifpresent**: para concordancia de valores de campos facultativos (si no se omite).

Los mecanismos de concordancia soportados y sus símbolos asociados (si hubiere alguno) y el alcance de su aplicación se muestran en el cuadro 5. La columna de la izquierda de este cuadro muestra todos los tipos equivalentes de TTCN-3 y ASN.1 definidos en la serie de Recomendaciones UIT-T X.680 [7], [8], [9] y [10] a los cuales se aplican estos mecanismos de concordancia. En el anexo C figura una descripción completa de cada mecanismo de concordancia.

Cuadro 5/Z.140 – Mecanismo de concordancia de TTCN-3

Utilizados con valores de	Valor	En vez de valores								Atributos	
	Valor específico	Lista de valores	Lista complementada	Omitir valor	Cualquier valor (?)	Cualquier valor o ninguno (*)	Gama	Cualquier elemento (?)	Cualquier elemento o ninguno (*)	Restricción de longitud	Si está presente
boolean	Sí	Sí	Sí	Sí	Sí	Sí					Sí
integer	Sí	Sí	Sí	Sí	Sí	Sí	Sí				Sí
char	Sí	Sí	Sí	Sí	Sí	Sí	Sí				Sí
universal char	Sí	Sí	Sí	Sí	Sí	Sí	Sí				Sí
float	Sí	Sí	Sí	Sí	Sí	Sí					Sí
bitstring	Sí	Sí	Sí	Sí	Sí	Sí		Sí	Sí	Sí	Sí
octetstring	Sí	Sí	Sí	Sí	Sí	Sí		Sí	Sí	Sí	Sí
hexstring	Sí	Sí	Sí	Sí	Sí	Sí		Sí	Sí	Sí	Sí
character strings	Sí	Sí	Sí	Sí	Sí	Sí		Sí	Sí	Sí	Sí
record	Sí	Sí	Sí	Sí	Sí	Sí					Sí
record of	Sí	Sí	Sí	Sí	Sí	Sí		Sí	Sí	Sí	Sí
array	Sí	Sí	Sí	Sí	Sí	Sí		Sí	Sí	Sí	Sí
set	Sí	Sí	Sí	Sí	Sí	Sí					Sí
set of	Sí	Sí	Sí	Sí	Sí	Sí		Sí	Sí	Sí	Sí
enumerated	Sí	Sí	Sí	Sí	Sí	Sí					Sí
union	Sí	Sí	Sí	Sí	Sí	Sí					Sí

14.4 Parametrización de plantillas

Las plantillas para las operaciones de envío y recepción pueden ser parametrizadas. Los parámetros formales de una plantilla pueden incluir plantillas, funciones y símbolos de concordancia especiales. Las reglas para las listas de parámetros formales y reales se conformarán con las definidas en 5.3.

Ejemplo:

```
// La plantilla
template MyMessageType MyTemplate (integer MyFormalParam):=
{
   field1 := MyFormalParam,
   field2 := pattern "abc*xyz",
   field3 := true
}
// se podría usar como sigue
pco1.send(MyTemplate(123));
```

14.4.1 Parametrización con atributos concordantes

Para poder pasar atributos concordantes como parámetros se añadirá la palabra clave suplementaria **template** antes del campo de tipo. Esto convierte el parámetro en una plantilla y amplía en efecto los parámetros permitidos para el tipo asociado con el fin de incluir el conjunto apropiado de atributos concordantes (véase el anexo C), así como el conjunto normal de valores. Los campos de parámetros de plantilla no serán invocados por referencia.

Ejemplo:

```
// La plantilla
template MyMessageType MyTemplate (template integer MyFormalParam):=
{  field1 := MyFormalParam,
   field2 := pattern "abc*xyz",
   field3 := true
}

// se podría usar como sigue
pco1.receive(MyTemplate(?));
// O, si field1 tiene que ser definido como facultativo
pco1.receive(MyTemplate(omit));
```

14.5 Paso de plantillas como parámetros

Sólo las definiciones **function**, **testcase**, **named alt** y **template** pueden tener plantillas como parámetros formales.

Ejemplo:

```
function MyBehaviour(template MyMsgType MyFormalParameter)
runs on MyComponentType
{  :
   pco1.receive(MyFormalParameter);
   :
}
```

14.6 Plantillas modificadas

Normalmente una plantilla especificará un conjunto de valores básicos, o supletorios, o símbolos concordantes para cada campo definido en la definición apropiada. Cuando se necesitan pequeños cambios para especificar una nueva plantilla, es posible especificar una plantilla modificada, que especifica las modificaciones de determinados campos de la plantilla original, directa o indirectamente.

La palabra clave **modifies** indica la plantilla progenitora de la cual se derivará la plantilla nueva o modificada. Esta plantilla progenitora puede ser la plantilla original o una plantilla modificada.

Las modificaciones se producen de manera vinculada, siguiendo la plantilla original. Si se especifica un campo de plantilla y su valor correspondiente o símbolo concordante en la plantilla modificada, el valor especificado o símbolo concordante sustituye al especificado en la plantilla progenitora. Si no se especifica un campo de plantilla ni su valor correspondiente o símbolo concordante en la plantilla modificada, se utilizará el valor o símbolo concordante de la plantilla progenitora.

Una plantilla modificada no hará referencia a sí misma, directa ni indirectamente, es decir, no se admite derivación recursiva.

Ejemplo:

```
// Dado
template MyRecordType MyTemplate1 :=
{
```

```
   field1 := 123,
   field2 := "A string",
   field3 := true
}

// escribir
template MyRecordType MyTemplate2 modifies MyTemplate1 :=
{
   field2 := "A modified string",
   field3 := omit   // field3 se debe especificar como facultativo en el tipo
                    // de registro correspondiente
}
// es igual que escribir
template MyRecordType MyTemplate2 :=
{
   field1 := 123,
   field2 := "A modified string",
   field3 := omit
}
```

14.6.1 Parametrización de plantillas modificadas

Si una plantilla básica tiene una lista de parámetros formales, se aplican las siguientes reglas a todas las plantillas modificadas derivadas de esa plantilla básica, se hayan derivado o no en uno o varios pasos de modificación:

a) la plantilla derivada no omitirá parámetros, aunque una plantilla derivada puede tener parámetros adicionales (anexados), si se desea;

b) la lista de parámetros formales seguirá el nombre de plantilla para cada plantilla modificada;

c) las plantillas parametrizadas en campos de plantillas no serán modificadas ni omitidas explícitamente en una plantilla modificada.

Ejemplo:

```
// Dado

template MyRecordType MyTemplate1(integer Mypar):=
{
   field1 := MyPar,
   field2 := "A string",
   field3 := true
}
// una modificación podría ser
   template MyRecordType MyTemplate2(integer MyPar) modifies MyTemplate1 :=
{  // field1 is parameterized in Template1
   field2 := "A modified string",
   field3 := omit   // field3 se debe especificar como facultativo en el tipo
                    // de registro correspondiente
}
```

14.6.2 Plantillas modificadas en línea

Así como la creación de constricciones modificadas denominadas explícitamente, TTCN-3 permite la definición de constricciones modificadas en línea.

Ejemplo:

```
// Dado
template MyMessageType Setup :=
{  field1 := 75,
   field2 := "abc",
```

```
   field3 := true
}

// Se podría utilizar para definir una plantilla modificada en línea de Setup
pco1.send (modifies Setup := {field1 76});
```

14.7 Cambio de campos de plantilla

Todos los cambios de campos de plantilla sólo se efectuarán mediante parametrización o mediante plantillas derivadas en línea en el momento de ejecutar una operación de comunicación (por ejemplo, **send**, **receive**, **call**, **getcall**, etc.). Los efectos de estos cambios en el valor del campo de plantilla no persisten en la plantilla subsiguiente al evento de comunicación correspondiente.

La notación de la clase *MyTemplateId.Fieldid* no se utilizará para fijar o extraer valores en plantilla en eventos de comunicación. Para este fin se utilizará el símbolo "->" (véase la cláusula 22).

14.8 Operación Match

La operación **match** permite comparar el valor de una variable con una plantilla. La operación devuelve un valor booleano. Si el tipo de la plantilla y la variable no son compatibles, la operación devuelve falso. Si los tipos son compatibles, el valor devuelto de la operación indica si el valor de la variable se conforma con la plantilla especificada.

```
template integer LessThan10 := (1..10);

testcase TC001()
runs on MyMTCType
{
   var integer RxValue;
   ...
   PCO1.receive(integer:?) -> value RxValue;

   if(match( RxValue, LessThan10)) { ... }
   ...
}
```

14.9 Operación Valueof

La operación **valueof** permite asignar el valor especificado dentro de una plantilla a los campos de una variable. La variable y la plantilla serán compatibles en tipos (véase 6.7) y cada campo de la plantilla se resolverá en un solo valor.

```
type record ExampleType
{
 integer field1,
 boolean field2
}

   template ExampleType SetupTemplate :=
   {
      field1 := 1,
      field2 := true
   }

   ...
      var ExampleType RxValue := valueof( SetupTemplate);
   ...
```

15 Operadores

TTCN-3 soporta varios operadores predefinidos que pueden ser utilizados en los términos de expresiones de TTCN-3. Los operadores predefinidos se dividen en siete categorías:

a) operadores aritméticos;

b) operadores de cadena;

c) operadores relacionales;

d) operadores lógicos;

e) operadores para bits;

f) operadores de cambio;

g) operadores de rotación.

Estos operadores se enumeran en el cuadro 6.

Cuadro 6/Z.140 – Lista de operadores de TTCN-3

Categoría	Operador	Símbolo o palabra clave
Operadores aritméticos	adición	`+`
	substracción	`-`
	multiplicación	`*`
	división	`/`
	módulo	`mod`
	residuo	`rem`
Operadores de cadena	concatenación	`&`
Operadores relacionales	igual	`==`
	menor que	`<`
	mayor que	`>`
	no igual	`!=`
	mayor que o igual	`>=`
	menor que o igual	`<=`
Operadores lógicos	no lógico	`not`
	y lógica	`and`
	o lógica	`or`
	o exclusiva lógica	`xor`
Operadores de bits	no para bits	`not4b`
	y para bits	`and4b`
	o para bits	`or4b`
	o exclusiva para bits	`xor4b`
Operadores de cambio	cambio izquierda	`<<`
	cambio derecha	`>>`
Operadores de rotación	rotación izquierda	`<@`
	rotación derecha	`@>`

La precedencia de estos operadores se muestra en el cuadro 7. Dentro de cada fila del cuadro, los operadores enumerados tienen igual precedencia. Si en una expresión aparece más de un operador de igual precedencia, las operaciones son evaluadas de izquierda a derecha. Se puede utilizar paréntesis para agrupar operandos en expresiones, en cuyo caso una expresión entre paréntesis tienen la precedencia más alta para evaluación.

Cuadro 7/Z.140 – Precedencia de operadores

Prioridad	Tipo de operador	Operador
Más alta		`( ... )`
	Unario	`+, –, not, not4b`
	Binario	`*, /, mod, rem`
		`+, –`
		`<<, >>, <@, @>`
		`<, >, <=, >=`
		`==, !=`
		`and4b`
		`xor4b`
		`or4b`
		`and`
		`xor`
		`or`
Más baja		`&`

15.1 Operadores aritméticos

Los operadores aritméticos representan las operaciones de adición, sustracción, multiplicación, división y módulo. Los operandos de estos operadores serán de tipo **`integer`** (incluidas derivaciones de **`integer`**) o **`float`** (incluidas derivaciones de **`float`**), salvo para **`mod`** que se utilizará solamente con tipos **`integer`** (incluidas derivaciones de **`integer`**).

Con tipos **`integer`** el tipo de resultado de operaciones aritméticas es **`integer`**. Con tipos **`float`** el resultado del tipo de operaciones aritméticas es **`float`**.

En el caso cuando se utiliza (**`+`**) o menos (**`–`**) como el operador unario se aplican también las reglas para operandos. El resultado de utilizar el operador menos es el valor negativo del operando si éste era positivo y viceversa.

El resultado de ejecutar la operación de división (**`/`**) en dos:

a) valores **`integer`** da el valor **`integer`** completo resultante de dividir el primer **`integer`** por el segundo (es decir, se descartan las fracciones);

b) valores **`float`** da el valor **`float`** resultante de dividir el primer **`float`** por el segundo (es decir, no se descartan las fracciones).

Los operadores **`rem`** y **`mod`** computan operandos de tipo **`integer`** y tienen un resultado de tipo **`integer`**. Las operaciones `x` **`rem`** `y` y `x` **`mod`** `y` calculan el resto que queda de una división de enteros de `x` por `y`. Por tanto, sólo se definen para operandos no cero `y`. Para `x` e `y`, `x` **`rem`** `y` y `x` **`mod`** `y` tienen el mismo resultado pero difieren para argumentos negativos.

Formalmente, **mod** y **rem** se definen como sigue:

```
x rem y  = x - y * (x/y)
x mod y  = x rem |y|          si x >= 0
         = 0                  si x < 0 y  x rem |y| = 0
         = y + x rem |y|      si x < 0 y  x rem |y| < 0
```

El cuadro 8 ilustra la diferencia entre los operadores **mod** y **rem**:

Cuadro 8/Z.140 – Efecto de los operadores mod y rem

x	–3	–2	–1	0	1	2	3
x mod 3	0	1	2	0	1	2	0
x rem 3	0	–2	–1	0	1	2	0

15.2 Operadores de cadena

Los operadores relacionales predefinidos ejecutan concatenación de tipos de cadena. Los operandos pueden ser cualesquiera valores de tipos de cadena que sean compatibles. La operación es una simple concatenación de izquierda a derecha. No implica ninguna forma de adición aritmética. El tipo de resultado es el tipo de cadena compatible, por ejemplo:

```
'1111'B & '0000'B & '1111'B da '111100001111'B
```

15.3 Operadores relacionales

Los operadores relacionales predefinidos representan las relaciones de igualdad, menor que, mayor que, no igual a, mayor que o igual a y menor que o igual a. Los operandos de igualdad (**==**) y de no igualdad (**!=**) pueden ser de un tipo arbitrario. Todos los demás operadores relacionales sólo tendrán operandos de tipo **integer** (incluidas derivaciones de **integer**) o **float** (incluidas derivaciones de **float**). En todos los casos, los dos operandos serán de tipo compatible. El tipo de resultado de estas operaciones es **boolean**.

15.4 Operadores lógicos

Los operadores **boolean** predefinidos ejecutan las operaciones de negación, **and** lógica, **or** lógica y **xor** lógica. Sus operandos serán de tipo **boolean**. El tipo de resultado de los operadores lógicos es **boolean**.

not lógico es el operador unario que devuelve el valor **true** si su operando era de valor **false** y devuelve el valor **false** si el operando era de valor **true**.

and lógica devuelve el valor **true** si sus operandos son **true**; en los demás casos, devuelve el valor **false**.

or lógica devuelve el valor **true** si por lo menos uno de sus operandos es **true**; devuelve el valor **false** solamente si ambos operandos son **false**.

xor lógica devuelve el valor **true** si uno de sus operandos es **true**; devuelve el valor **false** si ambos operandos son **false** si ambos operandos son **true**.

15.5 Operadores para bits

Los operadores para bits predefinidos ejecutan las operaciones de bits **not** para bits, **and** para bits, **or** para bits y **xor** para bits. Estos operadores se conocen como **not4b**, **and4b**, **or4b** y **xor4b** respectivamente.

NOTA – Se ha de leer como "no para bits", "y para bits", etc.

Sus operandos serán de tipo **bitstring**, **hexstring**, **octetstring**. El tipo de resultado de los operadores para bits será igual que el de los operandos.

El operador unario **not4b** para bits invierte los valores de bits individuales de su operando. Para cada bit en el operando, un bit 1 se pone a 0 y un bit 0 se pone a 1. Es decir:

```
not4b  '1'B  da  '0'B
not4b  '0'B  da  '1'B
```

Ejemplo:

```
not4b  '1010'B   da    '0101'B
not4b  '1A5'H    da    'E5A'H
not4b  '01A5'O   da    'FE5A'O
```

El operador **and4b** para bits acepta dos operandos. Para cada posición de bit correspondiente, el valor resultante es 1 si ambos bits están puestos a 1; en los demás casos, el valor para el bit resultante es 0. Es decir:

```
'1'B and4b '1'B da  '1'B
'1'B and4b '0'B da  '0'B
'0'B and4b '1'B da  '0'B
'0'B and4b '0'B da  '0'B
```

Ejemplo:

```
'1001'B  and4b '0101'B da  '0001'B
'B'H     and4b '5'H    da  '1'H
'FB'O    and4b '15'O   da  '11'O
```

El operador **or4b** para bits acepta dos operandos. Para cada posición de bit correspondiente, el valor resultante es 0 si ambos bits están puestos a 0; en los demás casos, el valor para el bit resultante es 1. Es decir:

```
'1'B or4b '1'B da  '1'B
'1'B or4b '0'B da  '1'B
'0'B or4b '1'B da  '1'B
'0'B or4b '0'B da  '0'B
```

Ejemplo:

```
'1001'B  or4b '0101'B  da  '1101'B
'9'H     or4b '5'H     da  'D'H
'A9'O    or4b 'F5'O    da  'FD'O
```

El operador **xor4b** para bits acepta dos operandos. Para cada posición de bit correspondiente, el valor resultante es 0 si ambos bits están puestos a 0 o si ambos bits están puestos a 1; en los demás casos, el valor para el bit resultante es 0. Es decir:

```
'1'B xor4b '1'B da  '0'B
'0'B xor4b '0'B da  '0'B
'0'B xor4b '1'B da  '1'B
'1'B xor4b '0'B da  '1'B
```

Ejemplo:

```
'1001'B  xor4b '0101'B da  '1100'B
'9'H     xor4b '5'H    da  'C'H
'39'O    xor4b '15'O   da  '2C'O
```

15.6 Operadores de cambio

Los operadores de cambio predefinidos ejecutan los operadores de cambio izquierda (**<<**) y cambio derecha (**>>**). Su operando de izquierda será de tipo **bitstring**, **hexstring**, **octetstring** o

integer. Su operando de derecha será de tipo **integer**. El tipo de resultado de estos operadores será igual que el del operando izquierdo.

Los operadores de cambio se comportan diferentemente de acuerdo con el tipo de su operando izquierdo. Si el tipo del operando izquierdo es:

a) **bitstring** o **integer** la unidad de cambio aplicada es 1 bit;

b) **hexstring** la unidad de cambio aplicada es un dígito hexadecimal;

c) **octetstring** la unidad de cambio aplicada es 1 octeto.

El operador de cambio izquierda (**<<**) acepta dos operandos. Cambia el operando de izquierda por el número de unidades de cambio a la izquierda especificadas por el operando de la derecha. Se descartan las unidades de cambio en exceso (bits, dígitos hexadecimales u octetos). Para cada unidad de cambio cambiada a la izquierda, se inserta un cero ('0'B, '0'H, o '00'O determinados de acuerdo con el tipo del operando de la izquierda) del lado derecho del operando izquierdo.

NOTA 1 – Si el operando de la izquierda es de tipo **integer**, para cada bit cambiado a la izquierda, esto es equivalente a multiplicar el operando de la izquierda por dos.

NOTA 2 – Se asignará un veredicto de error si se produce un desbordamiento que depende del sistema al aplicar la operación cambio izquierda al operando de la izquierda.

Ejemplo:

```
'111001'B  << 2 da  '100100'B
'12345'H   << 2 da  '34500'H
'1122334455'O  << (1+1)  da  '3344550000'O
32  << 2  da  128
−32 << 2 da  −128
```

El operador de cambio derecha (**>>**) acepta dos operandos. Cambia el operando de la izquierda por el número de unidades de cambio a la derecha especificadas por el operando de la derecha. Se descartan las unidades de cambio en exceso (bits, dígitos hexadecimales u octetos. Para cada unidad de cambio cambiada a la derecha, se inserta un 0 ('0'B, '0'H, o '00'O determinados de acuerdo con el tipo del operando de la izquierda) del lado izquierdo del operando izquierdo.

NOTA 3 – Si el operando de la izquierda es de tipo **integer**, para cada bit cambiado a la derecha, esto es equivalente a efectuar división de enteros del operando de la izquierda por dos (2).

NOTA 4 – Cuando el operando izquierdo es de tipo **integer** y su valor es negativo, al efectuarse un cambio a la derecha, el bit de signo será propagado.

Ejemplo:

```
'111001'B  >> 2 da  '001110'B
'12345'H   >> 2 da  '00123'H
'1122334455'O  >> (1+1)  da  '0000112233'O
32  >> 2 da  8
-32 >> 2 da  −8
```

15.7 Operadores de rotación

Los operadores de rotación predefinidos ejecutan los operadores de rotación a la izquierda (**<@**) y de rotación a la derecha (**@>**). Su operando de la izquierda será de tipo **bitstring**, **hexstring**, **octetstring**, **charstring or universal charstring**. Su operando de la derecha será de tipo **integer**. El tipo de resultado de estos operadores será igual que el del operando de la izquierda.

Los operadores de rotación se comportan diferentemente de acuerdo con el tipo de su operando de la izquierda. Si el tipo del operando de la izquierda es:

a) **bitstring**, la unidad de rotación aplicada es 1 bit;

b) **hexstring**, la unidad de rotación aplicada es un dígito hexadecimal;

c) **octetstring**, la unidad de rotación aplicada es 1 octeto;

d) **charstring** o **universal charstring**, la unidad de rotación aplicada es 1 carácter.

El operador de rotación izquierda (**<@**) acepta dos operandos. Rota el operando de la izquierda por el número de unidades de cambio a la izquierda especificada por el operando de la derecha. Las unidades de cambio en exceso (bits, dígitos hexadecimales, octetos o caracteres) son reinsertadas en el operando de la izquierda de su lado derecho.

Ejemplo:

```
'101001'B  <@ 2 da  '100110'B
'12345'H   <@ 2 da  '34512'H
'1122334455'O  <@ (1+2)  da  '4455112233'O
"abcdefg"  <@ 3 da  "defgabc"
```

El operador de rotación derecha (**@>**) acepta dos operandos. Rota el operando de la izquierda por el número de unidades de cambio a la derecha especificadas por el operando de la derecha. Las unidades de cambio en exceso (bits, cifras hexadecimales, octetos o caracteres) son reinsertadas en el operando de la izquierda de su lado izquierdo.

Ejemplo:

```
'100001'B  @> 2 da  '0110001'B
'12345'H   @> 2 da  '45123'H
'1122334455'O  @> (1+2)  da  '3344551122'O
"abcdefg"  @> 3 da  "efgabcd"
```

16 Funciones

En TTCN-3 las funciones se utilizan para expresar comportamientos de la prueba o para estructurar cálculos en un módulo, por ejemplo, para calcular un valor, para inicializar un conjunto de variables o para verificar alguna condición. Las funciones pueden devolver un valor. Esto es indicado por la palabra clave **return** seguida por un identificador de tipo. Si no se especifica **return**, la función está vacía. En TTCN-3 no existe una palabra clave explícita para vacío. La palabra clave **return**, cuando se utiliza en el cuerpo de la función, hace que la función termine y devuelva un valor compatible con el tipo return. Por ejemplo:

```
// Definición de MyFunction que no tiene parámetros
function MyFunction() return integer
{

   return 7;  // devuelve el valor entero 7 cuando la función termina
}
```

NOTA – Las funciones de TTCN-3 sustituyen las operaciones de series de pruebas y las definiciones de procedimiento de series de prueba de TTCN-2. Las funciones informales pueden ser declaradas como funciones externas con comentarios explicativos o utilizando una función formal vacía con comentarios.

Una función puede ser definida dentro de un módulo o puede ser declarada como definida externamente (es decir, **external**). Para una función externa sólo hay que proporcionar la interfaz de función en el módulo TTCN-3. La realización de la función externa está fuera del ámbito de la presente Recomendación. Las funciones externas no están autorizadas a contener operaciones de puertos.

```
external function MyFunction4() return integer; // Función externa sin
                                                // parámetros que devuelve un
                                                // valor entero

external function InitTestDevices(); // Una función externa que sólo tiene un
                                      // efecto fuera del módulo TTCN-3
```

En un módulo, el comportamiento de una función puede ser definido utilizando los enunciados de programa y operaciones definidos en la cláusula 18. Si una función incluye operaciones de puertos, se hará referencia al tipo de componente asociado utilizando las palabras clave **runs on** en el encabezamiento de la función para definir el número, tipo e identificadores de los puertos disponibles. La única excepción de esta regla es si todos los puertos utilizados dentro de la función son introducidos como parámetros.

Si una función incluye operaciones de puertos, todos los puertos utilizados dentro de la función serán introducidos como parámetros o se hará referencia a un tipo de componente asociado utilizando **runs on** en el encabezamiento de la función para definir el número, tipo e identificadores de los puertos disponibles. Por ejemplo:

```
function MyFunction() runs on MyComponent return integer
{
   :
}
```

Diferentes ejemplares de componente pueden utilizar la misma función si cumplen la siguiente regla de coherencia:

"Sean C1 y C2 los dos tipos de componentes y FUNC una función que hace referencia a C1 en su cláusula ***runs on****. Un ejemplar de componente C2 puede utilizar FUNC si la función de tipo C2 incluye la definición de tipo completa de C1. Esto significa que C2 incluye los mismos nombres para direccionar puertos del mismo tipo que C1."*

16.1 Parametrización de funciones

Las funciones pueden ser parametrizadas. Las reglas para las listas de parámetros formales se ajustarán a lo definido en 5.3. Por ejemplo:

```
function MyFunction2(inout integer MyPar1)
}
                          // MyFunction2 no devuelve un valor
   MyPar1 := 10 * MyPar1; // pero cambia el valor de MyPar1 que
                          // se introduce por referencia
}

function MyFunction3() runs on MyPTCType
   {
                             // MyFunction3 no devuelve un valor, pero
var integer MyVar := 5;      // usa la operación de puerto
   PCO1.send(MyVar);         // send y por tanto requiere una cláusula runs on
                             // para resolver los identificadores de puerto
   }                         // referenciando un tipo de componente
)
```

16.2 Invocación de funciones

Una función es invocada haciendo referencia a su nombre y por la lista real de parámetros. Las funciones que no devuelven valores pueden ser invocadas directamente. Las funciones que devuelven valores pueden ser invocadas dentro de expresiones. Se aplicarán las reglas para las listas de parámetros reales según se define en 5.3.

```
MyVar := MyFunction4();   // El valor devuelto por  MyFunction4 se asigna a
                          // MyVar. Los tipos del valor devuelto y MyVar tienen
                          // que ser iguales

MyFunction2(MyVar2);      // MyFunction2 no devuelve un valor y es llamada con
                          // el parámetro real MyVar2, que puede ser introducido
                          // por referencia
```

```
MyVar3 := MyFunction6(4)+ MyFunction7(MyVar3);   // Funciones usadas en
                                                 // expresiones
```

Se aplican restricciones especiales a funciones limitadas a componentes de prueba que utilizan la operación **start**. Estas restricciones se describen en 21.5.

16.3 Funciones predefinidas

TTCN-3 contiene varias funciones predefinidas (incorporadas) (véase el cuadro 9) que no tienen que ser declaradas antes de su utilización.

Cuadro 9/Z.140 – Lista de funciones predefinidas de TTCN-3

Categoría	Función	Clave
Funciones de conversión	Convertir valor **integer** a valor **char**	**int2char**
	Convertir valor **char** a valor **int**	**char2int**
	Convertir valor **integer** a valor **universal char**	**int2unichar**
	Convertir valor **universal char** a valor **int**	**unichar2int**
	Convertir valor **bitstring** a valor **integer**	**bit2int**
	Convertir valor **hexstring** a valor **integer**	**hex2int**
	Convertir valor **octetstring** a valor **integer**	**hex2int**
	Convertir valor **charstring** a valor **integer**	**str2int**
	Convertir valor **integer** a valor **bitstring**	**int2bit**
	Convertir valor **integer** a valor **hexstring**	**int2hex**
	Convertir valor **integer** a valor **octetstring**	**int2oct**
	Convertir valor **integer** a valor **charstring**	**int2str**
Funciones de longitud/tamaño	Devolver la longitud de un valor de cualquier tipo de cadena	**lengthof**
	Devolver el número de elementos en **record**, **record of**, **template**, **set**, **set of** o **array**	**sizeof**
Funciones de presencia/ elección	Determinar si está presente un campo facultativo en **record**, **record of**, **template**, **set** o **set of**	**ispresent**
	Determinar la elección hecha en un tipo **union**	**ischosen**

Cuando se invoca una función predefinida:

1) el número de parámetros reales será igual que el número de parámetros formales; y
2) cada parámetro real evaluará a un elemento de su tipo de parámetro formal correspondiente; y
3) todas las variables que aparecen en la lista de parámetros estarán vinculadas.

En el anexo D figura una descripción completa de las funciones predefinidas.

17 Casos de prueba

Los casos de prueba son una clase de función especial. Su ejecución en la parte de control del módulo se relaciona con el enunciado **execute** (véase 26.1). El resultado de un caso de prueba ejecutado es siempre un valor de tipo **verdicttype**. Cada caso de prueba contendrá solamente un MTC a cuyo tipo se hace referencia en el encabezamiento de la definición de caso de prueba. El comportamiento definido en el cuerpo del caso de prueba es el comportamiento del MTC.

Cuando se invoca un caso de prueba, se ejemplifican los puertos de la interfaz del sistema de prueba, se crea el MTC y se comienza el comportamiento especificado en la definición del caso de prueba en el MTC. Todas estas acciones serán ejecutadas implícitamente, es decir, sin las operaciones **create** y **start** explícitas.

Con el fin de proporcionar la información para que estas operaciones implícitas puedan ocurrir, una definición de caso de prueba tiene dos partes:

a) la parte de interfaz (obligatoria): indicada por la palabra clave **runs on** que hace referencia al tipo de componente requerido para el MTC y hace que los nombres de puertos asociados sean visibles dentro del comportamiento del MTC; y

b) la parte de sistema de prueba (facultativa): indicada por la palabra **system** que hace referencia al tipo de componente que define los puertos requeridos para la interfaz del sistema de prueba. La parte de sistema de prueba se omitirá únicamente si, durante la ejecución de la prueba, sólo se ha ejemplificado el MTC. En este caso, el tipo MTC define implícitamente los puertos de la interfaz del sistema de prueba.

Ejemplo:

```
testcase MyTestCaseOne()
runs on MyMtcType1        // Define el tipo del MTC
system  MyTestSystemType  // Hace visibles los nombres de puertos de TSI al MTC
{
   :  // El comportamiento definido aquí se ejecuta en el MTC cuando se invoca
      // el caso de prueba
}

// o, un caso de prueba donde sólo se ejemplifica el MTC
testcase MyTestCaseTwo() runs on MyMtcType2
{
   :  // El comportamiento definido aquí se ejecuta en el MTC cuando se invoca
      // el caso de prueba
}
```

18 Enunciados de programa y operaciones

Los elementos de programa fundamentales de la parte de control de los módulos y funciones TTCN-3 son enunciados de programa básicos, tales como expresiones, asignaciones, construcciones de bucle, etc., enunciados comportamentales, tales como comportamiento secuencial, comportamiento alternativo, entrelazado, comportamiento por defecto, y operaciones, tales como **send**, **receive**, **create**, etc.

Los enunciados pueden ser enunciados simples (que no incluyen otros enunciados de programa) o enunciados compuestos (que pueden incluir otros enunciados).

Los bloques de enunciados son un mecanismo para agrupar enunciados y se pueden utilizar en diferentes unidades de ámbito, es decir, control de módulo, funciones y comportamientos de prueba. La clase de enunciado que puede ser utilizada en un bloque dependerá de la unidad de ámbito en la cual se utiliza el bloque. Por ejemplo, un bloque de enunciados que aparece en una función sólo utilizará los enunciados de programa que pueden ser empleados en funciones.

En 5.4 se describen las reglas de alcance generales.

Un bloque de enunciados equivale sintácticamente a un solo enunciado, de modo que siempre que se permite un enunciado en una función, puede aparecer un bloque. Esto entraña que los bloques pueden estar jerarquizados. Las declaraciones, si hubiere alguna, se harán al principio del bloque. Estas declaraciones sólo son visibles dentro del bloque y a los subbloques jerarquizados.

Las declaraciones en el bloque se ejecutarán en el orden de su aparición. Se permite la especificación de un bloque de enunciados vacío, es decir { }. Un bloque de enunciados vacío supone que no se ha ejecutado ninguna acción.

Cuadro 10/Z.140 – Visión general de los enunciados y operaciones de TTCN-3

Enunciado	Palabra clave o símbolo asociados	Se puede utilizar en control de módulo	Se puede utilizar en funciones, casos de prueba y alternativas denominadas
Enunciados de programas básicos			
Expressions	`(...)`	Sí	Sí
Assignments	`:=`	Sí	Sí
Logging	`log`	Sí	Sí
Label and Goto	`label` / `goto`	Sí	Sí
If-else	`if (...) { ... }` `else { ... }`	Sí	Sí
For loop	`for (...) { ... }`	Sí	Sí
While loop	`while (...) { ... }`	Sí	Sí
Do while loop	`do { ... }` `while (...)`	Sí	Sí
Stop execution	`stop`	Sí	Sí
Enunciados de programa comportamentales			
Comportamiento alternativo	`alt { ... }`	Sí (nota 1)	Sí
Alternativa denominada	`named alt { ... }`	Sí (nota 1)	Sí
Comportamiento entrelazado	`interleave { ... }`	Sí (nota 1)	Sí
Activar un comportamiento por defecto	`activate`	Sí (nota 1)	Sí
Desactivar un comportamiento por defecto	`deactivate`	Sí (nota 1)	Sí
Devolver control	`return`		Sí
Operaciones de configuración			
Crear componente de prueba paralelo	`create`		Sí
Conectar componente a componente	`connect`		Sí
Desconectar dos componentes	`disconnect`		Sí
Corresponder puerto con interfaz de prueba	`map`		Sí
Descorresponder puerto de la interfaz del sistema de prueba	`unmap`		Sí
Obtener dirección de MTC	`mtc`		Sí
Obtener dirección de interfaz del sistema de prueba	`system`		Sí
Obtener dirección propia	`self`		Sí
Comenzar ejecución de	`start`		Sí

Cuadro 10/Z.140 – Visión general de los enunciados y operaciones de TTCN-3

Enunciado	Palabra clave o símbolo asociados	Se puede utilizar en control de módulo	Se puede utilizar en funciones, casos de prueba y alternativas denominadas
componente de prueba			
Operaciones de configuración			
Detener ejecución de componente de prueba	`stop`		Sí
Comprobar terminación de un PTC	`running`		Sí
Esperar terminación de un PTC	`done`		Sí
Operaciones de comunicación			
Enviar mensaje	`send`		Sí
Invocar llamada de procedimiento	`call`		Sí
Responder a llamada de procedimiento desde entidad distante	`reply`		Sí
Plantear excepción (a una llamada aceptada)	`raise`		Sí
Recibir mensaje	`receive`		Sí
Activar mensaje	`trigger`		Sí
Aceptar llamada de procedimiento desde entidad distante	`getcall`		Sí
Tratar respuesta de una llamada previa	`getreply`		Sí
Capturar excepción (de entidad llamada)	`catch`		Sí
Comprobar mensaje/llamada (vigente) recibidos	`check`		Sí
Liberar puerto	`clear`		Sí
Liberar y dar acceso a puerto	`start`		Sí
Detener acceso (receptor & emisor) en puerto	`stop`		Sí
Operaciones de temporizador			
Arrancar temporizador	`start`	Sí	Sí
Detener temporizador	`stop`	Sí	Sí
Leer tiempo transcurrido	`read`	Sí	Sí
Comprobar si el temporizador funciona	`running`	Sí	Sí
Evento de expiración de temporizador	`timeout`	Sí	Sí
Operaciones de veredicto			
Fijar veredicto local	`verdict.set`		Sí
Obtener veredicto local	`verdict.get`		Sí

Cuadro 10/Z.140 – Visión general de los enunciados y operaciones de TTCN-3

Enunciado	Palabra clave o símbolo asociados	Se puede utilizar en control de módulo	Se puede utilizar en funciones, casos de prueba y alternativas denominadas
Operaciones de SUT			
Acción a distancia que ha de ser efectuada por el SUT	`sut.action`		Sí
Ejecución de casos de prueba			
Ejecutar el caso de prueba	`execute`	Sí	Sí (nota 2)
NOTA 1 – Se puede utilizar en control con operaciones de temporizador solamente. NOTA 2 – Se puede utilizar en funciones y alternativas denominadas que se usan en control de módulo.			

19 Enunciados de programa básicos

Los enunciados de programa básicos son expresiones, asignaciones, operaciones, construcciones de bucle, etc. Todos los enunciados de programa básico pueden ser utilizados en la parte de control de un módulo y en funciones TTCN-3.

Cuadro 11/Z.140 – Visión general de enunciados de programa básicos de TTCN-3

Enunciados de programa básico	
Enunciados	**Palabra clave o símbolo asociado**
Expressions	`(...)`
Assignments	`:=`
Logging	`log`
Label and Goto	`label / goto`
If-else	`if (...) { ... } else { ... }`
For loop	`for (...) { ... }`
While loop	`while (...) { ... }`
Do while loop	`do { ... } while (...)`
Stop execution	`stop`

19.1 Expresiones

TTCN-3 permite especificar expresiones utilizando los operadores definidos en la cláusula 15. Las expresiones se construyen a partir de otras expresiones (simples). Las expresiones pueden contener funciones. El resultado de una expresión será el valor de un tipo específico y los operadores utilizados serán compatibles con el tipo de los operandos. Por ejemplo:

```
(x + y - increment(z))*3;
```

19.1.1 Expresiones booleanas

Una expresión **`boolean`** sólo contendrá valores **`boolean`** y/u operadores **`boolean`** y/u operadores relacionales y dará un valor **`boolean`** de **`true`** o **`false`**. Por ejemplo:

```
((A and B) or (not C) or (j<10));
```

19.2 Asignaciones

Se puede asignar valores a variables. Esto se indica mediante el símbolo ":=". Durante la ejecución de una asignación, el lado derecho de la asignación evaluará un elemento del mismo tipo del lado izquierdo. El efecto de una asignación es vincular la variable (que puede ser también un elemento de **record** o **set** etc.) con el valor de expresión. La expresión no contendrá variables no vinculadas. Todas las asignaciones se producen en el orden en que aparecen, es decir, procesamiento de izquierda a derecha. Por ejemplo:

```
MyVariable := (x + y – increment(z))*3;
```

19.3 El enunciado Log

El enunciado **log** proporciona los medios para escribir una cadena de caracteres en algún dispositivo de registro cronológico asociado con control de prueba o el componente de prueba en el cual se usa el enunciado. Por ejemplo:

```
log("Line 248 in PTC_A");
// La cadena "Line 248 en PTC_A" se escribe a algún dispositivo de registro del
// sistema de prueba
```

NOTA – La definición de capacidades complejas de registro y rastreo, que pueden depender de las herramientas, está fuera del ámbito de la presente Recomendación.

19.4 El enunciado Label

El enunciado **label** permite especificar etiquetas en casos de prueba, funciones, alternativas denominadas y la parte de control de un módulo. Un enunciado **label** puede ser utilizado libremente como otros enunciados de programa comportamentales de TTCN-3 de acuerdo con las reglas de sintaxis definidas en el anexo A. Puede ser utilizado antes o después de un enunciado TTCN-3 pero, por ejemplo, no como primer enunciado de una alternativa en un enunciado **alt** o **interleave** (véase 20.2.7).

19.5 El enunciado Goto

El enunciado **goto** pude ser utilizado en funciones, casos de prueba, alternativas denominadas y en la parte de control de un módulo TTCN. El enunciado **goto** ejecuta un salto a **label** o al comienzo de un enunciado **alt** para forzar comportamiento repetido (véase 20.2.8).

19.6 El enunciado If-else

El enunciado **if-else**, también conocido como el enunciado condicional, se utiliza para indicar ramificación en el control de flujo debido a expresiones **boolean**. Esquemáticamente el condicional aparece como sigue:

```
if (expression1)
   statementblock1
else
   statementblock2
donde statementblockx, hace referencia a un bloque de enunciados.
```

Ejemplo:

```
if (date == "1.1.2000") return { fail };

if (MyVar < 10) {
      MyVar := MyVar * 10;
      log ("MyVar < 10");
}
```

```
else {
   MyVar := MyVar/5;
}
```

Un esquema más complejo podría ser:

```
if (expression1)
   statementblock1
else if (expression2)
   statementblock2
:
else if (expressionn)
   statementblockn
else
   statementblockn+1
```

En estos casos, la legibilidad depende considerablemente del formato, pero éste no tendrá significado sintáctico ni semántico.

19.7 El enunciado For

El enunciado **for** define un bucle de contador. El valor de la variable índice es aumentado, disminuido o manipulado, de manera que después de un determinado número de bucles de ejecución se alcanzan un criterio de terminación.

El enunciado **for** contiene dos asignaciones y una expresión **boolean**. La primera asignación es necesaria para inicializar la variable índice (o contador) del bucle. La expresión **boolean** termina el bucle y la segunda asignación se usa para manipular la variable índice. Por ejemplo:

```
for (j:=1; j<=10; j:= j+1) { ... }
```

El criterio de terminación del bucle será expresado por la expresión **boolean**. Se comprueba al principio de cada nueva iteración de bucle. Si evalúa **true**, la ejecución continúa con el enunciado que sigue inmediatamente al bucle **for**.

La variable de índice de un bucle **for** puede ser declarada antes de ser utilizada en el enunciado **for** o puede ser declarada e inicializada en el encabezamiento del enunciado **for**. Si la variable índice es declarada e inicializada en el encabezamiento del enunciado **for**, el alcance de la variable índice está limitada al cuerpo del bucle, es decir, sólo es visible dentro del cuerpo del bucle. Por ejemplo:

```
var integer j;                          // Declaración de variable de entero j
for (j:=1; j<=10; j:= j+1) { ... }      // Uso de la variable j como variable
                                        // de índice del bucle for

for (var float i:=1.0; i<7.9; i:= i*1.35) { ... }  // La variable de índice i es
                                        // declarada e inicializada en el
                                        // encabezamiento del bucle for. La
                                        // variable i sólo es visible en el cuerpo
                                        // del bucle.
```

19.8 El enunciado While

El bucle **while** se ejecuta mientras se mantiene la condición de bucle. La condición de bucle será verificada al principio de cada nueva iteración de bucle. Si la condición de bucle no se mantiene, se sale del bucle y la ejecución continuará con el enunciado, que sigue inmediatamente al bucle **while**. Por ejemplo:

```
while (j<10){ ... }
```

19.9 El enunciado Do-while

El bucle **do-while** es idéntico a un bucle **while**, con la excepción de que la condición de bucle será verificada al *final* de cada iteración de bucle. Esto significa cuando se utiliza un bucle **do-while** que el comportamiento se ejecuta por lo menos una vez antes de evaluar la condición de bucle por la primera vez. Por ejemplo:

```
do { ... } while (j<10);
```

19.10 El enunciado Stop execution

El enunciado **stop execution** termina la ejecución de diferentes, maneras según el contexto en el cual se utiliza. Cuando se emplea en la parte de control de un módulo, termina la ejecución de todo el módulo. Cuando se utiliza en una función que está ejecutando comportamiento, termina el componente de prueba pertinente.

20 Enunciados de programa comportamentales

Los enunciados de programas comportamentales pueden ser utilizados en casos de prueba, funciones y control de módulo, salvo para el enunciado **return**, que sólo se empleará en casos de pruebas y funciones. Especifican el comportamiento dinámico de los componentes de prueba en los puertos de comunicación. El comportamiento de prueba puede ser expresado, secuencialmente, como un conjunto de alternativas o combinaciones de ambos. Un operador de entrelazado permite especificar secuencias entrelazadas o alternativas.

Cuadro 12/Z.140 – Visión general de enunciados de programa comportamentales de TTCN-3

Enunciados de programa comportamentales	
Enunciado	**Palabra clave o símbolo asociado**
Comportamiento alternativo	**alt { ... }**
Alternativa denominada	**named alt { ... }**
Comportamiento entrelazado	**interleave { ... }**
Activar un comportamiento por defecto	**activate**
Desactivar un comportamiento por defecto	**deactivate**
Devolver control	**return**

20.1 Comportamiento secuencial

La forma más sencilla de comportamiento es un conjunto de enunciados que son ejecutados secuencialmente, como se ilustra en la figura 5:

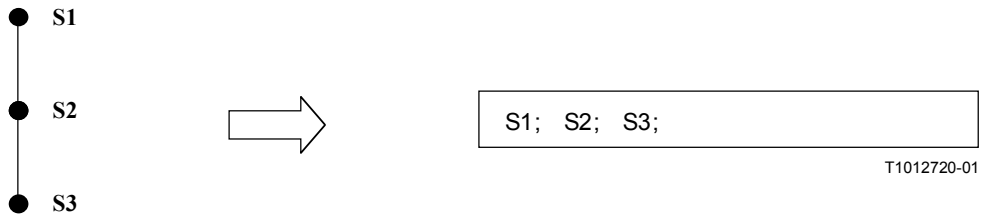


Figura 5/Z.140 – Ilustración de comportamiento secuencial

Los enunciados individuales en la secuencia serán separados por el delimitador ";". Por ejemplo:

```
MyPort.send(Mymessage); MyTimer.start; log("Done!");
```

20.2 Comportamiento alternativo

Una forma más compleja de comportamiento es cuando las secuencias de enunciados se expresan como conjuntos de posibles alternativas para formar un árbol de trayectos de ejecución, como se ilustra en la figura 6:

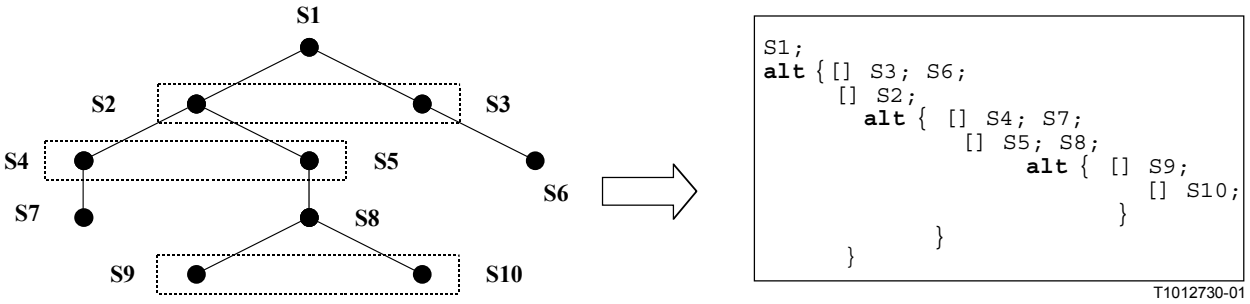


Figura 6/Z.140 – Ilustración de comportamiento alternativo

El enunciado **alt** indica ramificación de comportamiento de prueba debida a la recepción y tratamiento de eventos de comunicación y/o de temporizador y/o a la terminación de componentes de prueba paralelos, es decir, se relaciona con la utilización de las operaciones TTCN-3, **receive**, **trigger**, **getcall**, **getreply**, **catch**, **check**, **timeout** y **done**. El enunciado **alt** indica un conjunto de posibles eventos que han de ser concordados con una instantánea particular (véase 20.2.1).

NOTA – El enunciado **alt** corresponde con las alternativas en el mismo nivel de sangrado que en TTCN-2. Sin embargo, hay tres diferencias importantes:

a) las expresiones **boolean** para inhabilitar alternativas sólo pueden ser hechas en un enunciado alternativo;

b) no es posible examinar la cola de puerto utilizando la expresión **boolean** y después inhabilitar una alternativa;

c) no es posible invocar una función como una alternativa en el enunciado **alt**, salvo en el caso cuando **else** (es decir [**else**]) es la última elección en la alternativa (véase 20.2.3).

Ejemplo:

```
// Uso de enunciados alternativos jerarquizados
:
alt
{
[] L1.receive(DL_REL_CO:*)                  // UA o DM recibido; capa 2 liberada
   {  verdict.set(pass);
      TAC.stop;
      TNOAC.start;
      alt {
      [] L1.receive(DL_EST_IN)              // SABME recibido
         {  TNOAC.stop;
            verdict.set(pass);
         }
      [] TNOAC.timeout
         {  L1.send(DEL_EST_RQ:*);
            TAC.start;
```

```
            alt {
            [] L1.receive(DL_EST_CO:*)   // UA recibido; enlace de datos
                                         // establecido
               {  TAC.stop;
                  verdict.set(pass)
               }
            [] TAC.timeout               // Ninguna respuesta
               {verdict.set(inconc)}
            [] L1.receive                // Como OTHERWISE en TTCN-2
               {verdict.set(inconc)}
            }
         }
      [] L1.receive                      // Como OTHERWISE en TTCN-2
         {verdict.set(inconc)}
      }
   }
[] TAC.timeout                           // Ninguna respuesta
   {verdict.set(inconc)}
[] L1.receive                            // Como OTHERWISE en TTCN-2
   {verdict.set(inconc)}
}
:

// Uso de alternativa con expresiones (o guarda) Booleanas
:
alt {
[]     L1.receive(MyMessage1)
       {verdict.set(fail)}
[x>1]  L2.receive(MyMessage2)            // Guarda/expresión Booleana
       {verdict.set(pass)}
[x<=1] L2.receive(MyMessage3)            // Guarda/expresión Booleana
       {verdict.set(inconc)}
}
:

// Uso de done en alternativas
:
alt {
   [] MyPTC.done {
         verdict.set(pass)
      }
   [] any port.receive {
         goto alt
      }
}
:
```

20.2.1 Ejecución de comportamiento alternativo

Los enunciados alternativos en un enunciado **alt** son procesados en su orden de aparición. La semántica operacional de TTCN-3 (véase el anexo B) supone que el estado de cualquiera de los eventos no puede cambiar durante el proceso de tratar de concordar una alternativa en un conjunto de alternativas. Esto supone que se utiliza semántica de instantánea para eventos y temporizaciones recibidas, es decir, cada vez alrededor de un conjunto de alternativas se toma una instantánea de los eventos que han sido recibidos y las temporizaciones que han expirado. Sólo los identificados en la instantánea pueden ser concordados en el siguiente ciclo a través de las alternativas.

NOTA 1 – Estas semánticas son exactamente iguales que para TTCN-2.

NOTA 2 – Los eventos síncronos (por ejemplo, **call**) bloquean el bucle hasta que se completa una llamada.

20.2.2 Selección/deselección de una alternativa

Si es necesario, es posible habilitar/inhabilitar una alternativa por medio de una expresión **boolean** colocada entre los corchetes '[ ]' de la alternativa. Por ejemplo:

```
[MyVar==3] PCO.receive(MyMessage) {}
```

Los corchetes cuadrados de apertura y de cierre '[' ']' estarán presentes al comienzo de cada alternativa, incluso si están vacíos. Esto no sólo ayuda a la legibilidad sino que también es necesario para distinguir sintácticamente una alternativa de otra.

20.2.3 Rama Else en alternativas

Si es necesario, es posible definir una rama en el enunciado alternativo que se toma siempre si no se puede tomar otra alternativa definida previamente. Si se define una rama **else**, todas las alternativas definidas subsiguientemente son redundantes, es decir, nunca pueden ser alcanzadas. Por ejemplo:

```
:

alt {
[]       L1.receive(MyMessage1)
         {  verdict.set(fail);
            MyComponent.stop
         }
[x>1]    L2.receive(MyMessage2)    // Guarda/expresión Booleana
         {  verdict.set(pass);
            :
         }
[x<=1]   L2.receive(MyMessage3)    // Guarda/expresión Booleana
         {  verdict.set(inconc);
            :
         }
[else]   {  MyErrorHandling();     // Rama else
         verdict.set(fail);
         MyComponent.stop;
         }
   }
:
```

Se ha de señalar que se anexan siempre comportamientos por defecto al final de todas las alternativas. Si se define una rama **else**, nunca se introducirá un **default** activado.

NOTA – Es posible también utilizar **else** en alternativas denominadas.

20.2.4 Declaración de alternativas denominadas

Las alternativas que se utilizan en varios lugares pueden ser definidas en una alternativa denominada indicada por el par de palabras clave **named alt**. Las alternativas denominadas serán definidas globalmente en las definiciones del módulo. Cuando se invoca, una **named alt** es idéntica a la construcción **alt** de comportamiento, salvo que tiene un identificador y permite parametrización.

Una **named alt** cuando es referenciada tiene el mismo efecto que una substitución de macro. Se puede hacer referencia a una **named alt** en cualquier lugar en una definición de comportamiento donde es válido incluir una construcción **alt** normal.

Ejemplo:

```
// Definición del macro de alternativas denominadas
named alt HandlePCO2()
{
   [] PCO2.receive(DL_EST_IN)
      {PCO2.send(DL_EST_CO)}
```

```
   [] PCO2.receive(DL_EST_CO) {}
      // no hacer nada
}

// Uso de una alternativa denominada en línea
testcase TC001() runs on MyPTCtype
{
   :
     HandlePCO2();        // Llamar alternativa denominada
   :
}

// Que se amplía a
testcase TC001() runs on MyPTCtype
{
   :
   alt   {
         [] PCO2.receive(DL_EST_IN)
            {PCO2.send(DL_EST_CO)}
         [] PCO2.receive(DL_EST_CO) {}
            // no hacer nada
   }
   :
}
```

20.2.5 Expansión de alternativas con alternativas denominadas

Además de la referencia en línea directa, es posible también ampliar explícitamente las alternativas especificadas en la **named alt** utilizando el enunciado **expand**. Este enunciado puede ser colocado en cualquier posición dentro de un enunciado **alt** e insertará las guardas asociadas de **named alt** en esa posición.

Ejemplo:

```
// Uso de una named alt ampliado
testcase TC002()runs on MyPTCtype
{
  :

   alt   {
      [] PCO1.receive(DL_EST_IN)
         {PCO1.send(DL_EST_CO)}
      [] PCO1.receive(DL_EST_CO) {}
         // no hacer nada
      [expand] HandlePCO2()  // Ampliar alternativas named alt a enunciado alt
                             // especificado
      }
}

// Que se amplía a
testcase TC002()runs on MyPTCtype
{
   :
   alt   {
         [] PCO1.receive(DL_EST_IN)
            {PCO1.send(DL_EST_CO)}
         [] PCO1.receive(DL_EST_CO) {}
            // no hacer nada
         [] PCO2.receive(DL_EST_IN)
            {PCO2.send(DL_EST_CO)}
         [] PCO2.receive(DL_EST_CO) {}
            // no hacer nada
   }
}
```

20.2.6 Parametrización de alternativas denominadas

Las alternativas denominadas pueden ser parametrizadas con tipos, valores, funciones y plantillas. Como las alternativas denominadas no son una unidad de ámbito, los parámetros formales definidos son sustituidos simplemente por los parámetros reales cuando se ejecuta la expansión de macro.

Ejemplo:

```
named alt HandleAnyPCO(MyPortT PCO)
{
   [] PCO.receive(DL_EST_IN)
      {PCO.send(DL_EST_CO)}
   [] PCO.receive(DL_EST_CO) {}
      // no hacer nada
}

testcase TC001() runs on MyPTCtype
{
   HandleAnyPCO(PCO2);
   :
   alt   {
         [expand] HandleAnyPCO(PCO1);
         [expand] HandleAnyPCO(PCO2);
   }
}
```

20.2.7 El enunciado Label en comportamiento

El enunciado **label** permite especificar etiquetas en casos de prueba, funciones, alternativas denominadas y la parte de control de un módulo. Se puede utilizar antes o después de cualquier enunciado de TTCN-3, pero no será el primer enunciado de una alternativa en un enunciado **alt** o **interleave**.

Ejemplo:

```
label MyLabel;
// Define la etiqueta MyLabel

// Las etiquetas L1, L2 y L3 se definen en el siguiente fragmento de código
// TTCN-3
label L1;                        // Definición de etiqueta L1
alt{
[] PCO1.receive(MySig1)
   {  label L2;                  // Definición de etiqueta L2
      PCO1.send(MySig2);
      PCO1.receive(MySig3)
   }
[] PCO2.receive(MySig4)
   {  PCO2.send(MySig5);
      PCO2.send(MySig6);
      label L3;                  // Definición de etiqueta L3
      PCO2.receive(MySig7);
      goto L1;                   // Salto a etiqueta L1
   }
}
:
```

20.2.8 El enunciado Goto en comportamiento

El enunciado **goto** puede ser utilizado en funciones, casos de prueba, alternativas denominadas y en la parte control de un módulo TTCN. Ejecuta un salto a un enunciado **label** o al comienzo de un enunciado **alt** para forzar comportamiento repetido.

La reevaluación de un enunciado **alt** se puede lograr:

a) utilizando **goto** <*LabelId*> donde el enunciado **label** pertinente debe ser colocado inmediatamente antes de la palabra clave **alt** de la alternativa real a la que se ha de saltar; o

b) utilizando **goto alt** dentro del enunciado **alt** que debe ser reevaluado. En este caso, la palabra clave **alt** puede ser considerada como una etiqueta implícita para el enunciado **alt** dentro del cual se utiliza **goto**.

20.2.8.1 Restricción del uso de Goto

El enunciado **goto** proporciona la posibilidad de saltar libremente, es decir, hacia adelante y hacia atrás, dentro de una secuencia de enunciados, saltar fuera de un enunciado compuesto (por ejemplo, un bucle **while**) y saltar sobre varios niveles fuera de enunciados compuestos jerarquizados (por ejemplo, alternativas jerarquizadas). Sin embargo, el uso del enunciado **goto** será restringido por las siguientes reglas:

a) No se permite saltar fuera o dentro de funciones, casos de prueba, alternativas denominadas y la parte de control de un módulo TTCN.

b) No se permite saltar dentro una secuencia de enunciados definidos en un enunciado compuesto (es decir, enunciado **alt**, bucle **while**, bucle **for**, enunciado **if-else**, bucle **do-while** y el enunciado **interleave**).

c) Como una excepción a la regla a) para alternativas denominadas, se permite utilizar **goto alt** dentro de una alternativa denominada para forzar la reevaluación de un enunciado **alt** dentro del cual la alternativa denominada puede ser ampliada.

NOTA – Esta regla proporciona la posibilidad de saltar fuera de una alternativa denominada de una manera restringida con el fin de proporcionar la funcionalidad para describir supletorios.

d) No se permite utilizar el enunciado **goto** dentro de un enunciado **interleave**.

Ejemplo:

```
// El siguiente fragmento de código TTCN-3 incluye
:
label L1;
MyVar := 2 * MyVar;
if (MyVar < 2000) { goto L1; }       // ... un salto hacia atrás a L1 y
MyVar2 := Myfunction(MyVar);
if (MyVar2 > MyVar) { goto L2; }  // ... un salto hacia adelante a L2,
PCO1.send(MyVar);
PCO1.receive -> value MyVar2;
label L2;
PCO2.send(integer: 21);
alt {
   [] PCO1.receive
      { goto alt; }             // ... un salto que obliga a reevaluar el
                                // enunciado alt previo
   [] PCO2.receive(integer: 67)
      {  label L3;
         PCO2.send(MyVar);
         alt {
            [] PCO1.receive
               { goto alt; } // ... otro salto que obliga a reevaluar el
                             // enunciado alt previo (no el mismo que para el
                             // enunciado goto antes),
            [] PCO2.receive(integer: 90)
               {  PCO2.send(integer: 33);
                  PCO2.receive(integer: 13);
                  goto L4; // ... un salto hacia adelante de dos enunciados alt
                           // jerarquizados,
               }
```

```
            [] PCO2.receive(MyError)
               { goto L3; }  // ... un salto hacia atrás del enunciado alt
                             // vigente,
            [] any port.receive
               { goto L2; }  // ... un salto hacia atrás de los dos
                             // enunciados alt jerarquizados,
         }
      }
   [] any port.receive
      { goto L2; }  // ... y un largo salto hacia atrás de un enunciado alt
}
label L4;
:
```

20.3 Comportamiento entrelazado

Los enunciados de transferencia de control **for**, **while**, **do-while**, **goto**, **activate**, **deactivate**, **stop**, **return** y llamadas (directas o indirectas) de funciones definidas por el usuario, que incluyen operaciones de comunicación, no serán utilizados en enunciados **interleave**. Además, no se permite guardar ramas de un enunciado **interleave** con expresiones booleanas (es decir, '`[ ]`' estarán siempre vacíos). Tampoco se permite ampliar enunciados **interleave** con alternativas denominadas o especificar ramas **else** en comportamiento entrelazado.

El comportamiento entrelazado siempre puede ser sustituido por un conjunto equivalente de alternativas jerarquizadas. Los procedimientos para esta sustitución se describen en el anexo B.

La regla para evaluar un enunciado de entrelazado es el siguiente:

a) siempre que se ejecuta un enunciado de recepción, los siguientes enunciados de no recepción son ejecutados subsiguientemente hasta que se alcanza el siguiente enunciado de recepción o termina la secuencia entrelazada;

   NOTA – Los enunciados de recepción son enunciados TTCN-3 que pueden producirse en conjuntos de alternativa, es decir, **receive**, **check**, **trigger**, **getcall**, **getreply**, **catch** y **timeout**. Los enunciados de no recepción indican todos los demás enunciados de transferencia de no control que pueden ser utilizados dentro del enunciado entrelazador.

b) la evaluación continúa después tomando la instantánea siguiente.

En el anexo B se define completamente la semántica operacional de entrelazado.

Ejemplo:

```
// El siguiente fragmento de código TTCN-3
:
interleave {
[] PCO1.receive(MySig1)
   {  PCO1.send(MySig2);
      PCO1.receive(MySig3);
   }
[] PCO2.receive(MySig4)
   {  PCO2.send(MySig5);
      PCO2.send(MySig6);
      PCO2.receive(MySig7);
   }
}
:

// puede ser interpretado como una abreviatura para
:
alt {
[] PCO1.receive(MySig1)
   {  PCO1.send(MySig2);
```

```
      alt {
      [] PCO1.receive(MySig3)
         {  PCO2.receive(MySig4);
            PCO2.send(MySig5);
            PCO2.send(MySig6);
            PCO2.receive(MySig7)
         }
      [] PCO2.receive(MySig4)
         {  PCO2.send(MySig5);
            PCO2.send(MySig6);
            alt {
            [] PCO1.receive(MySig3) {
                  PCO2.receive(MySig7); }
            [] PCO2.receive(MySig7) {
                  PCO1.receive(MySig3); }
            }
         }
      }
   }
[] PCO2.receive(MySig4)
   {  PCO2.send(MySig5);
      PCO2.send(MySig6);
      alt {
      [] PCO1.receive(MySig1)
         {  PCO1.send(MySig2);
            alt {
            [] PCO1.receive(MySig3)
               {  PCO2.receive(MySig7);
               }
            [] PCO2.receive(MySig7)
               {  PCO1.receive(MySig3);
               }
            }
         }
      [] PCO2.receive(MySig7)
         {  PCO1.receive(MySig1);
            PCO1.send(MySig2);
            PCO1.receive(MySig3);
         }
      }
   }
}

:
```

20.4 Comportamiento por defecto

El comportamiento por defecto puede ser considerado como una extensión de un enunciado **`alt`** o una sola operación **`receive`** que se define de manera especial. Un comportamiento por defecto se definirá especificando una **`named alt`** y será activado antes de que pueda ser invocado y ejecutado.

La activación de un comportamiento por defecto significa que las alternativas definidas en la alternativa **`named alt`** pertinente son anexadas al nivel máximo de todas las alternativas subsiguientes.

El comportamiento por defecto se anexa también a cualesquiera operaciones de recepción (es decir, no en un enunciado **`alt`**), temporizaciones o enunciados **`done`**. Esto se debe a que estas operaciones son conceptualmente iguales que una sola alternativa. Por ejemplo:

```
:
MyPort.receive(MyMsg);
:
```

```
// Es igual que
:
alt {
   [] MyPort.receive(MyMsg) {}
}
:
```

20.4.1 Las operaciones Activate y Deactivate

Se activa un comportamiento por defecto utilizando la operación **activate** y se desactiva utilizando la operación **deactivate**. Una operación **deactivate** vacía desactiva todos los comportamientos por defecto activos.

En el caso de múltiple activación de múltiples alternativas denominadas, los elementos **alt** serán ampliados en el orden de activación.

Cuando el argumento de una operación de activación es una lista de alternativas denominadas, los elementos **alt** serán ampliados en el orden indicado por la lista.

Ejemplo:

```
named alt Default1() // Definición de named alt
{
   [] MyPort.check
      {MyBehaviour1()}
}

:

// dentro de la definición de comportamiento
activate( Default1() );

CL2.receive(MySetup);

alt{
   [] CL2.receive(MySig1)
      {CL2.send(MySig2)}

   [] CL2.receive(MySig2)
      {CL2.send(mySig1)}
}

// Este enunciado desactiva el comportamiento por defecto Default1
deactivate(Default1)
// Este enunciado desactiva todo comportamiento por defecto activado
deactivate;

// Conceptualmente, después de la definición y activación, alt por defecto se
// amplía al final de cualesquiera enunciados alt o receive siguientes

activate ( Default1() );
:
CL2.receive(MySetup);

alt {
      [] CL2.receive(MySig1)
         {CL2.send(MySig2)}
      [] CL2.receive(MySig2)
         {CL2.send(mySig1)}
}
```

```
// es equivalente a
:
alt {
      [] CL2.receive(MySetup);  // receive se convierte ahora en alt por derecho
                                // propio

      [] MyPort.check
         {MyBehaviour1()}
}

alt {
      [] CL2.receive(MySig1)
         {CL2.send(MySig2)}
      [] CL2.receive(MySig2)
         {CL2.send(mySig1)}

      [] MyPort.check
         {MyBehaviour1()}
}
```

20.5 El enunciado Return

El enunciado **return** termina la ejecución de una función y devuelve el control al punto desde el cual se llamó a la función. Un enunciado **return** puede ser asociado facultativamente con un valor devuelto. El uso de **return** en un caso de prueba o control es equivalente a **stop**.

Ejemplo:

```
function MyFunction() return boolean
{
   :
   if (date == "1.1.2000") { return false; }
// la ejecución se detiene el 1.1.2000 y devuelve falso como indicación de fallo
   :
   return true;  // se devuelve verdadero
}

function MyBehaviour() return verdicttype
{
   :
   if (MyFunction()) { verdict.set(pass); }   // uso de MyFunction en un
                                              // enunciado if
   else { verdict.set(inconc); }
   :
   return verdict.get; // Devolución explícita del veredicto
   }
```

21 Operaciones de configuración

Las operaciones de configuración (véase el cuadro 13) se utilizan para establecer y controlar componentes de prueba. Estas operaciones sólo serán utilizadas en casos de prueba y funciones TTCN-3 (es decir, no en la parte de control del módulo).

Cuadro 13/Z.140 – Visión general de operaciones de configuración de TTCN-3

Operaciones de configuración	
Enunciado	**Nombre de la operación**
Crear componente de prueba paralelo	**create**
Conectar un componente con otro componente	**connect**
Desconectar dos componentes	**disconnect**
Corresponder puerto de componente con puerto de interfaz de prueba	**map**
Descorresponder puerto de interfaz de sistema de prueba	**unmap**
Obtener dirección de MTC	**mtc**
Obtener dirección de interfaz de sistema de prueba	**system**
Obtener dirección propia	**self**
Comenzar ejecución de componente de prueba	**start**
Detener ejecución de componente de prueba	**stop**
Comprobar terminación de un PTC	**running**
Esperar terminación de un PTC	**done**

21.1 La operación Create

El MTC es el único componente de prueba que es creado automáticamente cuando un caso de prueba comienza. Todos los demás componentes de prueba son creados explícitamente durante la ejecución de la prueba mediante operaciones **create**. Un componente se crea con su conjunto completo de puertos cuyas colas de entrada están vacías. Además, si se define que un puerto es del tipo **in** o **inout**, estará en el estado de escucha preparado para recibir tráfico por la conexión.

Como todos los componentes y puertos son destruidos implícitamente al terminar cada caso de prueba, cada caso creará completamente su configuración requerida de componentes y conexiones cuando es invocado.

```
// Este ejemplo declara una variable de type address, que se usa para almacenar
// la referencia de un nuevo componente de tipo MyComponentType que es el
// resultado de la función create.
:
var MyComponenttype MyNewComponent;
 :
MyNewComponent := MyComponentType.create;
 :
```

La operación **create** devolverá la referencia de componente única del caso recientemente ejemplificado. La referencia única del componente será almacenada generalmente en una variable (véase 8.6) y puede ser utilizada para conectar casos y para fines de comunicación, tales como envío o recepción.

Los componentes pueden ser creados en cualquier punto en una definición de comportamiento que proporciona plena flexibilidad con respecto a configuraciones dinámicas (es decir, cualquier componente puede crear cualquier otro componente). La visibilidad de referencias de componentes se ajustará a las mismas reglas de alcance que las variables y para hacer referencia a componentes fuera de su alcance de creación, la referencia de componente será introducida como un parámetro o como un campo en un mensaje.

21.2 Las operaciones Connect y Map

Los puertos de un componente de prueba pueden ser conectados a otros componentes o a los puertos de la interfaz del sistema de prueba. En caso de conexiones entre dos componentes de prueba, se utilizará la operación **`connect`**. Cuando se conecta un componente de prueba a una interfaz de sistema de prueba, se utilizará la operación **`map`**. La operación **`connect`** conecta directamente un puerto a otro dentro del lado **`in`** conectado al lado **`out`** y viceversa. Por otro lado, la operación **`map`** puede ser considerada puramente como una traducción de nombres que define cómo deben ser referenciados los trenes de comunicación. (Véase la figura 7.)

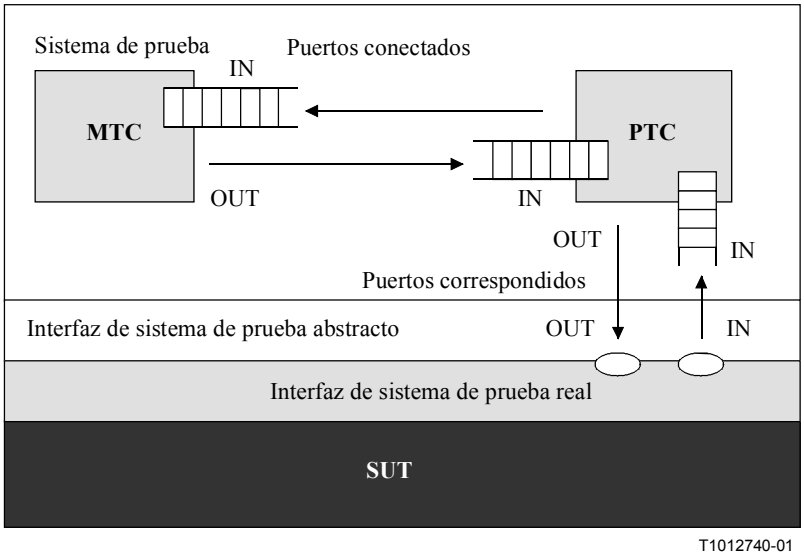


Figura 7/Z.140 – Ilustración de las operaciones Connect y Map

Con ambas operaciones **`connect`** y **`map`**, los puertos que han de ser conectados son identificados por las referencias de los componentes que han de ser conectados y los nombres de los puertos que han de ser conectados.

Hay dos operaciones para identificar el MTC, a saber, **`mtc`** y para identificar puertos de la interfaz del sistema de prueba, a saber, **`system`** (véase 8.6). Ambas operaciones pueden ser utilizadas para identificar y conectar puertos.

Las operaciones **`connect`** y **`map`** pueden ser llamadas desde cualquier definición de comportamiento (función). Sin embargo, antes de que cualquiera de estas operaciones sea llamada, los componentes que han de ser conectados habrán sido creados y sus referencias de componente serán conocidas junto con los nombres de los puertos pertinentes.

Ambas operaciones **`map`** y **`connect`** permiten la conexión de un puerto a más de un puerto. No se permite la conexión con un puerto correspondido o la correspondencia con un puerto conectado.

Ejemplo:

```
// Se supone que los puertos Port1, Port2, Port3 y PCO1 han sido definidos y
// declarados apropiadamente en las correspondientes definiciones de tipos de
// puerto y de componente
 :
var MyComponentType  MyNewComponent;
 :
MyNewComponent := MyComponentType.create;
 :
 :
connect(MyNewComponent:Port1, mtc:Port3);
```

```
map(MyNewComponent:Port2, system:PCO1);
 :
 :
// En este ejemplo se crea un nuevo componente de tipo MyComponentType y su
// referencia se almacena en la variable MyNewComponent. Después, en la
// operación connect, Port1 de este nuevo componente se conecta con Port3 del
// MTC. Mediante la operación map, Port2 del nuevo componente se conecta con el
// puerto PCO1 de la interfaz del sistema de prueba
```

21.2.1 Conexiones coherentes

Para ambas operaciones **connect** y **map**, sólo se permiten conexiones coherentes.

Suponiendo que:

a) los puertos PORT1 y PORT2 son los puertos que han de ser conectados;

b) inlist-PORT1 define los mensajes o procedimientos del sentido entrada de PORT1;

c) outlist-PORT1 define los mensajes o procedimientos del sentido salida de PORT1;

d) inlist-PORT2 define los mensajes o procedimientos del sentido entrada de PORT2; y

e) outlist-PORT2 define los mensajes y procedimientos del sentido salida de PORT2.

La operación **connect** se permite solamente si:

– outlist-PORT1 $\subseteq$ inlist-PORT2 *y* outlist-PORT2 $\subseteq$ inlist-PORT1.

La operación **map** (suponiendo que PORT2 es el puerto de la interfaz del sistema de prueba) se permite solamente si:

– outlist-PORT1 $\subseteq$ outlist-PORT2 *y* inlist-PORT2 $\subseteq$ inlist-PORT1.

En todos los demás casos, las operaciones no serán permitidas.

Como TTCN-3 permite configuraciones y direcciones dinámicas, no todas las pruebas de coherencia pueden ser hechas estáticamente en el tiempo de compilación. Todas las pruebas, que no podrán ser hechas en el tiempo de compilación, serán hechas en el tiempo de ejecución y originarán un error de caso de prueba cuando fracasan.

21.3 Las operaciones Disconnect y Unmap

Las operaciones **disconnect** y **unmap** son las operaciones opuestas de **connect** y **map**. Ejecutan la desconexión de puertos (previamente conectados) de componentes de prueba y la descorrespondencia de puertos (previamente correspondidos) de componentes de prueba y puertos en la interfaz del sistema de prueba.

Ambas operaciones **disconnect** y **unmap** pueden ser llamadas desde cualquier componente si se conocen las referencias de componentes pertinentes junto con los nombres de los puertos pertinentes. Una operación **disconnect** u **unmap** sólo tiene efecto si la conexión o correspondencia que ha de ser suprimida ha sido creada de antemano.

Ejemplo:

```
 :
 :
connect(MyNewComponent:Port1, mtc:Port3);
map(MyNewComponent:Port2, system:PCO1);
 :
 :
disconnect(MyNewComponent:Port1, mtc:Port3); // Desconectar la conexión hecha
                                             // previamente
unmap(MyNewComponent:Port2, system:PCO1);    // Descorresponder la
                                             // correspondencia hecha
                                             // previamente
```

21.4 Las operaciones MTC, System y Self

La referencia de componentes (véase 8.6) tiene dos operaciones, **mtc** y **system**, que devuelven la referencia del componente de prueba principal y de la interfaz del sistema de prueba, respectivamente. Además, la operación **self** se puede utilizar para devolver la referencia del componente en el cual es llamada. Por ejemplo:

```
var MyComponentType MyAddress;
MyAddress := self; // Almacenar la referencia de componente vigente
```

Las únicas operaciones permitidas en referencias de componentes son asignación y equivalente.

21.5 La operación Start test component

Cuando un componente ha sido creado y conectado, el comportamiento tiene que ser vinculado al componente y hay que comenzar la ejecución de su comportamiento. Esto se hace utilizando la operación **start** (la creación de componentes no arranca la ejecución del comportamiento del componente). La razón para distinguir entre **create** y **start** es permitir que se efectúen operaciones de conexión antes de la ejecución real del componente de prueba.

La operación **start** vinculará el comportamiento requerido con el componente de prueba. Este comportamiento se define mediante referencia a una función ya definida. Por ejemplo:

```
// Se supone que los puertos Port1, Port2, Port3 y PCO1 se han definido y
// declarado apropiadamente en las correspondientes definiciones de tipos de
// puerto y de componente
 :
var MyComponentType MyNewComponent;
 :
MyNewComponent := MyComponentType.create;
 :
connect(MyNewComponent:Port1, mtc:Port3);
connect(MyNewComponent:Port2, system:PCO1);
 :
 :
MyNewComponent.start(MyComponentBehaviour());
 :

// En este ejemplo, se crea primero un nuevo componente, después se conecta a su
// entorno y finalmente es arrancado por medio de la operación start. Para
// identificar el componente que se ha de ejecutar se usa su referencia
```

Se aplican las siguientes restricciones a una función invocada en una operación **start** componente de pruebas:

– Si esta función tiene parámetros, sólo serán parámetros **in**, es decir, parámetros de valor.

– Esta función tendrá una definición **runs on** que hace referencia al mismo tipo de componentes que el componente recientemente creado o introducirá toda la información necesaria de la definición del tipo de componente como parámetros.

– Los puertos y temporizadores sólo pueden ser introducidos en esta función si hacen referencia a puertos y temporizadores en la definición del tipo del componente recientemente creado, es decir, los puertos y temporizadores son locales de los componentes y no serán transferidos a otros componentes.

NOTA – La capacidad de introducir puertos como parámetros permite la especificación de funciones genéricas que no están vinculadas con un tipo de componente específico.

21.6 La operación Stop test component

El enunciado de **stop** componente de prueba detiene explícitamente la ejecución del componente de prueba en el cual se invoca. La operación no tiene argumentos. Por ejemplo:

```
if (date == "1.1.2000") { stop; } // la ejecución se detiene el 1.1.2000
```

Si el componente de prueba que es detenido es el MTC, todos los PTC restantes que están aún funcionando serán detenidos y el caso de prueba termina.

NOTA – El mecanismo concreto para detener los demás PTC que siguen funcionando está fuera del ámbito de la presente Recomendación.

Todos los recursos serán liberados cuando termina un componente de prueba, ya sea explícitamente utilizando la operación **stop** o mediante un enunciado **return** en la función que arrancó originalmente el componente de prueba, o implícitamente cuando el componente alcanza el final de su árbol de comportamiento. Cualesquiera variables que almacenan una referencia de componente detenido no referenciarán nada.

Las reglas para la terminación de casos de prueba y los cálculos del veredicto de prueba final se describen en la cláusula 24.

21.7 La operación Running

La operación **running** permite ejecutar el comportamiento en un componente de prueba para indagar si se ha terminado el funcionamiento del comportamiento en un componente de prueba diferente. Se considera que la operación **running** es una expresión **boolean** y, por ende, devuelve un valor **boolean** para indicar que si ha terminado el componente de prueba especificado (o todos los componentes de prueba). En contraste con la operación **done**, la operación **running** puede ser usada libremente en expresiones **boolean**. Por ejemplo:

```
if (PTC1.running)                          // Uso de running en un enunciado if
{
   // Hacer algo!
}


while (all component.running != true) { // Uso de running en una condición de
                                        // bucle
      MySpecialFunction()
}
```

21.8 La operación Done

La operación **done** permite ejecutar el comportamiento en un componente de prueba para indagar si se ha completado la ejecución del comportamiento en un componente de prueba diferente.

La operación **done** se utilizará de la misma manera que una operación receptora o una operación **timeout**. Esto significa que no se utilizará en una expresión **boolean**, pero se puede emplear para determinar una alternativa en un enunciado **alt** o como un enunciado autónomo en una descripción de comportamiento. En el segundo caso, se considera que la operación **done** es una abreviatura de un enunciado **alt** con una sola alternativa, es decir, tiene su semántica bloqueadora, por lo que proporciona la capacidad de espera pasiva para la terminación de componentes de prueba.

NOTA – La operación **done** de TTCN-3 y la operación DONE de TTCN-2 tienen semánticas idénticas.

Ejemplo:

```
// Uso de done en alternativas
:
alt {
   [] MyPTC.done {
         verdict.set(pass)
      }

   [] any port.receive {
         goto alt
      }
}
:

// el siguiente enunciado autónomo:
 :
all component.done;
 :

// tiene el siguiente significado:
 :
alt {
   [] all component.done {}
}
 :

// y por tanto, bloquea la ejecución hasta que todos los componentes de prueba
// paralelos han terminado
```

21.9 Utilización de matrices de componente

Las operaciones **create**, **connect**, **start** y **stop** no funcionan directamente en matrices de componente, por lo que se proporcionará un elemento específico de la matriz como el parámetro. Para los componentes, el efecto de una matriz se logra utilizando una matriz de referencias de componentes y asignando el elemento de matriz pertinente al resultado de la operación **create**.

```
// Este ejemplo muestra cómo modelar el efecto de crear, conectar y operar
// matrices de componentes usando un bucle y almacenando la referencia del
// componente creado en una matriz de referencias de componentes.
testcase MyTestCase() runs on MyMtcType system MyTestSystemInterface
{
    :
   var integer i;
   var MyPTCType1MyPtcType[11];
    :
   for (i:= 1; i<=10; i:=i+1)
   {
      MyPtcAddresses[i] := MyPtcType1.create;
      connect(self:PtcCoordination, MyPtcAddresses[i]:MtcCordination);
      MyPtcAddresses[i].start(MyPtcBehaviour());
   }
    :
}
```

21.10 Utilización de Any y All con componentes

Las palabras clave **any** y **all** pueden ser utilizadas con operaciones de configuración, como se indica en el cuadro 14.

Cuadro 14/Z.140 – Any y all con componentes

Operación	Permitido		Ejemplo
	any	**all**	
create			
start			
running	Sí, pero sólo desde el MTC	Sí, pero sólo desde el MTC	**Cualquier componente en funcionamiento** **Todos los componentes en funcionamiento**
done	Sí, pero sólo desde el MTC	Sí, pero sólo desde el MTC	**Cualquier componente completado** **Todos los componentes completados**
stop			

22 Operaciones de comunicación

TTCN-3 soporta comunicaciones basadas en mensaje (asíncronas) y basadas en procedimiento (síncronas) (véase 8.1). La comunicación asíncrona no bloquea en la operación **send**, como se ilustra en la figura, 8 donde el procesamiento en el MTC continúa inmediatamente después que se produce la operación **send**. El SUT es bloqueado en la operación **receive** hasta que recibe un mensaje enviado.

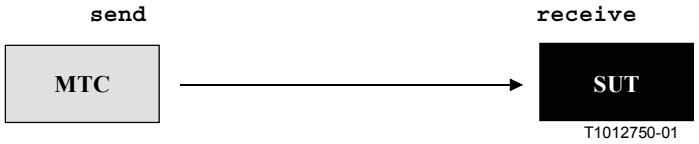


Figura 8/Z.140 – Ilustración de envío y recepción asíncronos

La comunicación síncrona bloquea en la operación **call**, como se ilustra en la figura 9, donde la operación **call** bloquea el procesamiento en el MTC hasta que se recibe **reply** o una excepción del SUT. De manera similar a **receive**, **getcall** bloquea el SUT hasta que se recibe la llamada.

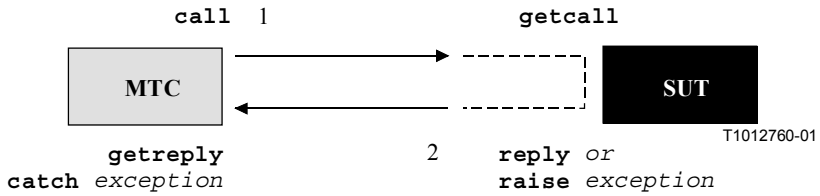


Figura 9/Z.140 – Ilustración de una llamada síncrona completa

Las operaciones tales como **send** y **call** son conocidas colectivamente como operaciones de comunicación y sólo se utilizarán en casos de prueba y funciones TTCN-3 (es decir, no directamente en la parte de control del módulo). Las operaciones de comunicación se dividen en tres grupos:

a) un componente envía un mensaje, llama a un procedimiento o responde a una llamada aceptada o plantea una excepción. Estas acciones se denominan colectivamente como *operaciones emisoras*;

b) un componente recibe un mensaje, acepta una llamada de procedimiento, recibe una respuesta para un procedimiento llamado previamente o capta una excepción. Estas acciones se denominan colectivamente *operaciones receptoras*;

c) se controla el acceso a un puerto mediante **`clear`**, **`start`** o **`stop`**. Estas acciones se denominan colectivamente como *operaciones controladoras*.

Estas operaciones pueden ser utilizadas en los puertos de comunicación de un componente de prueba como se resume en el cuadro 15. En casos de puertos mixtos, todas las operaciones son aplicables.

Cuadro 15/Z.140 – Visión general de las operaciones de comunicación de TTCN-3

Operaciones de comunicación			
Operación de comunicación	**Palabra clave**	**Se puede utilizar en puertos basados en mensaje**	**Se puede utilizar en puertos basados en procedimientos**
Operaciones emisoras			
Enviar mensaje	**`send`**	Sí	
Invocar llamada de procedimiento	**`call`**		Sí
Responder a llamada de procedimiento de entidad distante	**`reply`**		Sí
Plantear excepción (a una llamada aceptada)	**`raise`**		Sí
Operaciones receptoras			
Recibir mensaje	**`receive`**	Sí	
Activar mensaje	**`trigger`**	Sí	
Aceptar llamada de procedimiento de entidad distante	**`getcall`**		Sí
Tratar respuesta de una llamada previa	**`getreply`**		Sí
Capturar excepción (de entidad llamada)	**`catch`**		Sí
Comprobar mensaje/llamada/ excepción/respuesta recibidos	**`check`**	Sí	Sí
Controlar operaciones			
Liberar puerto	**`clear`**	Sí	Sí
Liberar y dar acceso a puerto	**`start`**	Sí	Sí
Detener acceso (receptor & emisor) a puerto	**`stop`**	Sí	Sí

22.1 Operaciones emisoras

Las operaciones emisoras son:

a) **`send`**: enviar un mensaje asíncronamente;

b) **`call`**: llamar a un procedimiento;

c) **`reply`**: responder a una llamada de procedimiento aceptada del SUT; y

d) **`raise`**: plantear una excepción cuando se recibe una llamada de procedimiento.

22.1.1 Formato general de las operaciones emisoras

Las operaciones emisoras consisten en una parte *emisión* y, en el caso de operaciones **`call`** basadas en procedimiento, en una parte *respuesta* y *tratamiento de excepciones*.

La parte emisión:

- especifica el puerto en el cual se producirá la operación especificada;
- define el valor de la información que se ha de transmitir;
- da una expresión de dirección (facultativa) que identifica de manera única al asociado de comunicación en el caso de una conexión de uno a muchos.

El nombre de puerto, el nombre de operación y el valor estarán presentes en todas las operaciones emisoras. La identificación del asociado de comunicación (indicado por la palabra clave **to**) es facultativa y sólo tiene que ser especificada en casos de conexiones de uno a muchos, donde la entidad receptora será identificada explícitamente.

22.1.1.1 Tratamiento de respuestas y excepciones

El tratamiento de respuestas y excepciones sólo se necesita en casos de comunicación síncrona. La parte de tratamiento de respuestas y excepciones de la operación **call** es facultativa y se requiere cuando el procedimiento llamado devuelve un valor o tiene parámetros **out** o **inout** cuyos valores son necesarios dentro del componente llamante y cuando el procedimiento llamado puede plantear excepciones que tienen que ser tratadas por el componente llamante.

La parte de tratamiento de respuesta y excepciones de la operación call utiliza las operaciones **getreply** y **catch** para proporcionar la funcionalidad requerida.

22.1.2 La operación Send

La operación **send** se utiliza para colocar un valor en una cola de puerto de mensajes salientes. El valor puede ser especificado por la referencia a una plantilla, una variable o una constante o puede ser definido en línea desde una expresión (que naturalmente puede ser un valor explícito). Al definir el valor en línea, se utilizará el campo de tipo facultativo si hay ambigüedad del tipo del valor que se envía.

La operación **send** sólo se utilizará en puertos basados en mensaje (o mixtos) y el tipo del valor que se ha de enviar estará en la lista de tipos salientes de la definición de tipos de puerto. Por ejemplo:

```
MyPort.send(MyTemplate(5,MyVar));
// Envía la plantilla MyTemplate con los parámetros reales 5 y MyVar por MyPort.

MyPort.send(integer:5);
// Envía el valor entero 5
```

En casos de conexiones de uno a muchos, se especificará de manera única al asociado de comunicación, que se indicará utilizando la palabra clave **to**. Por ejemplo:

```
MyPort.send("My string") to MyPartner;
// Envía la cadena "My string" a un componente con una referencia de componente
// almacenada en la variable MyPartner.

MyPCO.send(MyVariable + YourVariable – 2) to MyPartner;
// Envía el resultado de la expresión aritmética a MyPartner.
```

22.1.3 La operación Call

La operación **call** se utiliza para especificar que un componente de prueba llama a un procedimiento en el SUT o en otro componente de prueba. La operación **call** es una operación bloqueadora en cuanto a que esperará hasta que recibe una respuesta (es decir, una **reply**) o una excepción de la entidad llamada. En otras palabras, la operación **call** funciona de manera síncrona.

NOTA – Esto es comparable con la prueba de la funcionalidad de servidor, es decir, el SUT es el servidor y el componente desempeña el cometido de un cliente.

La operación **call** se utilizará solamente en puertos basados en procedimiento (o mixtos). La definición de tipos del puerto en el cual se ejecuta la operación **call** incluirá el nombre de procedimiento en su lista **out** o **inout**, es decir, se debe permitir llamar a este procedimiento en este puerto.

El valor de la operación **call** es una firma que puede ser definida en forma de una plantilla de firma o ser definida en línea. Por ejemplo:

```
signature MyProc (out integer MyPar1, inout boolean MyPar2);
:
MyPort.call(MyProc:{MyVar1,MyVar2});
// Llama al procedimiento distante MyProc at MyCL con los parámetros in e
// inout 5 y MyVar. No se prevé ningún valor devuelto ni una excepción de esta
// llamada. Si se define que uno (o ambos) parámetros es un parámetro inout,
// su valor no será considerado, es decir, no se asigna a una variable.

// El siguiente ejemplo explica las posibilidades de asignar valores a
// parámetros in e inout en el argumento de llamada. Se supone la siguiente
// firma para el procedimiento que se ha de llamar. Nótese que MyProc2 no
// tiene valor devuelto ni excepciones
signature MyProc2 (in integer A, out integer B, inout integer C);
:
MyPort.call(MyProc2:{1, - , 3});
// Sólo se especifican valores de parámetros in e inout. Los valores devueltos
// de los parámetros out e inout no se usan después de la llamada, por lo que
// no se asignan a variables.
```

Todos los parámetros **in** e **inout** tendrán un valor específico, es decir, no se permite el uso de mecanismos de concordancia tales como *AnyValue*.

Los argumentos de firma de la operación **call** no se utilizan para extraer nombres de variables para parámetros **out** e **inout**. La asignación real del valor de devolución del procedimiento y los valores de parámetros **out** e **inout** a variables se ejecutará explícitamente en la parte de tratamiento de respuestas (**getreply**) y excepciones (**catch**) de la operación **call**. Esto se indica mediante las palabras clave **value** y **param**, respectivamente y permite la utilización de plantillas de firma en operaciones **call** de la misma manera que es posible utilizar las plantillas para tipos.

En general, se supone que una operación **call** tenga semántica bloqueadora. Sin embargo, TTCN-3 soporta también llamadas no bloqueadoras. Se supone que una llamada que no devuelve valores no es una llamada bloqueadora. Las excepciones (si se especifican), planteadas por una llamada que no devuelve valores será capturada dentro de un siguiente enunciado **alt**. Además, es posible también forzar la semántica no bloqueadora mediante la palabra clave **nowait** (véase 22.1.3.3).

En casos de conexiones de uno a mucho, se especificará de manera única el asociado de comunicación. Esto será indicado mediante la palabra clave **to**. Por ejemplo:

```
MyPort.call(MyProc:{MyVar1, MyVar2}) to MyPartner;
// En este ejemplo la parte llamada es identificada explícitamente por la
// referencia de componente almacenada en la variable MyPartner.
```

22.1.3.1 Tratamiento de respuestas a llamada

El tratamiento de la respuesta a una **llamada** se hace por medio de la operación **getreply** (véase 22.2.5). Esta operación define el comportamiento alternativo que depende de la respuesta que ha sido generada como resultado de la operación **call**. Por ejemplo:

```
MyPort.call(MyProc:{MyVar1, MyVar2}) to MyPartner
                              // Where { ... } es una plantilla en línea
{
   [] MyCl.getreply(MyProc:{MyVar1, MyVar2}) {}
}
```

Si es necesario, el valor devuelto del procedimiento llamado será recogido explícitamente en la operación **getreply**. Esto se expresa utilizando '->' y la palabra clave (facultativa) **value**. Por ejemplo:

```
MyPort.call(MyProc:{MyVar1, MyVar2}) to MyPartner
{
   [] MyCl.getreply(MyProc:{MyVar1, MyVar2}) -> value MyResult {}
}
// MyProc devolverá un valor que se almacenará en la variable MyResult.
```

Los argumentos de firma de la operación **call** no se utilizan para extraer nombres de variables para los parámetros **out** e **inout**. La asignación real del valor de devolución del procedimiento y los valores de parámetro **out** e **inout** a variables se harán explícitamente en la parte de tratamiento de respuesta (**getreply**) y de excepciones (**catch**) de la operación **call**. Esto se indica mediante las palabras clave **value** y **param**, respectivamente y permite el uso de plantillas de firma en operaciones **call** de la misma manera que se pueden utilizar plantillas para tipos. Por ejemplo:

```
MyPort.call(MyProc:{5,MyVar}) to MyPartner
{
   []MyCl.getreply(MyProc:{MyVar1, MyVar2}) -> value MyResult param
      (MyPar1Var,MyPar2Var) {}
}
// En este ejemplo ambos parámetros de MyProc se especifican como inout y sus
// valores después de la terminación de MyProc se asignan a MyPar1Var y
// MyPar2Var.
```

22.1.3.2 Tratamiento de excepciones a una llamada

El tratamiento de excepciones a una **llamada** se efectúa por medio de la operación **catch** (véase 22.2.6). Esta operación define el comportamiento alternativo que depende de la excepción (si la hubiere) que ha sido generada como resultado de la operación **call**. Por ejemplo:

```
signature MyProc3 (out integer MyPar1, inout boolean MyPar2) return MyResultType
   exception (ExceptionTypeOne, ExceptionTypeTwo, ExceptionTypeThree);
:
// La siguiente operación call muestra la getreply y el mecanismo de tratamiento
// de excepciones de la operación call
MyPort.call(MyProc3:{5,MyVar}, 30E-3) to MyPartner
{
   [] MyCl.getreply(MyProc3:{MyVar1, MyVar2}) -> value MyResult param
                                                  (MyPar1Var,MyPar2Var) {}
   [] MyPort.catch(MyProc3, MyExceptionOne)
      {                              // Captura una excepción
         verdict.set(fail);          // Fija el veredicto y se
         stop                        // detiene como resultado de la
                                     // excepción
      }
   [] MyPort.catch(MyProc3, MyExceptionTwo)   // Captura una segunda excepción
      {verdict.set(inconc)}                   // Fija el veredicto y continúa
                                              // después la llamada como
                                              // resultado de la segunda
                                              // excepción

[MyCondition] MyPort.catch(MyProc3, MyExceptionThree) {}
                                 // Captura una tercera excepción que puede
                                 // ocurrir si MyCondition da verdadero

   [] MyPort.catch(timeout) {}   // Excepción de temporización, es decir, la
                                 // parte llamada no reacciona a tiempo, no se
                                 // hace nada
}
```

22.1.3.3 Tratamiento de excepciones de temporización a una llamada

La operación **call** puede incluir facultativamente una temporización. Esto se define como un valor o constante explícitos de tipo **float** y define el plazo de tiempo después que ha comenzado la operación **call** durante el cual una excepción **timeout** será generada por el sistema de prueba. Si en la operación **call** no está presente ninguna parte de valor de temporización, ninguna excepción **timeout** será generada. Por ejemplo:

```
MyPort.call(MyProc:{5,MyVar}, 20E-3)
{
   [] MyPort.catch(timeout)
      {
         verdict.set(fail);
         stop
      }
}
// Este ejemplo muestra una llamada con una temporización de 20 ms. Esto
// significa que si la parte llamada no responde con una respuesta o excepción
// dentro de este plazo, el sistema de prueba generará automáticamente una
// excepción de temporización. La temporización se trata por medio de una
// operación catch. Si el procedimiento se completa sin una excepción de
// temporización, la ejecución continúa con el enunciado que sigue a la
// operación call.
```

La utilización de la palabra clave **nowait** en la parte de valor de temporización de una operación **call** permite llamar a un procedimiento sin esperar una terminación como una respuesta, una excepción planteada por el procedimiento llamado o una excepción de temporización. Por ejemplo:

```
MyPort.call(MyProc:{5,MyVar}, nowait)
// En este ejemplo el componente de prueba continuará la ejecución sin
// esperar la terminación de MyProc.
```

En estos casos, hay que suprimir una posible respuesta o excepción de la cola utilizando una operación **getreply** o **catch** en un enunciado **alt** subsiguiente.

22.1.4 La operación Reply

La operación **reply** se utiliza para responder a una llamada aceptada previamente de acuerdo con la firma de procedimiento. Sólo se utilizará una operación **reply** en un puerto basado en procedimiento (o mixto). La definición de tipo del puerto incluirá el nombre del procedimiento al cual pertenece la operación **reply**.

La parte de valor de la operación **reply** consiste en una referencia de firma con una lista de parámetros reales asociada y valor devuelto (facultativo). La firma puede ser definida en forma de una plantilla de firma o puede ser definida en línea. Todos los parámetros **out** e **inout** de la firma tendrán un valor específico, es decir, no se permite el uso de mecanismos de concordancia tales como *AnyValue*. Por ejemplo:

```
MyPort.reply(MyProc2:{ - ,5});
// Respuestas a una llamada aceptada de MyProc2. MyProc2 no tiene valor devuelto
// sino dos parámetros.
// El primero es un parámetro in, es decir, su valor no será respondido y por
// tanto no tiene que ser especificado. El segundo es un parámetro out o inout.
// Su valor es 5.
```

En casos de conexiones de uno a muchos, el asociado de comunicación será especificado explícitamente y será único. Esto se indicará utilizando la palabra clave **to**. Por ejemplo:

```
MyPort.reply(MyProc3:{ - ,5}) to MyPartner;
// Este ejemplo es idéntico al anterior, pero la respuesta está dirigida a un
// componente cuya referencia está almacenada en MyPartner
```

Si se ha de devolver un valor a la parte llamante, éste será indicado explícitamente utilizando la palabra clave **value**.

```
MyPort.reply(MyProc:{5,MyVar} value 20);
// Repuesta a una llamada aceptada de MyProc. El valor devuelto de MyProc es 20
// y tiene dos parámetros que son out o inout. Su valor es proporcionado por 5
// y MyVar.
```

22.1.5 La operación Raise

La operación **raise** se utiliza para plantear una excepción, que sólo será planteada en un puerto basado en procedimiento (o mixto). Una excepción es una reacción a una llamada de procedimiento aceptada cuyo resultado origina un evento excepcional. El tipo de excepción será especificado en la firma del procedimiento llamado. La definición de tipo del puerto incluirá en su lista de llamadas de procedimiento aceptadas, el nombre del procedimiento al cual pertenece la excepción.

NOTA – La relación entre una llamada aceptada y la operación **raise** no puede ser verificada siempre estáticamente. Para la prueba se permite especificar una operación **raise** sin una operación **getcall** asociada.

La parte de valor de la operación **raise** consiste en la referencia de firma seguida por el valor de excepción. Por ejemplo:

```
MyPort.raise(MySignature, MyVariable + YourVariable – 2);
// Plantea una excepción con un valor resultante de la expresión aritmética
// en MyPort
```

Las excepciones se especifican como un tipo, por lo que el valor de excepción puede ser derivado de una plantilla o ser el valor resultante de una expresión (que naturalmente puede ser un valor explícito). El campo de tipo facultativo en la especificación de valor de la operación **raise** se utilizará cuando es necesario evitar toda ambigüedad del tipo del valor que se envía. Por ejemplo:

```
MyPort.raise(MyProc, MyExceptionType:{5, MyVar});
// Plantea una excepción del procedimiento distante definido por Myproc con el
// valor definido por MyExceptionTemplate con los parámetros reales 5 y MyVar en
// el puerto MyPort
```

En casos de conexiones de uno a muchos, el asociado de comunicación se especificará de manera única y se indicará utilizando la palabra clave **to**. Por ejemplo:

```
MyPort.raise(MySignature, "My string") to MyPartner;
// Plantea una excepción de cadena con el valor "My string" en MyPort a un
// componente cuya referencia está almacenada en la variable MyPartner
```

22.2 Operaciones receptoras

Las operaciones receptoras son:

a) **receive**: recibir un mensaje enviado asíncronamente;

b) **trigger**: activar la recepción de un mensaje específico;

c) **getcall**: aceptar una llamada de procedimiento;

d) **getreply**: tratar la respuesta a un procedimiento llamado previamente;

e) **catch**: capturar una excepción que tiene que ser planteada como una reacción a una operación **call**; y

f) **check**: verificar el elemento de nivel más alto en la cola de entrada de un puerto determinado.

22.2.1 Formato general de las operaciones receptoras

Una operación receptora consiste en una parte *recepción* y en una parte *asignación*.

La parte recepción:

a) especifica el puerto en el cual se ejecutará la operación;

b) define una parte concordante que especifica la entrada aceptable con la cual concordará el enunciado;

c) indica una expresión de dirección (facultativa) que identifica de manera única al asociado de comunicación (en el caso de conexiones de uno a muchos).

La parte de nombre de puerto, nombre de operación y valor de todas las operaciones receptoras estará presente. La identificación del asociado de comunicación (indicado por la palabra clave **`from`**) es facultativa y sólo tiene que ser especificada en los casos de conexiones de uno a muchos cuando la entidad receptora tiene que ser identificada explícitamente.

22.2.1.1 Asignaciones en operaciones receptoras

La parte de asignación de una operación receptora es facultativa. Para puertos basados en mensaje, se utiliza cuando se requiere almacenar mensajes recibidos. En el caso de puertos basados en procedimiento, se utiliza para almacenar los parámetros **`in`** e **`inout`** de una llamada aceptada o para almacenar excepciones.

Además, la parte de asignación puede ser utilizada también para asignar la dirección **`sender`** de un mensaje, excepción, **`reply`** o **`call`** a una variable. Esto es útil para conexiones de uno a muchos donde, por ejemplo, el mismo mensaje o llamada puede ser recibido de diferentes componentes, pero el mensaje, **`reply`** o excepción debe ser devuelto al componente emisor original.

22.2.2 La operación Receive

La operación **`receive`** se utiliza para recibir un valor de una cola de puerto de mensajes entrantes. El valor puede ser especificado haciendo referencia a una plantilla, una variable o a una constante o puede ser definido en línea desde una expresión (que naturalmente puede ser un valor explícito). Al definir el valor en línea, se utilizará el campo de tipo facultativo para evitar toda ambigüedad del tipo del valor que se recibe. La operación **`receive`** sólo se utilizará en puertos basados en mensaje (o mixtos) y el tipo del valor que se ha de recibir se incluirá en la lista de tipos entrantes de la definición del tipo de puerto.

La operación **`receive`** suprime el mensaje de nivel más alto de la cola de puerto entrante asociada solamente si ese mensaje satisface todos los criterios de concordancia asociados con la operación **`receive`**. No se producirá ninguna vinculación de los valores entrantes con los términos de la expresión o con la plantilla.

Si la concordancia no es satisfactoria, el mensaje de nivel más alto no será suprimido de la cola de puerto, es decir, si la operación **`receive`** no tiene éxito, la ejecución del caso de prueba continuará con la alternativa siguiente.

Los criterios de concordancia se relacionan con el tipo y valor del mensaje que se ha de recibir. El tipo y valor del mensaje que se ha de recibir puede ser derivado de una plantilla o ser el valor resultante de una expresión (que naturalmente puede ser un valor explícito).

```
MyPort.receive(MyTemplate(5, MyVar));
// Especifica la recepción de un valor que cumple las condiciones definidas por
// la plantilla MyTemplate con parámetros reales 5 y MyVar.

MyPort.receive(A<B);
// Especifica la recepción de un valor Booleano verdadero o falso dependiendo
// del resultado de A<B
```

Se utilizará un campo de tipo facultativo en los criterios de concordancia de la operación **receive** para evitar toda ambigüedad del tipo del valor que se recibe. Por ejemplo:

```
MyPort.receive(integer:MyVar);
// Especifica la recepción de un valor entero con el mismo valor que la variable
// MyVar en MyPort. El identificador de tipo entero (opcional) no es
// estrictamente necesario porque el tipo ya viene dado por la definición de
// MyVar. Sin embargo, en casos de prueba complejos y largos se puede usar este
// identificador de tipo para mejorar la legibilidad.

MyPort.receive(MyVar);
// Es una alternativa al ejemplo previo.
```

Si la concordancia es satisfactoria, el valor suprimido de la cola de puerto puede ser almacenado en una variable y la dirección del componente que envió el mensaje puede ser extraída y almacenada en una variable. Esto se indica por el símbolo '->' y la palabra clave **value**. Por ejemplo:

```
MyPort.receive(MyType:*) from MyPartner -> value MyVar;
// Especifica la recepción de un valor arbitrario de MyType (de un componente
// con una dirección almacenada en la variable MyPartner) que se asigna después
// a la variable MyVar. MyVar ha de ser del tipo MyType.
```

En el caso de conexiones de uno a muchos, la operación **receive** puede ser restringida a un determinado asociado de comunicación y esta restricción se indicará utilizando la palabra clave **from**.

```
MyPort.receive(charstring:"Hello")from MyPartner;
// Especifica la recepción de la cadena de caracteres "Hello" de un componente
// cuya referencia o dirección está almacenada en la variable MyPartner.
```

Es posible extraer la referencia de componente o dirección del emisor de un mensaje y esto se indica mediante la palabra clave **sender**. Por ejemplo:

```
MyPort.receive(MyTemplate:{5, MyVarOne}) -> value MyVarTwo sender MyPartner;
// Especifica la recepción de un valor que cumple las condiciones definidas por
// la plantilla MyTemplate con parámetros reales 5 y MyVarOne. Después de la
// recepción, el valor se asigna a la variable MyVarTwo. La referencia del
// componente del emisor es extraída por la operación call y asignada a la
// variable MyPartner.

MyPort.receive(A<B) -> sender MyPartner;
// Especifica la recepción de un valor Booleano de verdadero o falso dependiendo
// del resultado de A<B. La reference del componente del emisor es extraída por
// la operación call y asignada a la variable MyPartner.
```

22.2.2.1 Recepción de cualquier mensaje

Una operación **receive** sin lista de argumentos para los criterios de concordancia de tipo y valor del mensaje que se ha de recibir suprimirá el mensaje en la parte superior de la cola puerto entrante (si la hubiere) si se cumplen todos los demás criterios de concordancia.

NOTA – Esto equivale al enunciado OTHERWISE de TTCN-2.

No se asignará una variable a un mensaje recibido mediante *ReceiveAnyMessage*.

Ejemplo:

```
MyPort.receive;
// Suprime el valor más alto de MyPort.MyPort.

MyPort.receive from MyPartner;
// Suprime el valor más alto de CL1 si es un mensaje del componente con la
// referencia de dirección

MyPort.receive -> sender MySenderVar;
// Suprime el valor más alto de CL1, pero recuerda el ejemplar emisor
// almacenando su referencia en MySenderVar
```

22.2.2.2 Recepción en cualquier puerto

Para **recibir** un mensaje en cualquier puerto, se utilizará la palabra clave **any**. Por ejemplo:

```
any port.receive(MyMessage);
```

22.2.3 La operación Trigger

La operación **trigger** filtra los mensajes con determinados criterios de concordancia de un tren de mensajes recibidos por un puerto entrante dado. La operación **trigger** sólo se utilizará en puertos basados en mensaje (o mixtos) y el tipo del valor que se ha de recibir se incluirá en la lista de tipos entrantes de la definición de tipo de puerto. Los mensajes que no cumplen los criterios de concordancia serán suprimidos de la cola sin ninguna otra acción ulterior, es decir, la operación **trigger** espera el siguiente mensaje de la cola. Si un mensaje satisface los criterios de concordancia, la operación **trigger** se comporta de la misma manera que una operación **receive**. Por ejemplo:

```
MyPort.trigger(MyType:*);
// Especifica que la operación se activará al recibir el primer mensaje
// observado del tipo MyType con un valor arbitrario en el puerto MyPort.
```

La operación **trigger** requiere el nombre de puerto, criterios de concordancia para tipo y valor, una restricción **from** facultativa (es decir, selección de asociados de comunicación) y una asignación facultativa del mensaje concordante y el componente emisor a variable.

Ejemplo:

```
MyPort.trigger(MyType:*) from MyPartner;
// Especifica que la operación se activará al recibir el primer mensaje
// observado del tipo MyType con un valor arbitrario en el puerto MyCL
// proveniente de un componente con una referencia idéntica a la almacenada en
// la variable MyPartner.

MyPort.trigger(MyType:*) from MyPartner -> value MyRecMessage;
// Este ejemplo es casi idéntico al anterior. La adición es que el mensaje
// que activa, es decir, se cumplen todos los criterios de concordancia, está
// almacenado en la variable MyRecMessage.

MyPort.trigger(MyType:*) -> sender MyPartner;
// Especifica que la operación se activará al recibir el primer mensaje
// observado del tipo MyType con un valor arbitrario en MyPort. La referencia
// del componente del emisor de este mensaje se almacenará en la variable
// MyPartner.

MyPort.trigger(integer:*) -> value MyVar sender MyPartner;
// Especifica que la operación se activará al recibir un valor entero arbitrario
// que después es almacenado en la variable MyVar y la referencia del componente
// del emisor de este mensaje se almacenará en la variable MyPartner.
```

22.2.3.1 Activación de cualquier mensaje

Una operación **trigger** sin lista de argumentos se activará la recepción de cualquier mensaje. De este modo, su significado es idéntico al de recepción de cualquier mensaje. No se asignará una variable a un mensaje recibido por *TriggerOnAnyMessage*.

Ejemplo:

```
MyPort.trigger;

MyPort.trigger from MyPartner;

MyPort.trigger -> sender MySenderVar;
```

22.2.3.2 Activación en cualquier puerto

Para **activar** un mensaje en cualquier puerto, se utilizará la palabra clave **any**. Por ejemplo:

```
any port.trigger
```

22.2.4 La operación Getcall

La operación **getcall** se utiliza para especificar que un componente de prueba acepta una llamada del SUT, u otro componente de prueba. La operación **getcall** se utilizará solamente en puertos basados en procedimiento (o mixto) y la firma de la llamada de procedimiento que ha de ser aceptada se incluirá en la lista de procedimientos entrantes permitidos de la definición de tipo de puerto.

```
MyPort.getcall(MyProc(5, MyVar));
// Aceptará una llamada de MyProc en MyCL con los parámetros in o inout 5 y
// valor de MyVar.
```

La operación **getcall** suprimirá la llamada de nivel más alto de la cola de puerto entrante, solamente si se cumplen los criterios de concordancia asociados con la operación **getcall**. Estos criterios de concordancia se relacionan con la firma de la llamada que ha de ser procesada y el asociado de comunicación. Los criterios de concordancia para la firma pueden ser especificados en línea o derivados de una plantilla de firma.

Una operación **getcall** puede ser restringida a un determinado asociado de comunicación en el caso de conexiones de uno a muchos y esta restricción se indicará mediante la palabra clave **from**.

```
MyPort.getcall(MyProc:{5, MyVar}) from MyPartner;
// Aceptará una llamada de MyProc en MyCL (con los parámetros in o inout 5 y
// valor de MyVar) de una entidad par con la referencia de dirección o de
// componente almacenada en la variable MyPartner.
```

La parte de asignación de la operación **getcall** comprende la asignación facultativa de valores de parámetros **in** e **inout** a variables y la extracción y asignación de la dirección del componente llamante a una variable.

La palabra clave **param** se utiliza para extraer los valores de parámetro de una llamada. Por ejemplo:

```
MyPort.getcall(MyProc:{5, MyVar}) from MyPartner -> param (MyPar1Var,
MyPar2Var);
// Ambos parámetros de MyProc son inout y sus valores se asignan a MyPar1Var y
// MyPar2Var. La identificación de parámetros definidos en el procedimiento
// de firma y de los nombres en la lista de nombres de variables que siguen a la
// palabra clave param de la operación accept anterior se efectúa por el orden
// en la lista.
```

La palabra clave **sender** se utiliza cuando se requiere extraer la dirección del emisor (por ejemplo, para direccionar una respuesta o excepción a la parte llamante en una configuración de uno a muchos).

```
MyPort.getcall(MyProc:{5, MyVar}) -> sender MySenderVar;
// Aceptará una llamada de MyProc en MyCL con los parámetros in o inout 5 y
// MyVar. La parte llamante es extraída por la operación accept y almacenada
// en MySenderVar. Esto permite tratar llamadas del mismo procedimiento de
// varios componentes en el mismo puerto de la misma manera. MySenderVar se
// puede usar para responder o plantear una excepción al componente llamante.
```

El argumento de firma de la operación **getcall** no se utilizará para introducir nombres de variable para parámetros **in** e **inout**. La asignación de valores de parámetro **in** e **inout** a variables se

efectuará en la parte de asignación de la operación **getcall**. Esto permite utilizar plantillas de firma en operaciones **getcall** de la misma manera que se utilizan plantillas para tipos.

Las siguientes operaciones **getcall** muestran las posibilidades de utilizar atributos de concordancia y omitir partes facultativas, que pueden no ser de importancia para la especificación de la prueba.

Ejemplo:

```
MyPort.getcall(MyProc:{5, MyVar}) -> param(MyPar1Var, MyPar2Var) sender
MySenderVar;

MyPort.getcall(MyProc:{5, *}) -> param(MyPar1Var, MyPar2Var);

MyPort.getcall(MyProc:{*, MyVar}) -> param( - , MyPar2Var);
   // El valor del primer parámetro inout no es importante o no se usa

// Los siguientes ejemplos explicarán las posibilidades de asignar valores de
// parámetros in e inout // a variables. Se supone la siguiente firma para el
// procedimiento que se ha de llamar

signature MyProc2(in integer A, integer B, integer C, out integer D, integer E,
inout integer F);
// MyProc2 no tiene valor devuelto ni excepciones

MyPort.getcall(MyProc2:{*, *, 3, - , - , *}) ->
        param(MyVarIn1, MyVarIn2, MyVarIn3, - , - ,MyVarInout1);
// Los parámetros in A, B y C se asignan a las variables MyVarIn1, MyVarIn2 y
// MyVarIn3 el parámetro inout F se asigna a la variable MyVarInout1. Los
// parámetros out D y E no tienen que ser considerados en la parte de asignación
// de la operación accept.

MyPort.getcall(MyProc2:{*, *, *, - , - , *}) ->
        param(MyVarIn1:=A, MyVarIn2:=B, MyVarIn3:=C, MyVarInout1:=F);
// Notación alternativa para la asignación de valor de parámetros in e inout a
// variables. Nótese que los nombres en la lista de asignación se refieren a los
// nombres usados en la firma de MyProc2

MyPort.getcall(MyProc2:{1, 2, 3, - , - ,*}) -> param(MyVarInout1:=F);
// Sólo se necesita el valor del parámetro inout para la ulterior ejecución del
// caso de prueba
```

22.2.4.1 Aceptación de cualquier llamada

Una operación **getcall** sin lista de argumentos para los criterios de concordancia de firmas suprimirá la llamada en la parte superior de la cola de puerto entrante (si la hubiere) si se cumplen todos los demás criterios de concordancia. Los parámetros de llamada aceptados por *AcceptAnyCall* no se asignarán a una variable.

Ejemplo:

```
MyPort.getcall;
// Suprime la llamada en la cima de MyPort.

MyPort.getcall from MyPartner;
// Suprime la llamada en la cima de CL1 si la parte llamante es una entidad
// con una referencia de dirección o de componente almacenada en la variable
// MyPartner.

MyPort.getcall -> sender MySenderVar;
// Suprime la llamada en la cima de CL1, pero recuerda la parte llamante
// almacenando su referencia de dirección o de componente en la variable
// MySenderVar.
```

22.2.4.2 Obtención de llamada en cualquier puerto

La **obtención** de llamada en cualquier puerto se indica mediante la palabra clave **any**. Por ejemplo:

```
any port.getcall(MyProc)
```

22.2.5 La operación Getreply

La operación **getreply** se utiliza para tratar respuestas de un procedimiento llamado anteriormente y sólo se utilizará en un puerto basado en procedimiento (o mixto). Por ejemplo:

```
MyPort.getreply(MyProc:{5, MyVar} value 20);
// Acepta una respuesta de procedimiento MyProc donde el valor devuelto es 20 y
// los valores de los dos parámetros out o inout es 5 y el valor de MyVar.

MyPort.getreply (MyProc2:{ - , 5});
// Acepta una respuesta de MyProc2. MyProc2 no tiene valor devuelto sino dos
// parámetros. El primero es un parámetro in, su valor no será respondido y
// por tanto no será considerado para concordancia. El segundo parámetro es
// out o inout. Su valor tiene que ser 5.
```

Se puede emplear en la parte **getreply** y de excepción de una llamada, por ejemplo:

```
MyPort.call (MyProc) to MyPeer
{
   [ ] MyPort.getreply(MyProc:*) {}
   [ ] MyPort.catch {}
}
```

o dentro de un enunciado **alt**, por ejemplo:

```
MyPort.call (MyProc, nowait) to MyPeer;
:
alt
{
   [ ] MyPort.getreply(MyProc:*) {}
   :
}
```

Si se utiliza en un enunciado, **alt**, **getcall** debe abarcar los casos cuando la respuesta de un procedimiento llamado previamente llega demasiado tarde, es decir, se ha planteado una excepción de temporización.

Al igual que en otras operaciones receptoras, se permiten mecanismos de concordancia en la operación **getreply** para distinguir entre respuestas de un procedimiento llamado previamente que difieren en el valor devuelto y/o el valor de los parámetros **out** e **inout**.

```
MyPort.getreply(MyProc1:{*, MyVar});
// En este ejemplo no hay restricción impuesta al valor devuelto ni al valor
// del primer parámetro.

MyPort.getreply(MyProc1:{*, *});
// La operación getreply concordará con cualquier respuesta de MyProc1 con
// cualquier valor devuelto. Las estrellas son definiciones de plantilla en
// línea para MyProc1 y el tipo devuelto de MyProc1.
```

En casos de conexiones de uno a muchos, la operación **getreply** permite distinguir entre diferentes asociados de comunicación utilizando una cláusula **from**.

```
MyPort.getreply(MyProc2:{ - ,5}) from MyPartner;
// La respuesta solo es aceptada si es de un componente con la referencia
// especificada en la variable MyPartner
```

La parte de asignación facultativa de la operación **getreply** permite asignar valores de parámetros **out** e **inout** y valores devueltos a variables.

Ejemplo:

```
MyPort.getreply(MyProc1:{*, *} value *) -> value MyReturnValue
        param(MyPar1,MyPar2);
// Tras la aceptación, el valor devuelto es asignado a la variable MyReturnValue
// y el valor de los dos parámetros out o inout se asigna a las variables MyPar1
// y MyPar2.

MyPort.getreply(MyProc1:{*, *} value *) -> value MyReturnValue param( - ,
MyPar2)
   sender MySender;
// El valor del primer parámetro no se considera para la ulterior ejecución de
// la prueba y la referencia de dirección o de componente de la entidad de la
// cual se ha recibido la respuesta se almacena en la variable MySender.

// Los siguientes ejemplos describen algunas posibilidades de asignar valores de
// parámetros out e inout a variables. Se supone la siguiente firma para el
// procedimiento llamado
signature MyProc2(in integer A, integer B, integer C, out integer D, integer E,
   inout integer F);
// Nótese que MyProc2 no tiene valor devuelto ni excepciones

MyPort.getreply(MyProc2:*) -> param( - , - , - , MyVarOut1, MyVarOut2, - ,
   MyVarInout1)
// Los parámetros in D y E se asignan a las variables MyVarOut1 y MyVarOut2, el
// parámetro inout F se asigna a la variable MyVarInout1.

MyPort.getreply(MyProc2:*) -> param(MyVarOut1:=D, MyVarOut2:=E, MyVarInout1:=F);
// Notación alternativa para la asignación de valor de los parámetros in e inout
// a variables. Nótese que los nombres en la lista de asignaciones hacen
// referencia a los nombres usados en la firma de MyProc2

MyPort.getreply(MyProc2:{ - , - , - , 3, *, *}) -> param(MyVarInout1:=F);
// Sólo se necesita el valor del parámetro inout para la ulterior ejecución del
// caso de prueba
```

22.2.5.1 Obtención de cualquier respuesta de cualquier llamada

Una operación **getreply** sin lista de argumentos para los criterios de concordancia de firma suprimirá un mensaje **reply** en la parte superior de la cola de puerto entrante (si la hubiere) si se cumplen todos los demás criterios de concordancia. Los parámetros o valores devueltos de respuestas aceptadas por *GetAnyReply* no se asignarán a una variable.

Ejemplo:

```
MyPort.getreply;
// Suprime la respuesta en la cima de MyPort.

MyPort.getreply from MyPartner;
// Suprime la respuesta en la cima de CL1 si la parte respondedora es una
// entidad con la referencia de dirección o componente almacenada en la variable
// MyPartner.

MyPort.getreply -> sender MySenderVar;
// Suprime la respuesta en la cima de CL1, pero recuerda la parte responderá
almacenándola en la variable MySenderVar
```

22.2.5.2 Obtención de una respuesta en cualquier puerto

Para obtener una respuesta en cualquier puerto se utilizará la palabra clave **any**. Por ejemplo:

```
any port.getreply(Myproc)
```

22.2.6 La operación Catch

La operación **catch** se utiliza para capturar excepciones planteadas por una entidad par como reacción a una llamada de procedimiento. La operación **catch** se utilizará solamente en puertos basados en procedimiento (o mixtos). El tipo de la excepción capturada se especificará en la firma para el procedimiento que planteó la excepción.

```
MySyncPort.catch(MySignature, integer: MyVar);
// Especifica la captura de una excepción planteada por una procedimiento con
// una firma Mysignature en el puerto MySyncPort. La excepción es un valor
// entero que tiene el mismo valor que la variable MyVar. El identificador de
// tipo entero (opcional) no es estrictamente necesario porque el tipo ya viene
// dado por la definición de MyVar. Sin embargo, en casos de prueba complejos y
// largos, este identificador de tipo puede ser usado para mejorar la
// legibilidad.

MySyncPort.catch(MySignature, MyVar);
// Es una alternativa al ejemplo previo.

MySyncPort.catch(MySignature, A<B);
// Captura una excepción Booleana de verdadero o falso dependiendo del resultado
// de A<B planteado por un procedimiento MySignature en el puerto MySyncPort.
```

La operación **catch** puede formar parte de la parte aceptadora de un llamada o ser utilizada para determinar una alternativa en un enunciado **alt**. Si la operación **catch** se utiliza en la parte aceptadora de una operación **call**, la información sobre nombre de puerto y referencia de firma para indicar el procedimiento que planteó la excepción es redundante, porque esta información se deduce de la operación **call**. Sin embargo, por motivos de legibilidad (por ejemplo, en el caso de enunciados de **call** complejos) se repetirá esta información.

Las excepciones se especifican como tipos y por tanto pueden ser tratadas como mensajes, por ejemplo, se pueden utilizar plantillas para distinguir entre diferentes valores del mismo tipo de excepción.

```
MySyncPort.catch(MySignature, MyTemplate:{5, MyVar});
// Captura una excepción planteada por un procedimiento con una firma
// Mysignature en el puerto MySyncPort que cumple las condiciones definidas por
// la plantilla MyTemplate con parámetros reales 5 y MyVar.
```

La operación **catch** requiere el nombre de puerto, criterios de concordancia para tipo y valor, una restricción **from** facultativa (es decir, selección de asociado de comunicación) y una asignación facultativa de la excepción concordante y el componente **sender** a variables. Por ejemplo:

```
MySyncPort.catch(MySignature, charstring:"Hello")from MyPartner;
// Captura la cadena IA5 "Hello" planteada por un procedimiento con una firma
// Mysignature en el puerto MySyncPort de una entidad con una referencia de
// dirección o de componente almacenada en MyPartner.

MySyncPort.catch(MySignature, MyType:*) from MyPartner -> value MyVar;
// Captura una excepción con un valor arbitrario de MyType (planteado por un
// procedimiento con una firma Mysignature en el puerto MySyncPort de un
// componente con una referencia almacenada en la variable MyPartner) que es
// asignada después a la variable MyVar. MyVar ha de ser del tipo MyType.
```

```
MySyncPort.catch(MySignature, MyTemplate (5, MyVarOne)) -> value MyVarTwo sender
   MyPartner;
// Captura una excepción planteada por un procedimiento cuya firma Mysignature
// con un valor que cumple las condiciones definidas por la plantilla MyTemplate
// con parámetros reales 5 y MyVarOne. Después la excepción se asigna a
// MyVarTwo. La dirección o referencia de la entidad emisora es extraída por la
// operación catch y asignada a MyPartner.
```

22.2.6.1 La excepción Timeout

Hay una excepción especial **timeout** que es capturada por la operación **catch**. La excepción **timeout** es una salida de emergencia cuando un procedimiento llamado no responde ni plantea una excepción dentro de un plazo predeterminado. Por ejemplo.

```
MyPort.catch(timeout); // Captura una excepción de temporización.
```

La captura de excepciones **timeout** será restringida a la parte de tratamiento de excepciones de una llamada. No se permiten otros criterios de concordancia (incluida una parte **from**) y ninguna parte de asignación para una operación **catch** que trata una excepción **timeout**.

22.2.6.2 Captura de cualquier excepción

Una operación **catch** sin lista de argumentos permite capturar cualquier excepción válida. El caso más general es sin utilizar la palabra clave **from** y sin una parte de asignación. Este enunciado capturará también la excepción **timeout**. Por ejemplo:

```
MyPort.catch;

MyPort.catch from MyPartner;

MyPort.catch -> sender MySenderVar;
```

22.2.6.3 Captura en cualquier puerto

Para **capturar** una excepción en cualquier puerto se utiliza la palabra clave **any**. Por ejemplo:

```
any port.catch(timeout)
```

22.2.7 La operación Check

La operación **check** es una operación genérica que permite leer el acceso al elemento más alto de las colas de puertos *entrantes* basados en mensaje y basados en procedimiento sin suprimir dicho elemento de la cola. La operación **check** tiene que tratar valores de un cierto tipo en puertos basados en mensaje y distinguir entre llamadas que han de ser aceptadas, excepciones que han de ser capturadas y respuestas de llamadas previas en puertos basados en procedimiento.

Las operaciones receptoras **receive**, **getcall**, **getreply** y **catch** juntas con sus partes de concordancia y de asignación son utilizadas por la operación **check** para definir la condición que ha de ser comprobada y extraer el valor o valores de sus parámetros, si es necesario.

```
MyAsyncPort.check(receive(integer: 5));
// Comprobará un valor entero de 5 como mensaje en la cima en el puerto
// asíncrono MyAsyncPort.

MyPort.check(getcall(MyProc:{5, MyVar}) from MyPartner);
// Comprobará una llamada de MyProc en MyCL (con los parámetros in r inout 5 y
// MyVar) de una entidad par con una referencia de dirección o de componente
// almacenada en la variable MyPartner.

MyPort.check(getreply(MyProc:{5, MyVar} value 20));
// Comprobará una respuesta de procedimiento MyProc en MyPort donde el valor
// devuelto es 20 y los valores de los dos parámetros out o inout es 5 y el
// valor de MyVar.
```

```
ySyncPort.check(catch(MySignature, MyTemplate (5, MyVar)));
// Comprueba una excepción planteada por una procedimiento con una firma
// Mysignature en el puerto MySyncPort que cumple las condiciones definidas por
// la plantilla MyTemplate con parámetros reales 5 y MyVar.
```

Se comprobará el elemento en la *cima* de una cola de puerto entrante (no es posible mirar *dentro de* la cola). Si la cola está vacía, la operación **check** fracasa. Si la cola no está vacía, se hace una copia del elemento en la cima y la operación receptora especificada en la operación **check** se ejecuta en la copia. La operación **check** fracasa si la función receptora fracasa, es decir, no se cumplen los criterios de concordancia. En este caso, la *copia* del elemento en la cima de la cola es descartada y la ejecución de la prueba continúa de la manera normal, es decir, se evalúa la siguiente alternativa de la operación. La operación **check** es fructuosa si la función receptora es fructuosa.

La utilización de la operación **check** de manera errónea, por ejemplo, comprobar una excepción en un puerto basado en mensaje, originará un error de caso de prueba.

NOTA – En la mayoría de los casos, es posible comprobar estáticamente el uso correcto de la operación **check**, es decir, antes de la compilación.

Ejemplo:

```
MyPort.check(getreply(MyProc1:{*, MyVar} value *) -> value MyReturnValue
   param(MyPar1));
// En este ejemplo el valor devuelto es asignado a la variable MyReturnValue y
// el valor del primer parámetro out o inout se asigna a la variable MyPar1.

MyPort.check(getcall(MyProc:{5, MyVar}) from MyPartner -> param (MyPar1Var,
   MyPar2Var));
// En este ejemplo ambos parámetros de MyProc son considerados parámetros y
// sus valores se asignan a MyPar1Var y MyPar2Var.

MyPort.check(getcall(MyProc:{5, MyVar}) -> sender MySenderVar);
// Aceptará una llamada de MyProc en MyCL con los parámetros in o inout 5 y
// MyVar. La parte llamante es extraída y almacenada en MySenderVar.
```

22.2.7.1 La operación Check any

Una operación **check** sin lista de argumento permite comprobar si algo espera procesamiento en una cola de puerto entrante. La operación *CheckAny* permite distinguir entre diferentes emisores (en el caso de conexiones de uno a muchos) utilizando una cláusula **from** y extraer el emisor utilizando una parte de asignación abreviada con una cláusula **sender**.

Ejemplo:

```
MyPort.check;

MyPort.check(from MyPartner);

MyPort.check(-> sender MySenderVar);
```

22.3 Control de puertos de comunicación

Las operaciones de TTCN-3 para controlar puertos basados en mensaje, basados en procedimiento y mixtos son:

- **clear**: suprimir el contenido de una cola de puerto entrante;
- **start**: comenzar la escucha y dar acceso a un puerto;
- **stop**: detener la escucha y prohibir operaciones emisoras en un puerto.

22.3.1 La operación Clear port

La operación **clear** suprime el contenido de la cola *entrante* del puerto denominado. Si la cola del puerto ya está vacía, esta operación no tendrá ninguna acción.

```
MyPort.clear; // Libera puerto MyPort
```

22.3.2 La operación Start port

Si se define que un puerto permite operaciones receptoras, tales como **receive**, **getcall** etc., la operación **start** suprime la cola entrante del puerto denominado y comienza la escucha para el tráfico por el puerto. Si se ha definido que el puerto permite operaciones emisoras, se permite también ejecutar en ese puerto operaciones tales como **send**, **call**, **raise** etc. Por ejemplo:

```
MyPort.start; // Arranca MyPort
```

Por defecto, todos los puertos de un componente serán comenzados cuando un componente comienza la ejecución.

22.3.3 La operación Stop port

Si se ha definido que un puerto permite operaciones receptoras, tales como **receive**, **getcall,** etc., la operación **stop** detiene la escucha en el puerto denominado. Si se ha definido que el puerto permite operaciones emisoras, **stop** port prohíbe la ejecución de operaciones tales como **send**, **call**, **raise**, etc. Por ejemplo:

```
MyPort.stop;  // Detiene MyPort
```

22.4 Utilización de Any y All con puertos

Las palabras claves **any** y **all** pueden ser utilizadas con las operaciones de configuración indicadas en el cuadro 16.

Cuadro 16/Z.140 – Any y all con puertos

Operación	Permitido		Ejemplo
	any	**all**	
Recibir operaciones de comunicación (**receive**, **trigger**, **getcall**, **getreply**, **catch**, **check**)	Sí		**any port.receive**
connect / map			
start		Sí	**all port.start**
stop		Sí	**all port.stop**
clear		Sí	**all port.clear**

23 Operaciones de temporizador

TTCN-3 soporta varias operaciones de temporizador (véase el cuadro 17) que pueden ser utilizadas en casos de prueba, funciones y en control de módulo.

Cuadro 17/Z.140 – Visión general de las operaciones de temporizador de TTCN-3

Operaciones de temporizador	
Enunciado	**Palabra clave o símbolo asociados**
Arrancar temporizador	**start**
Detener temporizador	**stop**
Leer tiempo transcurrido	**read**
Comprobar si el temporizador está funcionando	**running**
Evento expiración de temporización	**timeout**

23.1 La operación Start timer

La operación **start** timer se utiliza para indicar que un temporizador debe arrancar. Los valores de temporizador serán de tipo **float**. Por ejemplo:

```
MyTimer1.start;        // MyTimer1 es arrancado con la duración por defecto
MyTimer2.start(20E-3); // MyTimer2 es arrancado con la duración de 20 ms.
```

El parámetro de valor de temporizador facultativo se utilizará si no se indica duración por defecto, o si se desea revocar el valor por defecto especificado en la declaración de temporizador. Cuando se revoca una duración de temporizador, el nuevo valor se aplica solamente al caso vigente del temporizador, y cualesquiera operaciones **start** ulteriores para este temporizador que no especifiquen una duración, utilizarán la duración por defecto. El reloj del temporizador funciona a partir del valor de coma flotante cero (0,0) hasta un máximo indicado por el parámetro de duración.

23.2 La operación Stop timer

La operación **stop** se utiliza para detener un temporizador en funcionamiento y suprimirlo de la lista de temporizadores en funcionamiento. Un temporizador detenido pasa a estar inactivo y el tiempo transcurrido se fija al valor de coma flotante cero (0.0). Si el nombre del temporizador en la operación **stop** es **all**, se detienen todos los temporizadores en funcionamiento (es decir, activos). Por ejemplo:

```
MyTimer1.stop;      // Detiene MyTimer1
all timer.stop;     // Detiene todos los temporizadores en funcionamiento
```

La detención de un temporizador inactivo es una operación válida, aunque no tiene ningún efecto.

23.3 La operación Read timer

La operación **read** se utiliza para extraer el tiempo que ha transcurrido desde que se arrancó el temporizador especificado y almacenarlo en la variable especificada. Esta variable será de tipo **float**. Por ejemplo:

```
var float Myvar;
MyVar := MyTimer1.read;   // Asigna a MyVar el tiempo transcurrido desde que se
                          // arrancó MyTimer1
```

La aplicación de la operación **read** en un temporizador inactivo devolverá el valor cero.

23.4 La operación Running timer

La operación **running** se utiliza para verificar si un temporizador está funcionando o no (es decir, que ha sido arrancado y que no ha expirado ni ha sido cancelado). La operación devuelve el valor **true** si el temporizador está funcionando y **false** en los demás casos. Por ejemplo:

```
if (MyTimer1.running) { ... }
```

23.5 El evento Timeout

La operación **timeout** indica la expiración de un temporizador arrancado previamente. La operación **timeout** se puede utilizar en alternativas junto con las operaciones **receive**, **getcall**, **getreply**, **catch** y **other timeout**.

Ejemplo:

```
MyTimer1.timeout;   // Comprueba la expiración de MyTimer1 arrancado previamente
```

Se utiliza la palabra clave **any** para indicar la **expiración** de cualquier temporizador (en vez de un temporizador denominado explícitamente) arrancado dentro del plazo de la temporización. Por ejemplo:

```
any timer.timeout;  // comprueba la expiración de cualquier temporizador
                    // arrancado previamente
```

23.6 Utilización de any y all con temporizadores

Las palabras clave **any** y **all** pueden ser utilizadas con operaciones de temporizador como se indica en el cuadro 18.

Cuadro 18/Z.140 – Any y all con temporizadores

Operación	Permitido		Ejemplo
	any	**all**	
start			
stop		Sí	**All timer.stop**
read			
running	Sí		**if (any timer.running) { ... }**
timeout	Sí		**Any timer.timeout**

24 Operaciones de veredicto de prueba

Las operaciones de veredicto (véase el cuadro 19) permiten fijar y extraer veredictos utilizando respectivamente las operaciones **get** y **set**. Estas operaciones sólo se utilizarán en casos de prueba y funciones.

Cuadro 19/Z.140 – Visión general de operaciones de veredicto de prueba de TTCN-3

Operaciones de veredicto de prueba	
Enunciado	**Palabra clave o símbolo asociado**
Fijar veredicto local	**Verdict.set**
Obtener veredicto local	**Verdict.get**

Cada componente de prueba de la configuración activa mantendrá su propio veredicto local, que es un objeto que es creado para cada componente de prueba en el momento de su ejemplificación. Se utiliza para seguir el veredicto individual en cada componente de prueba, (es decir, en el MTC y en cada uno de los PTC).

NOTA – A diferencia de TTCN-2, la asignación de un veredicto final no detendrá la ejecución del componente de prueba en el cual se está ejecutando el comportamiento. Si es necesario, esto se hará explícitamente utilizando el enunciado **stop**.

24.1 Veredicto de caso de prueba

Hay además un veredicto global que es actualizado cuando cada componente de prueba (es decir el MTC y cada uno de los PTC) termina la ejecución. Este veredicto no es accesible a las operaciones **get** y **set**. El valor de este veredicto será devuelto por el caso de prueba cuando termina la ejecución. Si el veredicto devuelto no es guardado explícitamente en la parte de control (por ejemplo, asignado a una variable), se pierde.

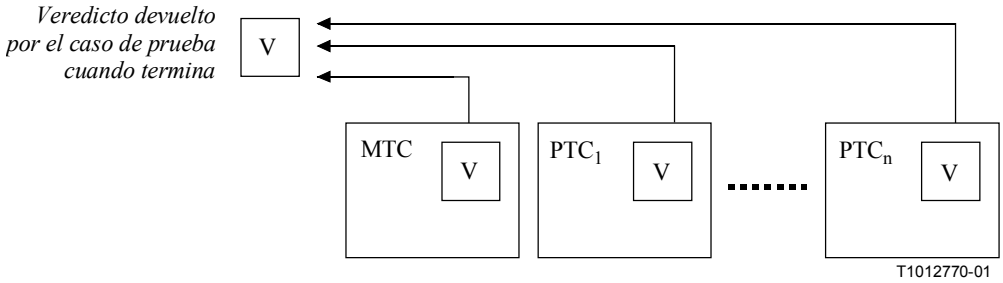


Figura 10/Z.140 – Ilustración de la relación entre veredictos

NOTA – TTCN-3 no especifica los mecanismos reales que ejecutan la actualización de los veredictos local y de casos de prueba, porque estos mecanismos dependen de la implementación.

24.2 Valores de veredicto y reglas de sobrescritura

El veredicto puede tener cinco valores diferentes: **pass**, **fail**, **inconc**, **none** y **error** . Es decir, los valores distinguidos del **verdicttype** (véase 6.1).

NOTA – **inconc** significa un veredicto no concluyente.

La operación **set** sólo se utilizará con los valores **pass**, **fail**, **inconc** y **none**. Por ejemplo:

```
verdict.set(pass);
verdict.set(inconc);
```

El valor del veredicto local puede ser extraído utilizando la operación **get**. Por ejemplo:

```
MyResult := verdict.get; // Donde MyResult es una variable de tipo verdicttype
```

Cuando se ejemplifica un componente de prueba, se crea su objeto de veredicto local y se pone al valor **none**.

Cuando se cambia el valor del veredicto (es decir, utilizando la operación **set**), el efecto de este cambio seguirá las reglas de sobrescritura del cuadro 20. El veredicto de caso de prueba se fija implícitamente en la terminación de un componente de prueba. El efecto de esta operación implícita seguirá también las reglas de sobrescritura indicadas en el cuadro 20.

Cuadro 20/Z.140 – Reglas de sobrescritura para el veredicto

Valor vigente de veredicto	Nuevo valor de asignación de veredicto			
	pass (éxito)	inconc (no concluyente)	fail (fracaso)	none (ninguno)
none	pass	inconc	fail	none
pass	pass	inconc	fail	pass
inconc	inconc	inconc	fail	inconc
fail	fail	fail	fail	fail

Ejemplo:

```
:
verdict.set(pass);  // El veredicto local se pone a éxito
:
verdict.set(fail);  // Hasta que se ejecuta esta línea, lo que resultará en el
:                   // valor de la sobreescritura del veredicto local de
:                   // fracaso.
:                   // Cuando el ptc termina el veredicto de caso de prueba se
:                   // pone a fracaso
```

24.2.1 Veredicto de error

El veredicto **error** es especial en cuanto a que es fijado por el sistema de prueba para indicar que se ha producido un error de caso de prueba (es decir, de tiempo de ejecución). No será fijado por la operación **set**. Ningún otro valor de veredicto puede revocar un veredicto **error**. Esto significa que un veredicto **error** sólo puede ser el resultado de una operación de caso de prueba **execute**.

25 Operaciones del SUT

En algunas situaciones de prueba puede no haber una interfaz explícita con el SUT y puede ser necesario que el SUT inicie ciertas acciones (por ejemplo, enviar un mensaje al sistema de prueba).

Esta acción se puede definir como una cadena, por ejemplo:

```
sut.action("Send MyTemplate on lower PCO");   // Descripción informal de la
                                              // acción del SUT
```

o como una referencia a una plantilla que especifica la estructura del mensaje que ha de ser enviado por el SUT, por ejemplo:

```
sut.action(MyTemplate);   // Esto equivale al enunciado IMPLICIT SEND de TTCN-2.
```

En ambos casos no hay especificación de lo que hay que hacer al SUT o de lo que el SUT ha de hacer para activar esta acción, solamente hay una especificación informal de la reacción requerida.

Las acciones del SUT pueden ser especificadas en casos de prueba, funciones, alternativas denominadas y control de módulo.

26 Parte de control de módulo

Los casos de prueba se definen en las definiciones de módulo y se ejecutan en el control de módulo. Todas las variables, temporizadores, etc. (si hubiere) definidos en la parte de control de un módulo serán introducidos en el caso de prueba mediante parametrización si han de ser utilizados en la definición de comportamiento de ese caso de prueba, es decir, TTCN-3 no soporta variables globales de ninguna clase.

Al comienzo de cada caso de prueba, se reiniciará la configuración de prueba, lo que significa que todas las operaciones **create**, **connect**, etc., que puedan haber sido ejecutadas en un caso de prueba anterior no son "visibles" para el nuevo caso de prueba.

26.1 Ejecución de casos de prueba

Un caso de prueba se invoca utilizando un enunciado **execute**. Como resultado de la ejecución de un caso de prueba, se devolverá un veredicto de prueba o **none**, **pass**, **inconclusive**, **fail** o **error** y se puede asignar a una variable para ulterior procesamiento.

Facultativamente, el enunciado **execute** permite la supervisión de un caso de prueba por medio de una temporización. Si el caso de prueba no termina dentro de esta duración, el resultado de la

ejecución del caso de prueba será un veredicto de error y el sistema de prueba terminará el caso de prueba.

Ejemplo:

```
execute(MyTestCase1());      // Ejecuta MyTestCase1, sin almacenar el veredicto
                             // de prueba devuelto y sin supervisión de tiempo

MyVerdict := execute(MyTestCase2()); // Ejecuta MyTestCase2 y almacena el
                                     // veredicto resultante en la variable
                                     // MyVerdict

MyVerdict := execute(MyTestCase3(),5E-3);  // Ejecuta MyTestCase3 y almacena el
                           // veredicto resultante en la variable MyVerdict. Si
                           // el caso de prueba no termina dentro de 5 ms,
                           // MyVerdict obtendrá el valor 'error'
```

26.2 Terminación de casos de prueba

Un caso de prueba finaliza con la terminación del MTC. Tras la terminación del MTC, todos los componentes de prueba paralelos en funcionamiento serán terminados por el medio de prueba (es decir, el sistema de prueba).

NOTA 1 – El mecanismo concreto para detener todos los PTC es específico de la herramienta, por lo que está fuera del ámbito de la presente Recomendación.

El veredicto final de un caso de prueba se calcula sobre la base de los veredictos locales finales de los diferentes componentes de prueba, de acuerdo con las reglas definidas en la cláusula 24. El veredicto local real de un componente de prueba se convierte en su veredicto local final cuando el componente de prueba termina por sí mismo o es detenido por el medio de prueba (es decir, el sistema de prueba).

NOTA 2 – Para evitar condiciones de competencia en el cálculo del veredicto de prueba debido a la detención retardada de los PTC, el MTC debe asegurar que todos los PTC están detenidos (por medio del enunciado **done**) antes de detenerse.

26.3 Control de la ejecución de casos de prueba

Se pueden utilizar enunciados de programa, limitados a los definidos en el cuadro 11, en la parte de control de un módulo para especificar el orden de ejecución de las pruebas o el número de veces que un caso de prueba puede ser ejecutado. Por ejemplo:

```
module MyTestSuite
{  :
   control
   {  :

      // Hacer esta prueba 10 veces
      count:=0;
      while (count < 10)
      {  execute (MySimpleTestCase1());
         count := count+1;
      }
   }
}
```

Si no se utilizan enunciados de programación entonces, por defecto, los casos de prueba son ejecutados en el orden secuencial en el cual aparecen en el control de módulo.

NOTA – Esto no excluye la posibilidad de que ciertas herramientas deseen revocar este orden por defecto para que un usuario o herramienta pueda seleccionar un orden de ejecución diferente.

Los casos de prueba devuelven un solo valor de tipo **verdicttype**, por lo que es posible controlar el orden de ejecución dependiendo del resultado de un caso de prueba. Por ejemplo:

```
if (MySimpleTestCase() == pass) { log("Success!") }
```

26.4 Selección de casos de prueba

Se pueden utilizar expresiones booleanas para seleccionar y deseleccionar los casos de prueba que han de ser ejecutados. Esto incluye naturalmente la utilización de funciones que devuelven un valor **boolean**.

NOTA – Esto equivale a las expresiones de selección de pruebas denominadas de TTCN-2.

Ejemplo:

```
module MyTestSuite
{  :
   control
   {  :
      if (MySelectionExpression1())
      {  execute(MySimpleTestCase1());
         execute(MySimpleTestCase2());
         execute(MySimpleTestCase3());
      }
      if (MySelectionExpression2())
      {  execute(MySimpleTestCase4());
         execute(MySimpleTestCase5());
         execute(MySimpleTestCase6());
      }
   :
   }
}
```

Otra manera de ejecutar casos de prueba como un grupo es recopilarlos en una función y ejecutar esa función a partir del control de módulo. Por ejemplo:

```
:
function MyTestCaseGroup1()
{  execute(MySimpleTestCase1());
   execute(MySimpleTestCase2());
   execute(MySimpleTestCase3());
}
function MyTestCaseGroup2()
{  execute(MySimpleTestCase4());
   execute(MySimpleTestCase5());
   execute(MySimpleTestCase6());
}
:
control
{  if (MySelectionExpression1()) { MyTestCaseGroup1(); }
   if (MySelectionExpression1()) { MyTestCaseGroup2(); }
   :
}
:
```

26.5 Utilización de temporizadores en control

Es posible utilizar temporizadores para controlar la ejecución de casos de prueba. Esto se puede hacer utilizando una temporización explícita en el enunciado de ejecución. Por ejemplo:

```
MyReturnVal := execute (MyTestCase(), 7E-3); // Variable de verdicttype
// Cuando el veredicto devuelto es error si el TestCase no completa la ejecución
// dentro de 7 ms
```

Es posible también utilizar operaciones de temporizador. Por ejemplo:

```
// Ejemplo del uso de la operación running timer
while (T1.running or x<10)   // Donde T1 es un temporizador arrancado
                             // previamente
{  execute(MyTestCase());
   x := x+1;
}

// Ejemplo del uso de las operaciones start y timeout

timer T1 := 1;
:
execute(MyTestCase1());
T1.start;
T1.timeout; // Pausa antes de ejecutar el siguiente caso de prueba
execute(MyTestCase2());
```

27 Especificación de atributos

Los atributos pueden ser asociados con elementos de lenguaje de TTCN-3 por medio del enunciado **with**. La sintaxis para el argumento del enunciado **with** (es decir, los atributos reales) se define sencillamente como una cadena de texto libre.

Hay tres clases de atributos:

a) **display**: permite la especificación de atributos de visualización relacionados con formatos de presentación específicos;

b) **encode**: permite referencias a reglas de codificación específicas;

c) **extension**: permite la especificación de atributos definidos por el usuario.

27.1 Atributos de visualización

Todos los elementos de lenguaje de TTCN-3 pueden tener atributos **display** para especificar cómo pueden ser visualizados determinados elementos de lenguaje, por ejemplo, un formato gráfico.

En UIT-T Z.141 [1] figuran cadenas de atributos especiales relacionadas con los atributos de visualización para el formato de presentación tabular (conformidad).

En UIT-T Z.142 [2] figuran cadenas de atributos especiales relacionadas con los atributos de visualización para el formato de presentación gráfica.

Otros atributos **display** pueden ser definidos por el usuario.

NOTA – Como los atributos definidos por el usuario no están normalizados, la interpretación de estos atributos entre herramientas suministradas por diferentes fabricantes puede diferir o incluso no ser soportada.

27.2 Atributos de codificación

Las reglas de codificación definen cómo se codifica y transmite un determinado valor como plantilla, etc., usualmente como un tren de bits, por un puerto de comunicación. TTCN-3 no tiene un mecanismo de codificación por defecto, lo que significa que las reglas de codificación o directrices de codificación se definen de alguna manera externa a TTCN-3.

El atributo **encode** permite asociar alguna regla o directiva de codificación referenciada con definiciones de tipo TTCN-3 (y con una definición de tipo solamente).

En el anexo E figuran cadenas de atributos especiales relacionadas con atributos de codificación de ASN.1.

La manera en que se definen las reglas de codificación reales (por ejemplo prosa, funciones, etc.) está fuera del alcance de la presente Recomendación. Si no se hace referencia a reglas específicas, la codificación será un asunto de cada implementación.

En la mayoría de los casos, los atributos de codificación se utilizarán de manera jerárquica. El nivel más alto es todo el módulo, el siguiente nivel es un grupo de tipos y el más bajo es un tipo:

a) **`module`**: la codificación se aplica a todos los tipos definidos en el módulo, incluidos los tipos básicos TTCN-3;

b) **`group`**: la codificación se aplica a un grupo de definiciones de tipos definidos por el usuario;

c) **`type`**: la codificación se aplica a un solo tipo definido por el usuario;

d) **`field`**: la codificación se aplica a un campo en un tipo **`record`** o **`set`**;

Ejemplo:

```
module MyTTCNmodule
{  :
   import type MyRecord from MySecondModule with {encode "MyRule 1"}
         // Todos los ejemplares de MyRecord se codificarán de acuerdo con
         // MyRule 1
      :
   type charstring MyType;   // Normalmente codificado de acuerdo con la regla
                             // global
   :
   group MyRecords
   {  :
      type record MyPDU1
      {
         integer     field1,  // field1 se codificará de acuerdo con la Regla 3
         boolean     field2,  // field2 se codificará de acuerdo con la Regla 3
         Mytype      field3   // field3 se codificará de acuerdo con la Regla 2
      }
      with {encode (field1, field2) "Rule 3"}
      :
   }
   with {encode "Rule 2"}

}
with {encode "Global encoding rule"}
```

27.2.1 Codificaciones no válidas

Si se desea especificar reglas de codificaciones no válidas, éstas serán especificadas en una fuente referenciable externa al módulo, de la misma manera que se hace referencia a reglas de codificación válidas.

27.3 Atributos de extensión

Todos los elementos de lenguaje de TTCN-3 pueden tener atributos de `extension` especificados por el usuario.

NOTA – Los atributos definidos por el usuario no están normalizados, por lo que la interpretación de estos atributos entre herramientas suministradas por distintos fabricantes puede diferir o incluso no ser soportada.

27.4 Alcance de atributos

Un enunciado **`with`** asocia siempre atributos con elementos de lenguaje individuales. También es posible asociar atributos con varios elementos del lenguaje asociando un enunciado **`with`** con la unidad de ámbito o **`group`** de elementos de lenguaje circundantes.

El enunciado **with** sigue las reglas de alcance definidas en 5.4, es decir, un enunciado **with** que se coloca dentro del alcance de otro enunciado **with** revocará el enunciado **with** más externo. Esto se aplicará también a la utilización del enunciado **with** con grupos. Se ha de tener cuidado cuando se emplea el esquema de sobrescritura en combinación con referencias a definiciones individuales. La regla general es que los atributos serán asignados y sobrescritos de acuerdo con su orden de aparición.

Ejemplo:

```
// MyPDU1 se visualizará como PDU
type record MyPDU1 { ... } with { display "PDU"}

// MyPDU2 se visualizará como PDU con el atributo de extensión específico de
// aplicación MyRule
type record MyPDU2 { ... }
with
{
   display "PDU";
   extension "MyRule"
}

// La siguiente definición de grupo …
group MyPDUs {
   type record MyPDU3 { ... }
   type record MyPDU4 { ... }
}
with {display "PDU"}   // Todos los tipos de MyPDUs se visualizarán como PDU

// es idéntico a
   type record MyPDU3 { ... } with { display "PDU"}
   type record MyPDU4 { ... } with { display "PDU"}
}

// Ejemplo del uso del esquema de sobreescritura del enunciado with
group MyPDUs
{
   type record MyPDU1 { ... }
   type record MyPDU2 { ... }

   group MySpecialPDUs
   {
      type record MyPDU3 { ... }
      type record MyPDU4 { ... }
   }
   with {extension "MySpecialRule"}  // MyPDU3 and MyPDU4 tendrán el atributo
                                     // de extensión específico de la aplicación
                                     // MySpecialRule
}
with
{
   display "PDU";      // Todos los tipos de MPDUs se visualizarán como PDU y
   extension "MyRule"; // (si no está sobreescrito) tendrá el atributo de
                       // extensión MyRule
   }

// es idéntico a ...
group MyPDUs
{
   type record MyPDU1 { ... } with {display "PDU"; extension "MyRule" }
   type record MyPDU2 { ... } with {display "PDU"; extension "MyRule" }
   group MySpecialPDUs {
      type record MyPDU3 { ... } with {display "PDU"; extension "MySpecialRule"
}
```

```
      type record MyPDU4 { ... } with {display "PDU"; extension "MySpecialRule"
}
   }
}
```

27.5 Reglas de sobrescritura para atributos

Una definición de atributo en una unidad de ámbito más baja revocará una definición de atributo general en un ámbito más alto. Por ejemplo:

```
type record MyRecordA
{
   :
} with {encode "RuleA"}

// A continuación, MyRecordA se codifica de acuerdo con RuleA y no de acuerdo
// con RuleB
type record MyRecordB
{  :
   field MyRecordA
} with {encode "RuleB"}
```

Una definición de atributo en un de ámbito más bajo puede ser sobrescrita en un ámbito más alto utilizando la directiva **override**. Por ejemplo:

```
type record MyRecordA
{
   :
} with {encode "RuleA"}

// A continuación, MyRecordA se codifica de acuerdo con RuleB
type record MyRecordB
{  :
   fieldA   MyRecordA
} with {encode override "RuleB"}
```

La directiva **override** constriñe todos los tipos contenidos en ámbitos más bajos al atributo especificado.

27.6 Cambio de atributos de elementos de lenguaje importados

En general, un elemento de lenguaje es importado junto con sus atributos. En algunos casos, hay que cambiar estos atributos cuando se importa el elemento de lenguaje, por ejemplo, un tipo puede ser visualizado en un módulo como ASP, después es importado por otro módulo donde debe ser visualizado como PDU. En estos casos, se permite cambiar atributos en el enunciado de importación.

Ejemplo:

```
Import type MyType from MyModule with {display "ASP"} // MyType se visualizará
                                                      // como ASP.

Import group MyGroup from MyModule with
{
   display "ASP";            // Por defecto todos los tipos se visualizarán
                             // como ASP.
   extension "MyRule"
}
```

Anexo A

Forma de Backus-Nauer y semántica estática

A.1 Forma de Backus-Nauer para TTCN-3

Este anexo define la sintaxis de TTCN-3 que utiliza la forma Backus-Nauer ampliada (en adelante denominada BNF).

A.1.1 Convenios para la descripción de la sintaxis

El cuadro A.1 define la metanotación empleada para especificar la gramática BNF ampliada para TTCN-3:

Cuadro A.1/Z.140 – Metanotación sintáctica

::=	se define que es
abc xyz	abc seguido por xyz
|	alternativa
[abc]	0 ó 1 ejemplar de abc
{abc}	0 ó más ejemplares de abc
{abc}+	1 ó más ejemplares de abc
(...)	agrupación textual
Abc	el símbolo no terminal abc
abc	un símbolo terminal abc
"abc"	un símbolo terminal abc

A.1.2 Símbolos de terminador de enunciado

En general todas las construcciones de lenguaje TTCN-3 (es decir, definiciones, declaraciones, enunciados y operaciones) son terminadas con un punto y coma `(;)`. El punto y coma es facultativo si la construcción de lenguaje termina con un corchete ondulado a la derecha `(})` o el siguiente símbolo es un corchete ondulado a la derecha `(})`, es decir, la construcción de lenguaje es el último enunciado en un bloque de enunciados.

A.1.3 Identificadores

Los identificadores de TTCN-3 son sensibles a mayúsculas y minúsculas y sólo pueden contener letras minúsculas (`a-z`), letras mayúsculas (`A-Z`) y cifras (`0-9`). Se permite también utilizar el símbolo de subrayado (_). Un identificador comenzará con una letra (es decir, no con un número ni un subrayado).

A.1.4 Comentarios

Los comentarios escritos en texto libre pueden aparecer en cualquier parte de una especificación de TTCN-3.

Los comentarios de bloque serán abiertos por el par de símbolos /* y cerrados por el par de símbolos */. Por ejemplo:

```
/* Este es un comentario de bloque
que abarca dos líneas */
```

Los comentarios de bloque no serán anidados.

```
/* Este no es /* un comentario */ legal */
```

Los comentarios de línea serán abiertos por el par de símbolos // y cerrados por una <*newline*> (nueva línea). Por ejemplo:

```
// Éste es un comentario de línea
// que abarca dos líneas
```

Los comentarios de línea pueden seguir a los enunciados de programa TTCN-3 pero no estarán insertados en un enunciado. Por ejemplo:

```
// Lo siguiente no es legal
const // Éste es MyConst integer MyConst := 1;

// Lo siguiente es legal
const integer MyConst := 1; // Éste es MyConst
```

A.1.5 Terminales de TTCN-3

En los cuadros A.2 y A.3 se enumeran los símbolos terminales y palabras reservadas de TTCN-3.

Cuadro A.2/Z.140 – Lista de símbolos terminales especiales de TTCN-3

Símbolos de comienzo/fin de bloque	{ }
Símbolos de comienzo/fin de lista	( )
Símbolos alternativos	[ ]
A símbolo (en una gama)	..
Comentarios de línea y comentarios de bloque	/* */ //
Símbolo terminador de línea/enunciado	;
Símbolos de operador aritmético	+ / –
Símbolo de operador de concatenación de cadenas	&
Símbolos de operador de equivalencia	!= == >= <=
Símbolos de contenido dentro de cadena	" '
Símbolos de comodín/concordancia	? *
Símbolo de asignación	:=
Asignación de operación de comunicación	->
Valores de cadena de bits, cadena hexadecimal y cadena de octetos	B H O
Exponente flotante	E

A continuación se enumeran los identificadores especiales reservados para las funciones predefinidas que figuran en el anexo D.

```
int2char, char2int, int2unichar, unichar2int, bit2int, hex2int, int2bit,
int2hex, int2oct, int2str, oct2int, str2int, lengthof, sizeof, ischosen,
ispresent
```

Los terminales de TTCN-3 enumerados en el cuadro A.3 no serán utilizados como identificadores en un módulo TTCN-3. Todos estos terminales se escribirán con letras minúsculas.

Cuadro A.3/Z.140 – Lista de terminales de TTCN-3 que son palabras reservadas

action	fail	named	self
activate	false	none	send
address	float	nonrecursive	sender
all	for	not	set
alt	from	not4b	signature
and	function	nowait	start
and4b		null	stop
any	get		sut
	getcall	objid	system
bitstring	getreply	octetstring	
boolean	goto	of	template
	group	omit	testcase
call		on	timeout
catch	hexstring	optional	timer
char		or	to
charstring	if	or4b	trigger
check	ifpresent	out	true
clear	import	override	type
complement	in		
component	inconc	param	union
connect	infinity	pass	universal
const	inout	pattern	unmap
control	integer	port	
create	interleave	procedure	value
			valueof
deactivate	label	raise	var
disconnect	language	read	verdict
display	length	receive	verdicttype
do	log	record	
done		rem	while
	map	repeat	with
else	match	reply	
encode	message	return	xor
enumerated	mixed	running	xor4b
error	mod	runs	
exception	modifies		
execute	module		
expand	mtc		
extension			
external			

A.1.6 Producciones BNF para sintaxis TTCN-3

A.1.6.1 Módulo TTCN

```
1.  TTCN3Module ::= TTCN3ModuleKeyword TTCN3ModuleId [ModuleParList]
                BeginChar
                [ModuleDefinitionsPart]
                [ModuleControlPart]
                EndChar
                [WithStatement] [SemiColon]
2.  TTCN3ModuleKeyword ::= "module"
3.  TTCN3ModuleId ::= ModuleIdentifier [DefinitiveIdentifier]
4.  ModuleIdentifier ::= Identifier
5.  DefinitiveIdentifier ::= Dot ObjectIdentifierKeyword "{"
                DefinitiveObjIdComponentList "}"
6.  DefinitiveObjIdComponentList ::= {DefinitiveObjIdComponent}+
7.  DefinitiveObjIdComponent ::= NameForm |
                DefinitiveNumberForm |
                DefinitiveNameAndNumberForm
8.  DefinitiveNumberForm ::= Number
9.  DefinitiveNameAndNumberForm ::= Identifier  "(" DefinitiveNumberForm ")"
10. ModuleParList ::= "(" ModulePar {"," ModulePar} ")"
11. ModulePar ::= [InParKeyword] ModuleParType ModuleParIdentifier
                [AssignmentChar ConstantExpression]
/* SEMÁNTICA ESTÁTICA - El valor de ConstantExpression será del mismo tipo que
el indicado para el parámetro */
12. ModuleParType ::= Type
13. ModuleParIdentifier ::= Identifier
```

A.1.6.2 Parte de definiciones del módulo

```
14. ModuleDefinitionsPart ::= ModuleDefinitionsList
15. ModuleDefinitionsList ::= {ModuleDefinition [SemiColon]}+
16. ModuleDefinition ::= (TypeDef |
                ConstDef |
                TemplateDef |
                FunctionDef |
                SignatureDef |
                TestcaseDef |
                NamedAltDef |
                ImportDef |
                GroupDef |
                ExtFunctionDef |
                ExtConstDef) [WithStatement]
```

A.1.6.2.1 Definiciones de tipos

```
17. TypeDef ::= TypeDefKeyword TypeDefBody
18. TypeDefBody ::= StructuredTypeDef | SubTypeDef
19. TypeDefKeyword ::= "type"
20. StructuredTypeDef ::= RecordDef | UnionDef | SetDef | RecordOfDef |
                SetOfDef | EnumDef | PortDef | ComponentDef
21. RecordDef ::= RecordKeyword StructDefBody
22. RecordKeyword ::= "record"
23. StructDefBody ::= (StructTypeIdentifier  [StructDefFormalParList] |
                AddressKeyword)
                BeginChar
                [StructFieldDef {"," StructFieldDef}]
                EndChar
24. StructTypeIdentifier ::= Identifier
25. StructDefFormalParList ::= "(" StructDefFormalPar {"," StructDefFormalPar}
                ")"
26. StructDefFormalPar ::= FormalValuePar | FormalTypePar
/* SEMÁNTICA ESTÁTICA -  FormalValuePar se resolverá en un parámetro in */
```

```
27. StructFieldDef ::= Type StructFieldIdentifier [ArrayDef] [SubTypeSpec]
                [OptionalKeyword]
28. StructFieldIdentifier ::= Identifier
29. OptionalKeyword ::= "optional"
30. UnionDef ::= UnionKeyword UnionDefBody
31. UnionKeyword ::= "union"
32. UnionDefBody ::= (StructTypeIdentifier [StructDefFormalParList] |
               AddressKeyword)
               BeginChar
               UnionFieldDef {"," UnionFieldDef}
               EndChar
33. UnionFieldDef ::= Type StructFieldIdentifier [ArrayDef] [SubTypeSpec]
34. SetDef ::= SetKeyword StructDefBody
35. SetKeyword ::= "set"
36. RecordOfDef ::= RecordKeyword OfKeyword [StringLength]  StructOfDefBody
37. OfKeyword ::= "of"
38. StructOfDefBody ::= Type (StructTypeIdentifier | AddressKeyword)
                [SubTypeSpec]
39. SetOfDef ::= SetKeyword OfKeyword [StringLength]  StructOfDefBody
40. EnumDef ::= EnumKeyword (EnumTypeIdentifier | AddressKeyword)
               BeginChar
               NamedValueList
               EndChar
41. EnumKeyword ::= "enumerated"
42. EnumTypeIdentifier ::= Identifier
43. NamedValueList ::= NamedValue {"," NamedValue}
44. NamedValue ::= NamedValueIdentifier ["(" Number ")"]
45. NamedValueIdentifier ::= Identifier
46. SubTypeDef ::= Type (SubTypeIdentifier | AddressKeyword)  [ArrayDef]
                [SubTypeSpec]
47. SubTypeIdentifier ::= Identifier
48. SubTypeSpec ::= AllowedValues | StringLength
/* SEMÁNTICA ESTÁTICA - Los valores serán del mismo tipo que el campo
subptificado */
49. AllowedValues ::= "(" ValueOrRange {"," ValueOrRange} ")"
50. ValueOrRange ::= IntegerRangeDef | SingleConstExpression
/* SEMÁNTICA ESTÁTICA - IntegerRangeDef  se usará sólo con tipos basados en
enteros */
51. IntegerRangeDef ::= LowerBound ".." UpperBound
52. StringLength ::= LengthKeyword "(" SingleConstExpression [".." UpperBound]
               ")"
/* SEMÁNTICA ESTÁTICA - StringLength se usará sólo con tipos  String o para
limitar set of y record of */
53. LengthKeyword ::= "length"
54. PortType ::= [GlobalModuleId Dot] PortTypeIdentifier
55. PortDef ::= PortKeyword PortDefBody
56. PortDefBody ::= PortTypeIdentifier PortDefAttribs
57. PortKeyword ::= "port"
58. PortTypeIdentifier ::= Identifier
59. PortDefAttribs ::= MessageAttribs | ProcedureAttribs | MixedAttribs
60. MessageAttribs ::= MessageKeyword
               BeginChar
               {MessageList  [SemiColon]}+
               EndChar
61. MessageList ::= Direction AllOrTypeList
62. Direction ::= InParKeyword | OutParKeyword | InOutParKeyword
63. MessageKeyword ::= "message"
64. AllOrTypeList ::= AllKeyword | TypeList
65. AllKeyword ::= "all"
66. TypeList ::= Type {"," Type}
67. ProcedureAttribs ::= ProcedureKeyword
               BeginChar
               {ProcedureList  [SemiColon]}+
               EndChar
```

```
68. ProcedureKeyword ::= "procedure"
69. ProcedureList ::= Direction AllOrSignatureList
70. AllOrSignatureList ::= AllKeyword | SignatureList
71. SignatureList ::= Signature {"," Signature}
72. MixedAttribs ::= MixedKeyword
                BeginChar
                {MixedList  [SemiColon]}+
                EndChar
73. MixedKeyword ::= "mixed"
74. MixedList ::= Direction ProcOrTypeList
75. ProcOrTypeList ::= AllKeyword | (ProcOrType {"," ProcOrType})
76. ProcOrType ::= Signature | Type
77. ComponentDef ::= ComponentKeyword ComponentTypeIdentifier
                BeginChar
                [ComponentDefList]
                EndChar
78. ComponentKeyword ::= "component"
79. ComponentType ::= [GlobalModuleId Dot] ComponentTypeIdentifier
80. ComponentTypeIdentifier ::= Identifier
81. ComponentDefList ::= {ComponentElementDef [SemiColon]}+
82. ComponentElementDef ::= PortInstance | VarInstance | TimerInstance |
                    ConstDef
83. PortInstance ::= PortKeyword PortType PortElement {"," PortElement}
84. PortElement ::= PortIdentifier [ArrayDef]
85. PortIdentifier ::= Identifier
```

A.1.6.2.2 Definiciones de constantes

```
86. ConstDef ::= ConstKeyword Type ConstList
87. ConstList ::= SingleConstDef  {"," SingleConstDef}
88. SingleConstDef ::= ConstIdentifier [ArrayDef] AssignmentChar
                ConstantExpression
/* SEMÁNTICA ESTÁTICA - El valor de ConstantExpression será del mismo tipo que
el indicado para la constante */
89. ConstKeyword ::= "const"
90. ConstIdentifier ::= Identifier
```

A.1.6.2.3 Definiciones de plantillas

```
91. TemplateDef ::= TemplateKeyword BaseTemplate  [DerivedDef]
                AssignmentChar TemplateBody
92. BaseTemplate ::= (Type | Signature) TemplateIdentifier  ["("
                TemplateFormalParList ")"]
93. TemplateKeyword ::= "template"
94. TemplateIdentifier ::= Identifier
95. DerivedDef ::= ModifiesKeyword TemplateRef
96. ModifiesKeyword ::= "modifies"
97. TemplateFormalParList ::= TemplateFormalPar {"," TemplateFormalPar}
98. TemplateFormalPar ::= FormalValuePar |
                FormalTemplatePar
/* SEMÁNTICA ESTÁTICA - FormalValuePar se resolverá en un parámetro in */
99. TemplateBody ::= SimpleSpec | FieldSpecList |
                ArrayValueOrAttrib
100. SimpleSpec ::= SingleValueOrAttrib
/* SEMÁNTICA ESTÁTICA - SimpleSpec no se utilizará para tipos construidos */
101. FieldSpecList ::= "{"[FieldSpec {"," FieldSpec}] "}"
102. FieldSpec ::= FieldReference AssignmentChar TemplateBody
103. FieldReference ::= RecordRef | ArrayOrBitRef | ParRef
104. RecordRef ::= StructFieldIdentifier
105. ParRef ::= SignatureParIdentifier
/* SEMÁNTICA OPERACIONAL - SignatureParIdentifier será un identificador de
parámetro de la definición de firma asociada */
106. SignatureParIdentifier ::= ValueParIdentifier
107. ArrayOrBitRef ::= "[" FieldOrBitNumber "]"
```

```
/* SEMÁNTICA ESTÁTICA - ArrayRef se utilizará facultativamente para tipos de
matrices, SET OF y SEQUENCE OF de ASN.1 y record, record of, set y set
Identifier of de TTCN. La misma notación se puede usar para una referencia de
bits dentro de un tipo bitstring  ASN.1 o TTCN*/
108. FieldOrBitNumber ::= SingleExpression
/* SEMÁNTICA ESTÁTICA - SingleExpression se resolverá en un valor de tipo entero
*/
109. SingleValueOrAttrib ::= MatchingSymbol [ExtraMatchingAttributes] |
               SingleExpression [ExtraMatchingAttributes] |
               TemplateRefWithParList
/* SEMÁNTICA ESTÁTICA - VariableIdentifier (accedido por singleExpression) sólo
se puede usar en definiciones de plantillas en línea de variables de referencia
en el ámbito vigente */
110. ArrayValueOrAttrib ::= "{" ArrayElementSpecList "}"
111. ArrayElementSpecList ::= ArrayElementSpec {"," ArrayElementSpec}
112. ArrayElementSpec ::= NotUsedSymbol | TemplateBody
113. NotUsedSymbol ::= Dash
114. MatchingSymbol ::= Complement | Omit | AnyValue | AnyOrOmit | ValueList |
               IntegerRange | BitStringMatch | HexStringMatch |
               OctetStringMatch | CharStringMatch
115. ExtraMatchingAttributes ::= LengthMatch | IfPresentMatch
116. BitStringMatch ::= "'" {BinOrMatch} "'" B
117. BinOrMatch ::= Bin | AnyValue | AnyOrOmit
118. HexStringMatch ::= "'" {HexOrMatch} "'" H
119. HexOrMatch ::= Hex | AnyValue | AnyOrOmit
120. OctetStringMatch ::= "'" {OctOrMatch} "'" O
121. OctOrMatch ::= Oct | AnyValue | AnyOrOmit
122. CharStringMatch ::= PatternKeyword CharStringPattern {StringOp
               CharStringPattern}
/* SEMÁNTICA ESTÁTICA - CharStringPatterns se resolverán en el mismo tipo de
carácter o de cadena de caracteres */
123. CharStringPattern ::= CharStringValue | TemplateRefWithParList
124. PatternKeyword ::= "pattern"
125. Complement ::= ComplementKeyword (SingleConstExpression | ValueList)
126. ComplementKeyword ::= "complement"
127. Omit ::= OmitKeyword
128. OmitKeyword ::= "omit"
129. AnyValue ::= "?"
130. AnyOrOmit ::= "*"
131. ValueList ::= "(" SingleConstExpression {"," SingleConstExpression}+ ")"
132. LengthMatch ::= StringLength
133. IfPresentMatch ::= IfPresentKeyword
134. IfPresentKeyword ::= "ifpresent"
135. IntegerRange ::= "(" LowerBound ".." UpperBound ")"
136. LowerBound ::= SingleConstExpression | Minus InfinityKeyword
137. UpperBound ::= SingleConstExpression | InfinityKeyword
138. InfinityKeyword ::= "infinity"
139. TemplateInstance ::= InLineTemplate
140. TemplateRefWithParList ::= [GlobalModuleId Dot] TemplateIdentifier
               [TemplateActualParList] | TemplateParIdentifier
141. TemplateRef ::= [GlobalModuleId Dot] TemplateIdentifier |
               TemplateParIdentifier
142. InLineTemplate ::= [(Type | Signature) Colon] [DerivedDef AssignmentChar]
               TemplateBody
/* SEMÁNTICA ESTÁTICA - El campo de tipo sólo se puede omitir cuando el tipo es
implícitamente inequívoco */
143. TemplateActualParList ::= "(" TemplateActualPar {"," TemplateActualPar} ")"
144. TemplateActualPar ::= TemplateInstance
/* SEMÁNTICA ESTÁTICA - Cuando el correspondiente parámetro formal no es el tipo
de plantilla, TemplateInstance se resolverá en una o más SingleExpressions */
145. TemplateOps ::= MatchOp | ValueofOp
146. MatchOp ::= MatchKeyword "(" Expression "," TemplateInstance")"
```

```
/* SEMÁNTICA ESTÁTICA - El tipo del valor devuelto por la expresión será igual
que el tipo de plantilla y cada campo de la plantilla se resolverá en un solo
valor */
147. MatchKeyword ::= "match"
148. ValueofOp ::= ValueofKeyword "(" TemplateInstance")"
149. ValueofKeyword ::= "valueof"
```

A.1.6.2.4 Definiciones de funciones

```
150. FunctionDef ::= FunctionKeyword FunctionIdentifier
                "("[FunctionFormalParList] ")" [RunsOnSpec] [ReturnType]
                BeginChar
                FunctionBody
                EndChar
151. FunctionKeyword ::= "function"
152. FunctionIdentifier ::= Identifier
153. FunctionFormalParList ::= FunctionFormalPar {"," FunctionFormalPar}
154. FunctionFormalPar ::= FormalValuePar |
                FormalTimerPar |
                FormalTemplatePar |
                FormalPortPar
155. ReturnType ::= ReturnKeyword Type
156. ReturnKeyword ::= "return"
157. RunsOnSpec ::= RunsKeyword OnKeyword (ComponentType | MTCKeyword)
158. RunsKeyword ::= "runs"
159. OnKeyword ::= "on"
160. MTCKeyword ::= "mtc"
161. FunctionBody ::= [FunctionStatementOrDefList]
162. FunctionStatementOrDefList ::= {FunctionStatementOrDef [SemiColon]}+
163. FunctionStatementOrDef ::= FunctionLocalDef |
                FunctionLocalInst |
                FunctionStatement
164. FunctionLocalInst ::= VarInstance |
                TimerInstance
165. FunctionLocalDef ::= ConstDef
166. FunctionStatement ::= ConfigurationStatements |
                TimerStatements |
                CommunicationStatements |
                BasicStatements |
                BehaviourStatements |
                VerdictStatements |
                SUTStatements
167. FunctionInstance ::= FunctionRef "(" [FunctionActualParList] ")"
168. FunctionRef ::= [GlobalModuleId Dot] FunctionIdentifier
169. FunctionActualParList ::= FunctionActualPar {"," FunctionActualPar}
170. FunctionActualPar ::= TimerRef |
                TemplateInstance |
                Port |
                ComponentRef
/* SEMÁNTICA ESTÁTICA - Cuando el correspondiente parámetro formal no es del
tipo de plantilla, TemplateInstance se resolverá en una o más SingleExpressions,
es decir, equivalente a la producción Expression */
```

A.1.6.2.5 Definiciones de firmas

```
171. SignatureDef ::= SignatureKeyword SignatureIdentifier
                "("[SignatureFormalParList] ")" [ReturnType]
                [ExceptionSpec]
172. SignatureKeyword ::= "signature"
173. SignatureIdentifier ::= Identifier
174. SignatureFormalParList ::= SignatureFormalPar {"," SignatureFormalPar}
175. SignatureFormalPar ::= FormalValuePar
176. ExceptionSpec ::= ExceptionKeyword "(" ExceptionTypeList ")"
```

```
177. ExceptionKeyword ::= "exception"
178. ExceptionTypeList ::= Type {"," Type}
179. Signature ::= [GlobalModuleId Dot] SignatureIdentifier
```

A.1.6.2.6 Definiciones de casos de prueba

```
180. TestcaseDef ::= TestcaseKeyword TestcaseIdentifier
                "("[TestcaseFormalParList] ")" ConfigSpec
                BeginChar
                FunctionBody
                EndChar
181. TestcaseKeyword ::= "testcase"
182. TestcaseIdentifier ::= Identifier
183. TestcaseFormalParList ::= TestcaseFormalPar {"," TestcaseFormalPar}
184. TestcaseFormalPar ::= FormalValuePar |
                FormalTemplatePar
185. ConfigSpec ::= RunsOnSpec [SystemSpec]
186. SystemSpec ::= SystemKeyword ComponentType
187. SystemKeyword ::= "system"
188. TestcaseInstance ::= ExecuteKeyword "(" TestcaseRef "("
                 [TestcaseActualParList] ")" ["," TimerValue]   ")"
189. ExecuteKeyword ::= "execute"
190. TestcaseRef ::= [GlobalModuleId Dot] TestcaseIdentifier
191. TestcaseActualParList ::= TestcaseActualPar {"," TestcaseActualPar}
192. TestcaseActualPar ::=
                TemplateInstance
/* SEMÁNTICA ESTÁTICA - Cuando el correspondiente parámetro formal no es de tipo
plantilla,  TemplateInstance se resolverá en una o más SingleExpressions, es
decir, equivalente a la producción Expression */
```

A.1.6.2.7 Definiciones de alternativas denominadas

```
193. NamedAltDef ::= NamedKeyword AltKeyword NamedAltIdentifier
                "("  [NamedAltFormalParList] ")"
                BeginChar
                AltGuardList                    EndChar
194. NamedKeyword ::= "named"
195. NamedAltIdentifier ::= Identifier
196. NamedAltFormalParList ::= NamedAltFormalPar {"," NamedAltFormalPar}
197. NamedAltFormalPar ::= FormalValuePar |
                FormalTimerPar |
                FormalTemplatePar |
                FormalPortPar
198. NamedAltInstance ::= NamedAltRef  "(" [NamedAltActualParList]")"
199. NamedAltRef ::= [GlobalModuleId Dot] NamedAltIdentifier
200. NamedAltActualParList ::= NamedAltActualPar {"," NamedAltActualPar}
201. NamedAltActualPar ::=
                TimerRef |
                TemplateInstance |
                Port |
                ComponentRef
/* SEMÁNTICA ESTÁTICA – Cuando el correspondiente parámetro formal no es de tipo
plantilla, TemplateInstance se resolverá en una o más SingleExpressions, es
decir, equivalente a la producción Expression */
```

A.1.6.2.8 Definiciones de importaciones

```
202. ImportDef ::= ImportKeyword ImportSpec
203. ImportKeyword ::= "import"
204. ImportSpec ::= ImportAllSpec |
                ImportGroupSpec |
                ImportTypeDefSpec |
                ImportTemplateSpec |
                ImportConstSpec |
```

```
                ImportTestcaseSpec |
                ImportNamedAltSpec |
                ImportFunctionSpec |
                ImportSignatureSpec
205. ImportAllSpec ::= AllKeyword [DefKeyword] ImportFromSpec
206. ImportFromSpec ::= FromKeyword ModuleId [NonRecursiveKeyword]
207. ModuleId ::= GlobalModuleId [LanguageSpec]
/* SEMÁNTICA ESTÁTICA – LanguageSpec sólo se puede omitir si el módulo
referenciado contiene notación TTCN-3 */
208. LanguageKeyword ::= "language"
209. LanguageSpec ::= LanguageKeyword  FreeText
210. GlobalModuleId ::= ModuleIdentifier [Dot ObjectIdentifierValue]
211. DefKeyword ::= TypeDefKeyword |
                ConstKeyword |
                TemplateKeyword |
                TestcaseKeyword |
                FunctionKeyword |
                SignatureKeyword |
                NamedKeyword AltKeyword
212. NonRecursiveKeyword ::= "nonrecursive"
213. ImportGroupSpec ::= GroupKeyword GroupIdentifier {"," GroupIdentifier}
                ImportFromSpec
214. ImportTypeDefSpec ::= TypeDefKeyword TypeDefIdentifier {","
TypeDefIdentifier} ImportFromSpec
215. TypeDefIdentifier ::= StructTypeIdentifier |
                EnumTypeIdentifier |
                PortTypeIdentifier |
                ComponentTypeIdentifier |
                SubTypeIdentifier
216. ImportTemplateSpec ::= TemplateKeyword TemplateIdentifier {","
                TemplateIdentifier} ImportFromSpec
217. ImportConstSpec ::= ConstKeyword ConstIdentifier {"," ConstIdentifier}
                ImportFromSpec
218. ImportTestcaseSpec ::= TestcaseKeyword TestcaseIdentifier {","
                TestcaseIdentifier} ImportFromSpec
219. ImportFunctionSpec ::= FunctionKeyword FunctionIdentifier {","
                FunctionIdentifier} ImportFromSpec
220. ImportSignatureSpec ::= SignatureKeyword SignatureIdentifier {","
                SignatureIdentifier} ImportFromSpec
221. ImportNamedAltSpec ::= NamedKeyword AltKeyword NamedAltIdentifier {","
                NamedAltIdentifier} ImportFromSpec
```

A.1.6.2.9 Definiciones de grupos

```
222. GroupDef ::= GroupKeyword GroupIdentifier
                BeginChar
                [ModuleDefinitionsPart]
                EndGroupChar
223. GroupKeyword ::= "group"
224. EndGroupChar ::= "}"
225. GroupIdentifier ::= Identifier
```

A.1.6.2.10 Definiciones de funciones externas

```
226. ExtFunctionDef ::= ExtKeyword FunctionKeyword ExtFunctionIdentifier
                "("[FunctionFormalParList] ")" [ReturnType]
227. ExtKeyword ::= "external"
228. ExtFunctionIdentifier ::= Identifier
```

A.1.6.2.11 Definiciones de constantes externas

```
229. ExtConstDef ::= ExtKeyword ConstKeyword Type ExtConstIdentifier
230. ExtConstIdentifier ::= Identifier
```

A.1.6.3 Parte de control

```
231. ModuleControlPart ::= ControlKeyword
               BeginChar
               ModuleControlBody
               EndChar
               [WithStatement] [SemiColon]
232. ControlKeyword ::= "control"
233. ModuleControlBody ::= [ControlStatementOrDefList]
234. ControlStatementOrDefList ::= {ControlStatementOrDef [SemiColon]}+
235. ControlStatementOrDef ::= FunctionLocalInst |
               ControlStatement |
               FunctionLocalDef
236. ControlStatement ::= TimerStatements |
               BasicStatements |
               BehaviourStatements |
               SUTStatements
```

A.1.6.3.1 Ejemplificación de variables

```
237. VarInstance ::= VarKeyword Type VarList
238. VarList ::= SingleVarInstance {"," SingleVarInstance}
239. SingleVarInstance ::= VarIdentifier [ArrayDef] [AssignmentChar
               VarInitialValue]
240. VarInitialValue ::= Expression
241. VarKeyword ::= "var"
242. VarIdentifier ::= Identifier
243. VariableRef ::= (VarIdentifier | ValueParIdentifier)
               [ExtendedFieldReference]
```

A.1.6.3.2 Ejemplificación de temporizadores

```
244. TimerInstance ::= TimerKeyword TimerIdentifier [ArrayDef]
               [AssignmentChar TimerValue]
245. TimerKeyword ::= "timer"
246. TimerIdentifier ::= Identifier
247. TimerValue ::= SingleExpression
/* SEMÁNTICA ESTÁTICA – SingleExpression se resolverá en un valor de tipo float
*/
248. TimerRef ::= TimerIdentifier [ArrayOrBitRef]|
               TimerParIdentifier [ArrayOrBitRef]
```

A.1.6.3.3 Operaciones de componentes

```
249. ConfigurationStatements ::= ConnectStatement |
               MapStatement |
               DisconnectStatement |
               UnmapStatement |
               DoneStatement |
               StartTCStatement |
               StopTCStatement
250. ConfigurationOps ::= CreateOp | SelfOp | SystemOp | MTCOp | RunningOp
251. CreateOp ::= ComponentType Dot CreateKeyword
252. SystemOp ::= "system"
253. SelfOp ::= "self"
254. MTCOp ::= MTCKeyword
255. DoneStatement ::= ComponentId Dot DoneKeyword
256. ComponentId ::= ComponentIdentifier | (AnyKeyword | AllKeyword)
               ComponentKeyword
257. DoneKeyword ::= "done"
258. RunningOp ::= ComponentId Dot RunningKeyword
259. RunningKeyword ::= "running"
260. CreateKeyword ::= "create"
```

```
261. ConnectStatement ::= ConnectKeyword PortSpec
262. ConnectKeyword ::= "connect"
263. PortSpec ::= "(" PortRef "," PortRef ")"
264. PortRef ::= ComponentRef Colon Port
265. ComponentRef ::= ComponentIdentifier | SystemOp | SelfOp | MTCOp
266. DisconnectStatement ::= DisconnectKeyword PortSpec
267. DisconnectKeyword ::= "disconnect"
268. MapStatement ::= MapKeyword PortSpec
269. MapKeyword ::= "map"
270. UnmapStatement ::= UnmapKeyword PortSpec
271. UnmapKeyword ::= "unmap"
272. StartTCStatement ::= ComponentIdentifier Dot StartKeyword "("
               FunctionInstance ")"
/* SEMÁNTICA ESTÁTICA - El ejemplar Function sólo puede tener parámetros in */
273. StartKeyword ::= "start"
274. StopTCStatement ::= StopKeyword
275. ComponentIdentifier ::= VariableRef | FunctionInstance
/* SEMÁNTICA ESTÁTICA - La variable asociada con VariableRef o el tipo Return
asociado con FunctionInstance será el tipo componente */
```

A.1.6.3.4 Operaciones de puertos

```
276. Port ::= (PortIdentifier | PortParIdentifier) [ArrayOrBitRef]
277. CommunicationStatements ::= SendStatement | CallStatement | ReplyStatement
| RaiseStatement |
               ReceiveStatement | TriggerStatement | GetCallStatement |
               GetReplyStatement | CatchStatement | CheckStatement |
               ClearStatement | StartStatement | StopStatement
278. SendStatement ::= Port Dot PortSendOp
279. PortSendOp ::= SendOpKeyword "(" SendParameter ")" [ToClause]
280. SendOpKeyword ::= "send"
281. SendParameter ::= TemplateInstance
282. ToClause ::= ToKeyword AddressRef
283. ToKeyword ::= "to"
284. AddressRef ::= VariableRef | FunctionInstance
/* SEMÁNTICA ESTÁTICA - VariableRef y FunctionInstance sólo devolverán el tipo
dirección o componente */
285. CallStatement ::= Port Dot PortCallOp  [PortCallBody]
286. PortCallOp ::= CallOpKeyword "(" CallParameters ")" [ToClause]
287. CallOpKeyword ::= "call"
288. CallParameters ::= TemplateInstance ["," CallTimerValue]
/* SEMÁNTICA ESTÁTICA - Sólo se puede omitir o especificar parámetros out con un
atributo concordante */
289. CallTimerValue ::= TimerValue | NowaitKeyword
/* SEMÁNTICA ESTÁTICA - El valor será del tipo float */
290. NowaitKeyword ::= "nowait"
291. PortCallBody ::= BeginChar
               CallBodyStatementList
               EndChar
292. CallBodyStatementList ::= {CallBodyStatement [SemiColon]}+
293. CallBodyStatement ::= CallBodyGuard  StatementBlock
294. CallBodyGuard ::= AltGuardChar CallBodyOps
295. CallBodyOps ::= GetReplyStatement | CatchStatement
296. ReplyStatement ::= Port Dot PortReplyOp
297. PortReplyOp ::= ReplyKeyword "(" TemplateInstance [ReplyValue]")"
               [ToClause]
298. ReplyKeyword ::= "reply"
299. ReplyValue ::= ValueKeyword Expression
300. RaiseStatement ::= Port Dot PortRaiseOp
301. PortRaiseOp ::= RaiseKeyword "(" Signature "," TemplateInstance ")"
               [ToClause]
302. RaiseKeyword ::= "raise"
303. ReceiveStatement ::= PortOrAny Dot PortReceiveOp
304. PortOrAny ::= Port | AnyKeyword PortKeyword
```

```
305. PortReceiveOp ::= ReceiveOpKeyword ["(" ReceiveParameter ")"] [FromClause]
               [PortRedirect]
/* SEMÁNTICA ESTÁTICA - La opción PortRedirect sólo puede estar presente si
ReceiveParameter está también  presente */
306. ReceiveOpKeyword ::= "receive"
307. ReceiveParameter ::= TemplateInstance
308. FromClause ::= FromKeyword AddressRef
309. FromKeyword ::= "from"
310. PortRedirect ::= PortRedirectSymbol (ValueSpec [SenderSpec] | SenderSpec)
311. PortRedirectSymbol ::= "->"
312. ValueSpec ::= ValueKeyword VariableRef
313. ValueKeyword ::= "value"
314. SenderSpec ::= SenderKeyword VariableRef
/* SEMÁNTICA ESTÁTICA - La variable ref será del tipo dirección o componente */
315. SenderKeyword ::= "sender"
316. TriggerStatement ::= PortOrAny Dot PortTriggerOp
317. PortTriggerOp ::= TriggerOpKeyword ["(" ReceiveParameter ")"] [FromClause]
               [PortRedirect]
/* SEMÁNTICA ESTÁTICA - La opción PortRedirect sólo puede estar presente si
ReceiveParameter está también presente */
318. TriggerOpKeyword ::= "trigger"
319. GetCallStatement ::= PortOrAny  Dot PortGetCallOp
320. PortGetCallOp ::= GetCallOpKeyword ["(" ReceiveParameter ")"] [FromClause]
               [PortRedirectWithParam]
/* SEMÁNTICA ESTÁTICA - La opción PortRedirectWithParam sólo puede estar
presente si ReceiveParameter está también presente */
321. GetCallOpKeyword ::= "getcall"
322. PortRedirectWithParam ::= PortRedirectSymbol RedirectSpec
323. RedirectSpec ::= ValueSpec  [ParaSpec] [SenderSpec] |
               ParaSpec [SenderSpec] |
               SenderSpec
324. ParaSpec ::= ParaKeyword ParaAssignmentList
325. ParaKeyword ::= "param"
326. ParaAssignmentList ::= "(" (AssignmentList | VariableList) ")"
327. AssignmentList ::= VariableAssignment {"," VariableAssignment}
328. VariableAssignment ::= VariableRef AssignmentChar ParameterIdentifier
/* SEMÁNTICA ESTÁTICA - Los parameterIdentifiers serán de la correspondiente
definición de firma */
329. ParameterIdentifier ::= ValueParIdentifier |
               TimerParIdentifier |
               TemplateParIdentifier |
               PortParIdentifier
330. VariableList ::= VariableEntry  {"," VariableEntry}
331. VariableEntry ::= VariableRef | NotUsedSymbol
332. GetReplyStatement ::= PortOrAny Dot PortGetReplyOp
333. PortGetReplyOp ::= GetReplyOpKeyword ["(" ReceiveParameter [ValueMatchSpec]
               ")"] [FromClause] [PortRedirectWithParam]
/* SEMÁNTICA ESTÁTICA - La opción PortRedirectWithParam sólo puede estar
presente si ReceiveParameter está también presente */
334. GetReplyOpKeyword ::= "getreply"
335. ValueMatchSpec ::= ValueKeyword TemplateInstance
336. CheckStatement ::= PortOrAny Dot PortCheckOp
337. PortCheckOp ::= CheckOpKeyword ["(" CheckParameter ")"]
338. CheckOpKeyword ::= "check"
339. CheckParameter ::= PortReceiveOp | PortGetCallOp | PortGetReplyOp |
               PortCatchOp | [FromClause] [PortRedirectSymbol SenderSpec]
340. CatchStatement ::= PortOrAny Dot PortCatchOp
341. PortCatchOp ::= CatchOpKeyword ["("CatchOpParameter ")"] [FromClause]
               [PortRedirect]
/* SEMÁNTICA ESTÁTICA - La opción PortRedirect sólo puede estar presente si
CatchOpParameter está también presente */
342. CatchOpKeyword ::= "catch"
343. CatchOpParameter ::= Signature "," TemplateInstance | TimeoutKeyword
344. ClearStatement ::= PortOrAll Dot PortClearOp
```

```
345. PortOrAll ::= Port | AllKeyword PortKeyword
346. PortClearOp ::= ClearOpKeyword
347. ClearOpKeyword ::= "clear"
348. StartStatement ::= PortOrAll Dot PortStartOp
349. PortStartOp ::= StartKeyword
350. StopStatement ::= PortOrAll Dot PortStopOp
351. PortStopOp ::= StopKeyword
352. StopKeyword ::= "stop"
353. AnyKeyword ::= "any"
```

A.1.6.3.5 Operaciones de temporizadores

```
354. TimerStatements ::= StartTimerStatement | StopTimerStatement |
                TimeoutStatement
355. TimerOps ::= ReadTimerOp | RunningTimerOp
356. StartTimerStatement ::= TimerRef Dot StartKeyword ["(" TimerValue ")"]
357. StopTimerStatement ::= TimerRefOrAll Dot StopKeyword
358. TimerRefOrAll ::= TimerRef | AllKeyword TimerKeyword
359. ReadTimerOp ::= TimerRef Dot ReadKeyword
360. ReadKeyword ::= "read"
361. RunningTimerOp ::= TimerRefOrAny  Dot RunningKeyword
362. TimeoutStatement ::= TimerRefOrAny  Dot TimeoutKeyword
363. TimerRefOrAny ::= TimerRef | AnyKeyword TimerKeyword
364. TimeoutKeyword ::= "timeout"
```

A.1.6.4 Tipos

```
365. Type ::= PredefinedType | ReferencedType
366. PredefinedType ::= BitStringKeyword |
                BooleanKeyword |
                CharStringKeyword |
                UniversalCharString |
                CharKeyword |
                UniversalChar |
                IntegerKeyword |
                OctetStringKeyword |
                ObjectIdentifierKeyword |
                HexStringKeyword |
                VerdictKeyword |
                FloatKeyword |
                AddressKeyword
367. BitStringKeyword ::= "bitstring"
368. BooleanKeyword ::= "boolean"
369. IntegerKeyword ::= "integer"
370. OctetStringKeyword ::= "octetstring"
371. ObjectIdentifierKeyword ::= "objid"
372. HexStringKeyword ::= "hexstring"
373. VerdictKeyword ::= "verdict"
374. FloatKeyword ::= "float"
375. AddressKeyword ::= "address"
376. CharStringKeyword ::= "charstring"
377. UniversalCharString ::= UniversalKeyword CharStringKeyword
378. UniversalKeyword ::= "universal"
379. CharKeyword ::= "char"
380. UniversalChar ::= UniversalKeyword CharKeyword
381. ReferencedType ::= [GlobalModuleId Dot] TypeReference
                [ExtendedFieldReference]
382. TypeReference ::= StructTypeIdentifier[TypeActualParList] |
                EnumTypeIdentifier |
                SubTypeIdentifier |
                TypeParIdentifier |
                ComponentTypeIdentifier
383. TypeActualParList ::= "(" TypeActualPar {"," TypeActualPar} ")"
384. TypeActualPar ::= SingleConstExpression | Type
```

A.1.6.4.1 Tipos de matriz

```
385. ArrayDef ::= {"[" ArrayBounds [".." ArrayBounds] "]"}+
386. ArrayBounds ::= SingleConstExpression
/* SEMÁNTICA ESTÁTICA – ArrayBounds se resolverá en un valor no negativo de tipo
entero */
```

A.1.6.5 Valores

```
387. Value ::= PredefinedValue | ReferencedValue
388. PredefinedValue ::= BitStringValue |
                BooleanValue |
                CharStringValue |
                IntegerValue |
                OctetStringValue |
                ObjectIdentifierValue |
                HexStringValue |
                VerdictValue |
                EnumeratedValue |
                FloatValue |
                AddressValue
389. BitStringValue ::= Bstring
390. BooleanValue ::= "true" | false
391. IntegerValue ::= Number
392. OctetStringValue ::= Ostring
393. ObjectIdentifierValue ::= ObjectIdentifierKeyword  "{" ObjIdComponentList
                "}"
/* SEMÁNTICA ESTÁTICA – ReferencedValue será del tipo objectIdentifer */
394. ObjIdComponentList ::= {ObjIdComponent}+
395. ObjIdComponent ::= NameForm |
                NumberForm |
                NameAndNumberForm
396. NumberForm ::= Number | ReferencedValue
/* SEMÁNTICA ESTÁTICA – referencedValue será del tipo entero y tendrá un valor
no negativo */
397. NameAndNumberForm ::= Identifier  NumberForm
398. NameForm ::= Identifier
399. HexStringValue ::= Hstring
400. VerdictValue ::= "pass" | fail | inconc | none | error
401. EnumeratedValue ::= NamedValueIdentifier
402. CharStringValue ::= Cstring | Quadruple | ReferencedValue
/* SEMÁNTICA ESTÁTICA – ReferencedValue se resolverá en un tipo cadena */
403. Quadruple ::= "(" Group "," Plane "," Row "," Cell ")"
404. Group ::= Number
405. Plane ::= Number
406. Row ::= Number
407. Cell ::= Number
408. FloatValue ::= FloatDotNotation | FloatENotation
409. FloatDotNotation ::= Number Dot DecimalNumber
410. FloatENotation ::= Number [Dot DecimalNumber] Exponential [Minus] Number
411. Exponential ::= E
412. ReferencedValue ::= ValueReference [ExtendedFieldReference]
413. ValueReference ::= [GlobalModuleId Dot] ConstIdentifier |
                ExtConstIdentifier |
                ValueParIdentifier |
                ModuleParIdentifier |
                VarIdentifier
414. Number ::= (NonZeroNum {Num}) | 0
415. NonZeroNum ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
416. DecimalNumber ::= {Num}
417. Num ::= 0 | NonZeroNum
418. Bstring ::= "'" {Bin} "'" B
419. Bin ::= 0 | 1
420. Hstring ::= "'" {Hex} "'" H
```

```
421. Hex ::= Num | A | B | C | D | E | F| a | b | c | d | e | f
422. Ostring ::= "'" {Oct} "'" O
423. Oct ::= Hex Hex
424. Cstring ::= """ {Char} """
425. Char ::= /* REFERENCIA – Un carácter definido por el tipo CharacterString
              pertinente */
426. Identifier ::= Alpha{AlphaNum | Underscore}
427. Alpha ::= UpperAlpha | LowerAlpha
428. AlphaNum ::= Alpha | Num
429. UpperAlpha ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
              P | Q | R | S | T | U | V | W | X | Y | Z
430. LowerAlpha ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
              p | q | r | s | t | u | v | w | x | y | z
431. ExtendedAlphaNum ::= /* REFERENCIA – Un carácter de cualquier juego de
              caracteres definido en ISO/CEI 10646-1/
432. FreeText ::= """ {ExtendedAlphaNum} """
433. AddressValue ::= "null"
```

A.1.6.6 Parametrización

```
434. InParKeyword ::= "in"
435. OutParKeyword ::= "out"
436. InOutParKeyword ::= "inout"
437. FormalValuePar ::= [(InParKeyword | InOutParKeyword | OutParKeyword)] Type
              ValueParIdentifier
438. ValueParIdentifier ::= Identifier
439. FormalTypePar ::= [InParKeyword] TypeParIdentifier
440. TypeParIdentifier ::= Identifier
441. FormalPortPar ::= [InOutParKeyword] PortTypeIdentifier PortParIdentifier
442. PortParIdentifier ::= Identifier
443. FormalTimerPar ::= [InOutParKeyword] TimerKeyword TimerParIdentifier
444. TimerParIdentifier ::= Identifier
445. FormalTemplatePar ::= [InParKeyword] TemplateKeyword Type
              TemplateParIdentifier
446. TemplateParIdentifier ::= Identifier
```

A.1.6.7 Enunciado With

```
447. WithStatement ::= WithKeyword WithAttribList
448. WithKeyword ::= "with"
449. WithAttribList ::= "{" MultiWithAttrib "}"
450. MultiWithAttrib ::= {SingleWithAttrib [SemiColon]}+
451. SingleWithAttrib ::= AttribKeyword  [OverrideKeyword] [AttribQualifier]
              AttribSpec
452. AttribKeyword ::= EncodeKeyword |
              DisplayKeyword |
              ExtensionKeyword
453. EncodeKeyword ::= "encode"
454. DisplayKeyword ::= "display"
455. ExtensionKeyword ::= "extension"
456. OverrideKeyword ::= "override"
457. AttribQualifier ::= "(" DefOrFieldRefList ")"
458. DefOrFieldRefList ::= DefOrFieldRef {"," DefOrFieldRef}
459. DefOrFieldRef ::= DefinitionRef | FieldReference
460. DefinitionRef ::= StructTypeIdentifier |
              EnumTypeIdentifier |
              PortTypeIdentifier |
              ComponentTypeIdentifier |
              SubTypeIdentifier |
              ConstIdentifier |
              TemplateIdentifier |
              NamedAltIdentifier |
              TestcaseIdentifier |
```

```
                FunctionIdentifier |
                SignatureIdentifier
461. AttribSpec ::= FreeText
```

A.1.6.8 Enunciados de comportamiento

```
462. BehaviourStatements ::= TestcaseInstance |
                FunctionInstance |
                ReturnStatement |
                AltConstruct |
                InterleavedConstruct |
                LabelStatement |
                GotoStatement |
                ActivateStatement |
                DeactivateStatement |
                NamedAltInstance
/* SEMÁNTICA ESTÁTICA - TestcaseInstance no será llamado dentro de un caso de
prueba o función en ejecución existentes llamados desde un caso de prueba, es
decir, los casos de prueba sólo pueden ser ejemplificados desde la parte de
control o desde funciones llamadas directamente desde la parte de control */
463. VerdictStatements ::= SetLocalVerdict
464. VerdictOps ::= GetLocalVerdict
465. SetLocalVerdict ::= SetVerdictKeyword "(" SingleExpression ")"
/* SEMÁNTICA ESTÁTICA - SingleExpression se resolverá en un valor de tipo
veredicto */
/* SEMÁNTICA ESTÁTICA - SetLocalVerdict  no se usará para asignar el valor
error */
466. SetVerdictKeyword ::= VerdictKeyword Dot SetKeyword
467. GetLocalVerdict ::= VerdictKeyword Dot GetKeyword
468. GetKeyword ::= "get"
469. SUTStatements ::= SUTAction "(" (FreeText | TemplateRefWithParList)  ")"
470. SUTAction ::= SUTKeyword Dot ActionKeyword
471. SUTKeyword ::= "sut"
472. ActionKeyword ::= "action"
473. ReturnStatement ::= ReturnKeyword  [Expression]
474. AltConstruct ::= AltKeyword BeginChar AltGuardList EndChar
475. AltKeyword ::= "alt"
476. AltGuardList ::= {AltGuardElement [SemiColon]}+ [ElseStatement [SemiColon]]
477. AltGuardElement ::= GuardStatement | ExpandStatement
478. GuardStatement ::= AltGuardChar GuardOp  StatementBlock
479. ExpandStatement ::= "["ExpandKeyword "]" NamedAltInstance
480. ElseStatement ::= "["ElseKeyword "]" StatementBlock
481. ExpandKeyword ::= "expand"
482. AltGuardChar ::= "[" [BooleanExpression] "]"
483. GuardOp ::= TimeoutStatement | ReceiveStatement | TriggerStatement |
                GetCallStatement | CatchStatement | CheckStatement |
                GetReplyStatement | DoneStatement
/* SEMÁNTICA ESTÁTICA - GuardOp usada dentro de la parte de control del módulo
sólo contendrá  timeoutStatement */
484. StatementBlock ::= BeginChar [FunctionStatementOrDefList] EndChar
485. InterleavedConstruct ::= InterleavedKeyword BeginChar InterleavedGuardList
                EndChar
486. InterleavedKeyword ::= "interleave"
487. InterleavedGuardList ::= {InterleavedGuardElement [SemiColon]}+
488. InterleavedGuardElement ::= InterleavedGuard InterleavedAction
489. InterleavedGuard ::= "[" "]" GuardOp
490. InterleavedAction ::= StatementBlock
/* SEMÁNTICA ESTÁTICA - StatementBlock no puede contener enunciados de bucle,
goto, activate, deactivate, stop, return o llamadas a funciones */
491. LabelStatement ::= LabelKeyword LabelIdentifier
492. LabelKeyword ::= "label"
493. LabelIdentifier ::= Identifier
494. GotoStatement ::= GotoKeyword (LabelIdentifier | AltKeyword)
```

```
/* SEMÁNTICA ESTÁTICA - La opción AltKeyword sólo se puede usar dentro de una
construcción ALT */
495. GotoKeyword ::= "goto"
496. ActivateStatement ::= ActivateKeyword "(" NamedAltList ")"
497. ActivateKeyword ::= "activate"
498. NamedAltList ::= NamedAltInstance {"," NamedAltInstance}
499. DeactivateStatement ::= DeactivateKeyword ["(" NamedAltRefList ")"]
500. DeactivateKeyword ::= "deactivate"
501. NamedAltRefList ::= NamedAltRef {"," NamedAltRef}
```

A.1.6.9 Enunciados básicos

```
502. BasicStatements ::= Assignment | LogStatement | LoopConstruct |
               ConditionalConstruct
503. Expression ::= SingleExpression | CompoundExpression
/* SEMÁNTICA ESTÁTICA - Expression no contendrá operaciones Configuration o
verdict dentro de la parte de control del módulo */
504. CompoundExpression ::= FieldExpressionList | ArrayExpression
505. FieldExpressionList ::= "{" FieldExpressionSpec {"," FieldExpressionSpec}
               "}"
506. FieldExpressionSpec ::= FieldReference AssignmentChar Expression
507. ArrayExpression ::= "{" [ArrayElementExpressionList] "}"
508. ArrayElementExpressionList ::= NotUsedOrExpression  {","
               NotUsedOrExpression}
509. NotUsedOrExpression ::= Expression | NotUsedSymbol
510. ConstantExpression ::= SingleConstExpression | CompoundConstExpression
511. SingleConstExpression ::= SingleExpression
/* SEMÁNTICA ESTÁTICA - SingleConstExpression no contendrá variables ni
parámetro de módulo y se resolverá en un valor de constante en el tiempo de
compilación */
512. BooleanExpression ::= SingleExpression
/* SEMÁNTICA ESTÁTICA - BooleanExpression se resolverá en un valor de tipo
Boolean */
513. CompoundConstExpression ::= FieldConstExpressionList | ArrayConstExpression
514. FieldConstExpressionList ::= "{" FieldConstExpressionSpec {","
               FieldConstExpressionSpec} "}"
515. FieldConstExpressionSpec ::= FieldReference AssignmentChar
               ConstantExpression
516. ArrayConstExpression ::= "{" [ArrayElementConstExpressionList] "}"
517. ArrayElementConstExpressionList ::= ConstantExpression {","
               ConstantExpression}
518. Assignment ::= VariableRef  ":=" Expression
/* SEMÁNTICA OPERACIONAL - La Expression en RHS de Assignment dará un valor
explícito del tipo de LHS. */
519. SingleExpression ::= SimpleExpression {BitOp SimpleExpression}
/* SEMÁNTICA OPERACIONAL - Si SimpleExpressions y  BitOp existen,
SimpleExpressions dará valores específicos de tipos compatibles */
520. SimpleExpression ::= SubExpression [RelOp SubExpression]
/* SEMÁNTICA OPERACIONAL - Si SubExpressions y RelOp existen,  SubExpressions
dará valores específicos de tipos  compatibles. */
/* SEMÁNTICA OPERACIONAL - Si RelOp es "<" | ">" | ">=" | "<=", cada
SubExpression dará un valor integer, Enumerated o float específico (estos
valores pueden ser valores TTCN o ASN.1) */
521. SubExpression ::= Product [ShiftOp Product]
/* SEMÁNTICA OPERACIONAL - Cada Product dará un valor específico. Si existe más
de un Product, el operando de la derecha será del tipo integer y si la operación
shift es '<<' or '>>' , el operando de la izquierda se resolverá en un tipo
bitstring, hexstring, octetstring o integer. Si la operación shift es '<@' o
'@>', el operando de la izquierda será del tipo   bitstring, hexstring,
charstring o universal charstring */
522. Product ::= Term {AddOp Term}
/* SEMÁNTICA OPERACIONAL - Cada Term se resolverá en un valor específico. Si
existe más de un Term, se resolverán en el tipo integer o float. */
523. Term ::= Factor {MultiplyOp Factor}
```

```
/* SEMÁNTICA OPERACIONAL – Cada Factor se resolverá en un valor específico. Si
existe más de un Factor, se resolverán en el tipo integer o float. */
524. Factor ::= [UnaryOp] Primary
/* SEMÁNTICA OPERACIONAL – Primary se resolverá en un valor específico. Si
existe  UnaryOp y es "not", Primary se resolverá en el tipo BOOLEAN, si UnaryOp
es "+" o "-", Primary se resolverá en el tipo integer o float. Si UnaryOp se
resuelve en  not4b, Primary se resolverá en el tipo bitstring, hexstring u
octetstring. */
525. Primary ::= OpCall | Value | "(" SingleExpression ")"
526. ExtendedFieldReference ::= {(Dot StructFieldIdentifier | ArrayOrBitRef)}+
527. OpCall ::= ConfigurationOps | VerdictOps | TimerOps | TestcaseInstance |
               FunctionInstance | TemplateOps
528. AddOp ::= "+" | "-"
/* SEMÁNTICA OPERACIONAL – Los operandos de los operadores "+" o "-" serán del
tipo integer o float (es decir, TTCN o ASN.1 predefinidos) o derivados de
integer o float (es decir, subgama) */
529. MultiplyOp ::= "*" | "/" | mod | rem
/* SEMÁNTICA OPERACIONAL – Los operandos de los operadores "*", "/", rem o mod
serán del tipo integer o  float (es decir,  TTCN o ASN.1 predefinidos) o
derivados de integer o float (es decir, subgama). */
530. UnaryOp ::= "+" | "-" | not | not4b
/* SEMÁNTICA OPERACIONAL – Los operandos de los operadores  "+" o "-" serán de
tipo integer o float (es decir, TTCN o ASN.1 predefinidos) o derivados de
integer o float (es decir, subgama). Los operandos del operador "not" serán del
tipo boolean (TTCN o ASN.1) o derivados del tipo Boolean.  Los operandos del
operador  not4b serán del tipo bitstring, octetstring o hexstring. */
531. RelOp ::= "==" | "<" | ">" | "!=" | ">=" | "<="
/* SEMÁNTICA OPERACIONAL – La precedencia de los operadores se define en el
cuadro 7 */
532. BitOp ::= "and4b" | xor4b |or4b | and | xor | or | StringOp
/* SEMÁNTICA OPERACIONAL – Los operandos de los operadores and, or o xor serán
de tipo boolean (TTCN o ASN.1) o derivados de tipo Boolean. Los operandos de los
operadores and4b, or4b o xor4b serán del tipo bitstring, hexstring u octetstring
(TTCN o ASN.1) o derivados de estos tipos. */
/* SEMÁNTICA OPERACIONAL – La precedencia de los operadores se define en el
cuadro 7 */
533. StringOp ::= "&"
/* SEMÁNTICA OPERACIONAL – Los operandos del operador string serán  bitstring,
hexstring, octetstring o character string */
534. ShiftOp ::= "<<" | ">>" | "<@" | "@>"
535. LogStatement ::= LogKeyword "(" [FreeText] ")"
536. LogKeyword ::= "log"
537. LoopConstruct ::= ForStatement |
               WhileStatement |
               DoWhileStatement
538. ForStatement ::= ForKeyword "(" Initial [SemiColon] Final [SemiColon] Step
               ")"
               StatementBlock
539. ForKeyword ::= "for"
540. Initial ::= VarInstance | Assignment
541. Final ::= BooleanExpression
542. Step ::= Assignment
543. WhileStatement ::= WhileKeyword "(" BooleanExpression ")"
               StatementBlock
544. WhileKeyword ::= "while"
545. DoWhileStatement ::= DoKeyword StatementBlock
               WhileKeyword "(" BooleanExpression ")"
546. DoKeyword ::= "do"
547. ConditionalConstruct ::= IfKeyword "(" BooleanExpression ")"
               StatementBlock
               {ElseIfClause}   [ElseClause]
548. IfKeyword ::= "if"
549. ElseIfClause ::= ElseKeyword  IfKeyword "(" BooleanExpression ")"
               StatementBlock
```

```
550. ElseKeyword ::= "else"
551. ElseClause ::= ElseKeyword  StatementBlock
```

A.1.6.10 Producciones varias

```
552. Dot ::= "."
553. Dash ::= "-"
554. Minus ::= Dash
555. SemiColon ::= ";"
556. Colon ::= ":"
557. Underscore ::= "_"
558. BeginChar ::= "{"
559. EndChar ::= "}"
560. AssignmentChar ::= ":="
```

Anexo B

Semántica operacional

Este anexo define el significado de un comportamiento en TTCN-3 de manera intuitiva e inequívoca. La semántica operacional no es formal, por lo que la capacidad de ejecutar pruebas matemáticas basadas en esta semántica es muy limitada.

Esta semántica operacional proporciona una visión de los estados en la ejecución de un módulo TTCN. Se introducen diferentes clases de estados y el significado de las diferentes construcciones TTCN-3 se describe:

1) utilizando información de estados para definir las condiciones previas para la ejecución de una construcción; y
2) definiendo cómo la ejecución de una construcción cambiará un estado.

La semántica operacional está restringida al significado de comportamiento en TTCN-3, es decir, funciones, casos de prueba, control de módulo y construcciones de lenguaje para definir comportamientos de prueba, por ejemplo, operaciones **`send`** y **`receive`**, enunciados **`if-else`** o **`while`**. El significado de varias construcciones TTCN-3 se explica sustituyéndolas con otras construcciones de lenguaje. Por ejemplo, las alternativas denominadas son macros y su significado se explica completamente sustituyendo todas las referencias a macros por las correspondientes definiciones de macro. Esto incluye el tratamiento de comportamiento por defecto.

En la mayoría de los casos, la definición de la semántica en un lenguaje se basa en un árbol de sintaxis abstracta del código que se describirá. Esta semántica no funciona en un árbol de sintaxis abstracta pero requiere una representación gráfica de descripciones de comportamiento en TTCN-3 en forma de gráficos de flujo o flujogramas. Un flujograma describe el flujo de control en un caso de prueba, función o el control de módulo. La correspondencia de las descripciones de comportamiento de TTCN-3 con los flujogramas es directa.

B.1 Estructura de este anexo

Este anexo se estructura en dos partes:

1) La primera parte (véase B.2) define el significado de notaciones abreviadas y llamadas de macros TTCN-3 mediante su sustitución por otras construcciones de lenguaje TTCN-3. Estas sustituciones en un módulo TTCN-3 se pueden considerar como un paso previo al procesamiento antes de que el módulo pueda ser interpretado de acuerdo con la siguiente descripción de semántica operacional.
2) La segunda parte (véase B.3) describe la semántica operacional de TTCN-3 por medio de la interpretación de flujogramas y modificación de estados.

B.2 Sustitución de notaciones abreviadas y llamadas de macro

Las notaciones abreviadas tienen que ser ampliadas y las referencias a macros tienen que ser sustituidas por las correspondientes definiciones en un nivel textual antes de poder utilizar esta semántica operacional para la explicación del comportamiento en TTCN-3.

Las notaciones abreviadas de TTCN-3 son:

- operaciones receptoras autónomas;
- operaciones **`trigger`**;
- usos de la palabra clave **`any`** en operaciones de temporizador y receptoras;
- usos de la palabra clave **`all`** en operaciones de temporizador y de puertos;
- enunciados **`return`** y **`stop`** omitidos al final de las definiciones de funciones y de casos de prueba.

Los macros de TTCN-3 son alternativas denominadas, es decir, definiciones **`named alt`** y son invocadas:

- explícitamente en vez de un enunciado **`alt`**, es decir, aparecen como una llamada de función;
- explícitamente en enunciados **`alt`** utilizando la palabra clave **`expand`**;
- implícitamente cuando son referenciadas como comportamiento por defecto en enunciados **`activate`** y **`deactivate`**.

Además de las notaciones abreviadas y las llamadas de macro, la semántica operacional requiere un tratamiento especial de los parámetros de módulo y constantes globales, es decir, constantes que son definidas en la parte de definiciones del módulo. Todas las referencias a parámetros de módulo y constantes globales serán sustituidas por valores concretos. Esto significa que se supone que el valor de parámetros de módulo y constantes globales puede ser determinado antes de que la semántica operacional sea aplicable.

NOTA 1 – El tratamiento de parámetros de módulo y constantes globales en la semántica operacional será diferente de su tratamiento en un compilador TTCN-3. La semántica operacional describe el significado de comportamiento TTCN-3 y no es una directriz para la implementación de un compilador TTCN-3.

NOTA 2 – La semántica operacional trata parámetros y constantes locales en componentes de prueba, casos de prueba, funciones y control de módulo como variables. El uso erróneo de constantes locales en los parámetros **`in`**, **`out`** e **`inout`** tiene que ser comprobado estáticamente.

B.2.1 Orden de pasos de sustitución

Las sustituciones textuales de notaciones abreviadas, llamadas de macro, constantes globales y parámetros de módulo han de hacerse en el siguiente orden:

1) adición de los enunciados **`stop`** y **`return`** en el control de módulo, funciones y casos de prueba;
2) sustitución de constantes globales y parámetros de módulo por valores concretos;
3) inserción de operaciones receptoras autónomas en enunciados **`alt`**;
4) expansión de macros de *llamadas de macro* puras, lo que significa:
   - expansiones explícitas de enunciados **`alt`** que incluyen la palabra clave **`expand`** (y hacen referencia a una definición **`named alt`**);
   - expansión explícita de llamadas de definiciones **`named alt`**.
5) expansión de enunciados **`interleave`**;
6) expansión de comportamiento por defecto;

7) sustitución de todas las operaciones **trigger** por operaciones **receive** equivalentes y enunciados **goto**;

8) sustitución de todos los usos de las palabras clave **any** y **all** en operaciones de temporizador y de puertos.

NOTA – Si no se mantiene este orden de pasos de sustitución, el resultado de las sustituciones no representará el comportamiento definido.

B.2.2 Adición de operaciones stop y return en descripciones de comportamiento

TTCN-3 permite dejar el control de módulo, los casos de prueba y funciones que no devuelven ningún valor sin especificar una operación **stop** o **return** explícita. Para la semántica operacional se supone que se añaden las operaciones **return** y **stop** omitidas, es decir, se añaden operaciones **stop** en un control de módulo y casos de prueba y operaciones **return** en funciones.

Ejemplo:

```
// La definición de función y caso de prueba sin enunciados return y stop
// explícitos al final de su descripción de comportamiento

   function MyFunction(inout integer MyPar) {
                                // MyFunction no devuelve un valor pero cambia el
      MyPar := 10 * MyPar1;     // valor de MyPar que es pasado por referencia
      if (MyPar == 999) stop;   // Detiene la ejecución si MyPar tiene el
                                // valor 999

      // Devolución de IMPLICIT si MyPar != 999
      }

   testcase MyTestCase() runs on MyMTCtype {
      MyMTCbehaviour();      // Función que define el comportamiento del MTC

      // Parada de IMPLICIT después de devolución de MyMTCbehaviour
      }

// MyFunction y MyTestCase tras añadir operaciones return y stop explícitas

   function MyFunction(inout integer MyPar) {
                                // MyFunction no devuelve un valor pero cambia el
      MyPar := 10 * MyPar1;     // valor de MyPar que es pasado por referencia
      if (MyPar == 999) stop;   // Detiene la ejecución si MyPar tiene el
                                // valor 999

         return;                // devuelve EXPLICIT
      }

      testcase MyTestCase() runs on MyMTCtype {

      MyMTCbehaviour();      // Función que define el comportamiento del MTC

      stop;                  // Parada de EXPLICIT
      }
```

B.2.3 Sustitución de constantes globales y parámetros de módulo

Las constantes declaradas en la parte de definiciones del módulo son globales para el control de módulo y todos los componentes de prueba que son creados durante la ejecución de un módulo TTCN-3. Los parámetros de módulo son constantes globales en el tiempo de ejecución.

Todas las referencias a constantes globales y parámetros de módulo serán sustituidas por los valores reales antes de que la semántica operacional comience la interpretación del módulo. Si el valor de una constante o parámetro de módulo se da en forma de una expresión, la expresión tiene que ser

evaluada y el resultado de la evaluación sustituirá a todas las referencias de la constante o parámetro de módulo.

B.2.4 Inserción de operaciones receptoras en enunciados alt

Las operaciones receptoras de TTCN-3 son: **receive**, **trigger**, **getcall**, **getreply**, **catch**, **check**, **timeout**, y **done**.

NOTA – Las operaciones **receive**, **trigger**, **getcall**, **getreply**, **catch** y **check** funcionan en puertos y permiten ramificación debido a la recepción de mensajes, llamadas de procedimiento, respuesta y excepciones. Las operaciones **timeout** y **done** no son operaciones receptoras reales, pero pueden ser utilizadas de la misma manera como operaciones receptoras, es decir, como alternativas en enunciados **alt**. Por tanto, la semántica operacional trata **timeout** y **done** como operaciones receptoras.

Una operación receptora puede ser utilizada como enunciado autónomo en una función, una alternativa denominada o un caso de prueba. En este caso, se considera que la operación receptora es una notación abreviada para un enunciado **alt** con sólo una alternativa definida por la operación receptora. Para la semántica operacional, un enunciado **alt** en el cual se inserta el enunciado receptor sustituirá a todas las ocurrencias autónomas de operaciones receptoras.

Ejemplo:

```
// La ocurrencia de
    :
   MyCL.trigger(MyType:*);
    :

// será sustituida por
    :
   alt {
      [] MyCL.trigger (MyType:*);
   }
    :

// o
    :
   MyPTC.done;
    :

// será sustituida por
    :
   alt {
      [] MyPTC.done;
   }
    :
```

B.2.5 Expansión de macro

La expansión de macro en TTCN-3 se relaciona con el uso de alternativas denominadas (definiciones de **named alt**) en enunciados **alt** o en vez de enunciados **alt**, es decir, la definición **named alt** es referenciada de manera similar a una llamada de función en una secuencia de enunciados.

B.2.5.1 Expansión de alternativas denominadas en enunciados alternativos

La expansión de alternativas denominadas en enunciados **alt** se relaciona con las ramas alternativas indicadas por la palabra clave **expand** colocada entre corchetes cuadrados seguida por una referencia a una definición **named alt** (como único enunciado de esa rama). En este caso, las ramas alternativas de la alternativa denominada referenciada sustituyen a la rama con la palabra clave **expand**. Para la semántica operacional, se supone que esta sustitución se hace en un nivel sintáctico. Un ejemplo de esta expansión figura en la parte principal de la presente Recomendación.

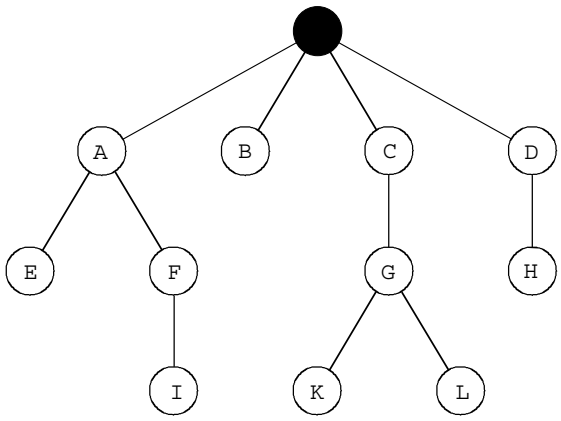
B.2.5.2 Llamada explícita de una alternativa denominada

Las alternativas denominadas pueden ser referenciadas también de manera similar a una llamada de función en una secuencia de enunciados. En este caso, la referencia será ampliada por la correspondiente definición **named alt**. Un ejemplo de esta expansión figura en el cuerpo principal de la presente Recomendación.

B.2.6 Sustitución de la construcción interleave

El significado del enunciado **interleave** es definido por sustitución por una serie de enunciados **alt** jerarquizados que tienen el mismo significado. El algoritmo para la construcción de la sustitución para un enunciado **interleave** se describe en esta cláusula. La sustitución se hará en un nivel sintáctico.

Una serie de enunciados **alt** pueden ser descritos por medio de un árbol. Tres nodos representan los enunciados en los enunciados **alt**. Una ramificación indica un enunciado **alt** y los enunciados en la misma rama describen enunciados en la misma alternativa. Esto se muestra esquemáticamente en la figura B.1. La figura B.1 a) presenta un árbol y la figura B.1 b) muestra la representación correspondiente en forma de una serie de enunciados **alt** jerarquizados.




```
alt {

    []   A  {
             alt {
                  [] E ;
                  [] F {
                        I
                     }
             }
         }
    []   B;
    []   C  {
             G;
             alt {
                  [] K;
                  [] L;
             }
         }
    []   D  {
                  H
         }
}
```

T1012780-01

a) Árbol **b) Representación de a) como TTCN-3**

Figura B.1/Z.140 – Enunciados alt jerarquizados y la representación correspondiente en forma de árbol

A continuación se presenta la construcción de una representación en árbol de un enunciado **interleave**. La transformación del árbol en la serie de enunciados **alt** jerarquizados es directa y no necesita ulterior explicación.

Un enunciado **interleave** puede ser considerado como un conjunto ordenado parcial (***POS, partial ordered set***) de enunciados TTCN-3 permitidos. Formalmente:

- $\boldsymbol{POS} = (\boldsymbol{S}, <)$

  donde:

  $\boldsymbol{S}$ es el conjunto de enunciados TTCN-3 permitidos; y

  $< \subseteq (\boldsymbol{S} \times \boldsymbol{S})$ describe la relación de orden reflexiva y transitiva.

El término *enunciados TTCN-3 permitidos* indica el hecho de que no se permite utilizar los enunciados de transferencia de control **for**, **while**, **do-while**, **goto**, **activate**, **deactivate**, **stop**, **return** y llamadas de funciones definidas por el usuario que incluyen operaciones de comunicación en enunciados **interleave**. Además, no se permite tampoco guardar ramas de un enunciado **interleave** con expresiones Boolean para ampliar los enunciados **interleave** con alternativas denominadas o para especificar ramas de **else**.

Para el algoritmo de construcción hay que definir las siguientes funciones:

- La función *DISCARD (DESCARTAR)* suprime un elemento **s** de un conjunto parcialmente ordenado ***POS*** y devuelve el conjunto parcialmente ordenado resultante ***POS'***:

  $$\underline{DISCARD}(\boldsymbol{s}, \boldsymbol{POS}) = \boldsymbol{POS'}$$

  donde:

  $\boldsymbol{POS'} = (\boldsymbol{S'}, <')$; y

  $\boldsymbol{S'} = \boldsymbol{S} \backslash \{\boldsymbol{s}\}$; y

  $<' = < \cap (\boldsymbol{S} \backslash \{\boldsymbol{s}\} \times \boldsymbol{S} \backslash \{\boldsymbol{s}\})$.

- La función *ENABLED (HABILITADO)* toma un conjunto parcialmente ordenado $\boldsymbol{POS} = (\boldsymbol{S}, <)$ y devuelve todos los elementos que no tienen predecesores en ***POS***:

  $$\underline{ENABLED}(\boldsymbol{POS}) = \{ \boldsymbol{s} \mid \boldsymbol{s} \in \boldsymbol{S} \wedge (< \cap (\boldsymbol{S} \times \{\boldsymbol{s}\}) = \varnothing)\}$$

- La función *RECEIVING (RECEPCIÓN)* toma un conjunto de enunciados TTCN-3 ***S*** y devuelve todos los enunciados receptores de este conjunto.

  $$\underline{RECEIVING}(\boldsymbol{S}) = \{ \boldsymbol{s} \mid \boldsymbol{s} \in \boldsymbol{S} \wedge \underline{kind}(\boldsymbol{s}) \in \{\texttt{receive}, \texttt{trigger}, \texttt{getcall}, \texttt{getreply}, \texttt{catch}, \texttt{check}, \texttt{done}, \texttt{timeout}\}\}$$

- La función *SELECT (SELECCIONAR)* selecciona aleatoriamente un elemento ***s*** de un conjunto ***S*** dado y devuelve ***s***.

  $$\underline{SELECT}(\boldsymbol{S}) = \boldsymbol{s} \quad \text{donde } \boldsymbol{s} \in \boldsymbol{S}$$

NOTA – La función *kind (clase)* en la función *RECEIVING* anterior no está definida formalmente. *kind* (o tipo) devuelve la clase de un enunciado TTCN-3 dado.

El algoritmo de construcción del árbol es un procedimiento recursivo donde en cada llamada recursiva se construyen los nodos sucesores para un nodo dado. El procedimiento se proporciona en una notación de pseudocódigo similar a C que utiliza las funciones definidas anteriormente y cierta notación matemática adicional:

```
CONSTRUCT-SUCCESSORS (treeNode *predecessor, partiallyOrderedSet POS) {
   // treeNode hace referencia al tipo de nodo del árbol que se ha de construir
   // partiallyOrderedSet indica tipo para un conjunto parcialmente ordenado de
   // enunciados TTCN-3

   var statement myStmt;                  // para el almacenamiento de un
                                          // enunciado TTCN-3
   var treeNode *newSonNode;              // para el tratamiento de nuevos nodos
                                          // de árbol

// EXTRACCIÓN DE CONJUNTOS DE ENUNCIADOS TTCN-3 QUE NO TIENEN PREDECESORES
// EN 'POS'
   var statementSet enabStmts := ENABLED(POS);
                       // Todos los enunciados sin predecesor
   var statementSet enabRecStmts := RECEIVING(enabStmts);
                       // Enunciados receptores en 'enabStmts'
   var statementSet enabNonRecStmts := enabStmts\enabRecStmts;
                       // Enunciados no receptores en 'enabStmts'
```

```
   if (POS == ∅)
      return;          // CRITERIO DE TERMINACIÓN DE RECURSIÓN
   else {
      if (enabNonRecStmts != ∅) { // Tratamiento de enunciados no receptores en
                                  // 'enabStmts'
         myStmt := SELECT(enabNonRecStmts);
         newSonNode := create(myStmt, predecessor);
                  // Creación de un nuevo nodo de árbol que representa 'myStmt'
                  // en el árbol y actualizacion de punteros en el 'newSonNode' y
                  // 'predecessor'.
         CONSTRUCT-SUCCESSORS(newSonNode, DISCARD(myStmt, POS));
                  // SIGUIENTE PASO DE RECURSIÓN
      }
      else {   // Tratamiento de eventos receptores, el árbol se ramificará
         for each (myStmt in enabRecStmts) {
               newSonNode := create(myStmt, predecessor); // Nuevo nodo de árbol
               CONSTRUCT-SUCCESSORS(newSonNode, DISCARD (myStmt, POS));
               // SIGUIENTE(S) PASO(S) DE RECURSIÓN
         }
      }
   }
}
```

Inicialmente, la función CONSTRUCT-SUCCESSORS será llamada con un *nodo raíz* de un árbol vacío y el conjunto parcialmente ordenado de enunciados TTCN-3 que describen el enunciado **`interleave`** que será sustituido. Tras la terminación, el *nodo raíz* puede ser utilizado para acceder al árbol construido.

B.2.7 Expansión de comportamientos por defecto

El mecanismo de comportamiento por defecto TTCN-3 se define por medio de un mecanismo de expansión de macro. El comportamiento por defecto tiene que ser proporcionado en forma de definiciones **`named alt`**. Se hace referencia a una definición **`named alt`** utilizada como comportamiento por defecto en un enunciado **`activate`**. El alcance de un comportamiento por defecto es determinado por un enunciado **`deactivate`** y enunciados de **`activate`** correspondientes o por un enunciado **`activate`** y el final de la función o caso de prueba en el cual se utiliza el enunciado **`activate`**. Dentro de este ámbito, las alternativas de enunciados **`alt`** son ampliadas por el comportamiento especificado en las definiciones **`named alt`** activadas. La semántica operacional supone que esta extensión se efectúa en el nivel sintáctico. En el cuerpo principal de la presente Recomendación figura un ejemplo del mecanismo de extensión.

B.2.8 Sustitución de operaciones Trigger

La operación **`trigger`** filtra mensajes, con un determinado criterio de concordancia, de un tren de mensajes en un puerto dado. La semántica de la operación **`trigger`** puede ser descrita por su sustitución con dos operaciones **`receive`** y un enunciado **`goto`**. La semántica operacional supone que esta sustitución se efectúa en el nivel sintáctico.

Ejemplo:

```
// La siguiente operación trigger ...

   alt   {
         [] MyCL.trigger (MyType:*);
         }
```

```
// será sustituida por ...

   alt   {
         [] MyCL.receive (MyType:*);
         [] MyCL.receive {
               goto alt
         }
   }
```

Si se utiliza el enunciado **trigger** en más de un enunciado **alt** complejo, la sustitución se efectúa de la misma manera.

Ejemplo:

```
// El siguiente enunciado alt incluye un enunciado trigger ...
   alt   {
         [] PCO2.receive {
               stop;
            }
         [] MyCL.trigger (MyType:*);
         [] PCO3.catch {
               verdict.set(fail);
               stop;
            }
      }

// que será sustituido por

   alt   {
         [] PCO2.receive {
               stop;
            }
         [] MyCL.receive (MyType:*);
         [] MyCL.receive {
               goto alt;
            }
         [] PCO3.catch {
               verdict.set(fail);
               stop;
            }
      }
```

B.2.9 Sustitución de las palabras clave 'any' y 'all'

Se permite utilizar la palabra clave **any** para:

- las operaciones de temporizador **running** y **timeout;**
- las operaciones receptoras **receive**, **trigger**, **getcall**, **getreply**, **catch**, **check**.

Se permite utilizar la palabra clave **all** para:

- la operación de temporizador **stop**;
- las operaciones de puerto **start**, **stop** y **clear**.

Se permite utilizar ambas palabras clave para:

- las operaciones **done** y **running** para componentes.

B.2.9.1 Sustitución de 'all' en operaciones de temporizador y de puerto

La aplicación de operaciones de temporizador y de puerto se relaciona con el ámbito en el cual se utilizan. Esto significa que la palabra clave **all** direcciona a todos los temporizadores y puertos conocidos en la unidad de ámbito en la cual se utiliza **all** (+ operación). La sustitución de los usos de **all** en operaciones de temporizador y de puerto es directa.

El uso de **all port** en operaciones **start**, **stop**, o **clear** se sustituirá con una operación **start**, **stop**, o **clear** separada para cada puerto conocido. El uso de **all timer** en una operación **stop** será sustituido por una operación **stop** separada para cada temporizador conocido.

Ejemplo:

```
// Se supone que se conocen los puertos PCO1, PCO2 y los temporizadores T1 y T2

    :
   all port.clear;
    :
    :
   all timer.stop;
    :

// serán sustituidos por

    :
   PCO1.clear;
   PCO2.clear;
    :
    :
   T1.stop;
   T2.stop;
    :
```

B.2.9.2 Sustitución de 'any' en operaciones de temporizador y receptoras

La aplicación de operaciones de temporizador y receptoras se relaciona con el ámbito en el cual se utilizan. Esto significa que la palabra clave **any** direcciona a todos los temporizadores y puertos (en el caso de operaciones receptoras) conocidos en la unidad de ámbito en la cual se utiliza **any** (+ operación). La sustitución de los usos de **any** en operaciones de temporizador y receptoras es directa.

El uso de **any port** en una operación **receive**, **trigger**, **getcall**, **getreply**, **catch** o **check** se sustituirá con operaciones alternativas separadas para cada puerto conocido y posible. Posible significa que una ocurrencia de **any port.receive** sólo es pertinente para puertos basados en mensaje.

El uso de **any timer** en una operación **timeout** se sustituirá con operaciones alternativas separadas para cada temporizador conocido en la unidad de ámbito.

Ejemplo:

```
// Se supone que se conocen los puertos PCO1, PCO2 y los temporizadores T1 y T2

   alt   {
         [] PCO2.receive {
               aTestStep();
            }
         [] any port.receive {
               verdict.set(fail);
               stop;
            }
         [] any timer.timeout {
               verdict.set(fail);
               stop;
            }
      }
```

```
// serán sustituidos por

   alt   {
         [] PCO2.receive {
               stop;
            }
         [] PCO1.receive {
               verdict.set(fail);
               stop;
            }
         [] PCO1.receive {
               verdict.set(fail);
               stop;
            }
         [] T1.receive {
               verdict.set(fail);
               stop;
            }
         [] T2.receive {
               verdict.set(fail);
               stop;
            }
      }
```

El uso de **any timer** en una operación **running** se sustituirá con operaciones **running** separadas para todos los temporizadores conocidos en la unidad de ámbito que están combinados por medio de operadores **or**.

Ejemplo:

```
// Se supone que se conocen los temporizadores T1 y T2 en la unidad de ámbito

    :
   if (any timer.running) {
      verdict.set(fail);
      stop;
   }
    :

// serán sustituidos por

    :
   if (T1.running or T2.running) {
      verdict.set(fail);
      stop;
   }
    :
```

B.2.9.3 Las palabras clave 'any' y 'all' en 'done' y 'running'

Las operaciones **any component.done**, **all component.done**, **any component.running** y **all component.running** sólo pueden ser ejecutadas por el MTC. Debido a la creación de componentes de prueba dinámicos, el MTC puede no conocer todos los componentes que han sido creados durante la ejecución del caso de prueba. Por tanto, la ejecución de estas operaciones requiere comunicación con el medio de prueba. Se supone pues que **any component.done**, **all component.done**, **any component.running** y **all component.running** son instrucciones del sistema, es decir, que no pueden ser sustituidas por otras instrucciones.

B.3 Semántica de gráficos de flujo de TTCN-3

La semántica operacional de TTCN-3 se basa en la interpretación de gráficos de flujos o flujogramas. En esta cláusula, se presentan los flujogramas (véase B.3.1), se explica la construcción de flujogramas que representan control de módulo TTCN-3, casos de prueba, funciones y definiciones de tipos de componentes (véase B.3.2). Se definen estados de módulo y de componente para la descripción de los estados de ejecución de un módulo TTCN-3 (véase B.3.3), se describe el tratamiento de mensajes, llamadas de procedimiento distantes, respuestas a llamadas de procedimientos distantes y excepciones (véase la B.3.4), se explica el procedimiento de evaluación de control de módulo y casos de prueba (véase B.3.6) y se describe el significado de los diferentes enunciados TTCN-3 (véase B.3.7).

B.3.1 Flujogramas

Un flujograma es un gráfico cuyos flujos tienen sentidos de dirección y que consta de nodos etiquetados y bordes etiquetados. El trayecto a través de un flujograma describe el flujo de control durante la ejecución de una descripción de comportamiento representada.

B.3.1.1 Marco de flujograma

Un flujograma se pondrá en un marco que define la frontera del flujograma. El nombre del flujograma sigue a la palabra clave **flow graph** (que no es una palabra clave del lenguaje núcleo de TTCN-3) y se pondrá en la esquina superior izquierda del flujograma. Por convenio, se supone que el nombre del flujograma hace referencia a la descripción de comportamiento TTCN representada en el flujograma. En la figura B.2 se muestra un flujograma simple.

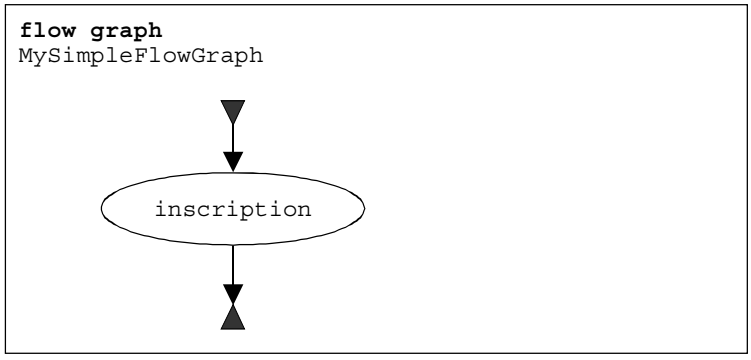


Figura B.2/Z.140 – Flujograma simple

B.3.1.2 Nodos de flujograma

Los flujogramas consisten en *nodos de comienzo*, *nodos de fin*, *nodos básicos* y *nodos de referencia*.

B.3.1.2.1 Nodos de comienzo

El nodo de comienzo describe el punto de partida del flujograma. Un flujograma sólo tendrá un nodo de comienzo. En la figura B.3 a) se muestra un nodo de comienzo.

a) Nodo de comienzo de flujograma **b) Nodo de fin de flujograma**

Figura B.3/Z.140 – Nodos de comienzo y de fin

B.3.1.2.2 Nodos de fin

Los nodos de fin describen los puntos finales de un flujograma. Un flujograma puede tener varios nodos de fin o, en el caso de bucles, ningún nodo de fin. Los nodos básicos (véase B.3.1.2.3) y los nodos de referencia (véase B.3.1.2.4) que no tienen nodos sucesores serán conectados a un nodo de fin para indicar que describen la última acción de un trayecto a través de un flujograma. En la figura B.3 b) se muestra un nodo de fin.

B.3.1.2.3 Nodos básicos

Un nodo básico describe una unidad de ejecución, es decir, es ejecutado en un paso. Un nodo básico tiene un tipo y, dependiendo del tipo, puede tener una lista de atributos asociados. En la figura B.4 a) se muestra un nodo básico.

En la descripción de un nodo básico, los atributos de un nodo siguen al tipo de nodo y se colocan entre paréntesis redondos. El tipo y los atributos se utilizan para determinar la acción que se ha de realizar durante la ejecución de la construcción de lenguaje representada. Los atributos describen información que ha de ser extraída de la construcción TTCN-3 correspondiente.

Los atributos tienen valores y la semántica operacional extraerá estos valores mediante referencia al nombre de atributo. Si es necesario, se permite asignar valores explícitos en nodos básicos utilizando la asignación '='. En la figura B.4 b) se muestra un ejemplo.

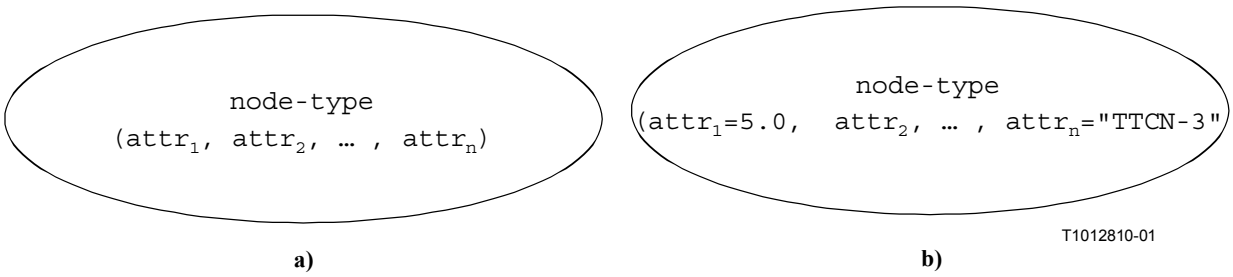


Figura B.4/Z.140 – Nodos básicos con atributos

B.3.1.2.4 Nodos de referencia

Los nodos de referencia indican segmentos de flujograma (véase B.3.1.4) que son subflujogramas. El significado de un nodo de referencia es definido por sustitución por el segmento de flujograma referenciado en el flujograma. La inscripción del nodo de referencia proporciona la referencia a un segmento de flujograma. En la figura B.5 a) se muestra un nodo de referencia.

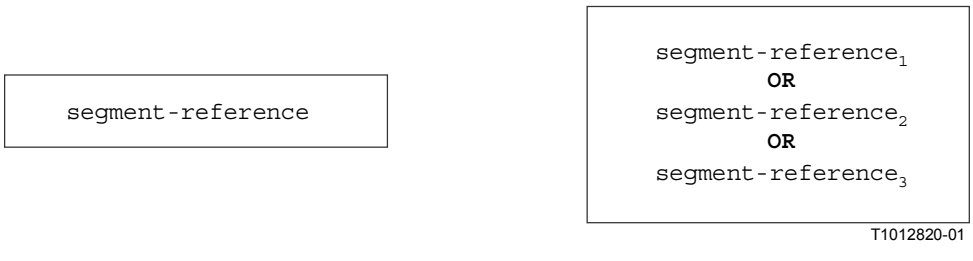


Figura B.5/Z.140 – Nodo de referencia

B.3.1.2.4.1 Combinación mediante OR de nodos de referencia

En algunos casos, varios segmentos de flujograma pueden sustituir a un nodo de referencia. Para estos casos, se puede utilizar un operador OR para hacer referencia a varios segmentos de flujograma [figura B.5 b)]. En el flujograma real que representa el control de módulo, un caso de prueba o una función, una alternativa se determina mediante la construcción representada.

B.3.1.2.4.2 Múltiples ocurrencias de modos de referencia

En algunos casos, la misma clase de nodo de referencia puede aparecer ninguna, una o más veces en un flujograma. En expresiones regulares, la posible repetición de partes de una expresión regular se describe utilizando los símbolos de operador '+' (una o más repeticiones) y '*' (ninguna o más repeticiones). Como se muestra en la figura B.6, estos operadores han sido adoptados para los flujogramas introduciendo nodos de referencia con doble marco con símbolos de operadores asociados. Una sola línea de flujo sustituirá a un nodo de referencia, en el caso de ninguna ocurrencia (utilización de un nodo de referencia con doble marco con operador '*').

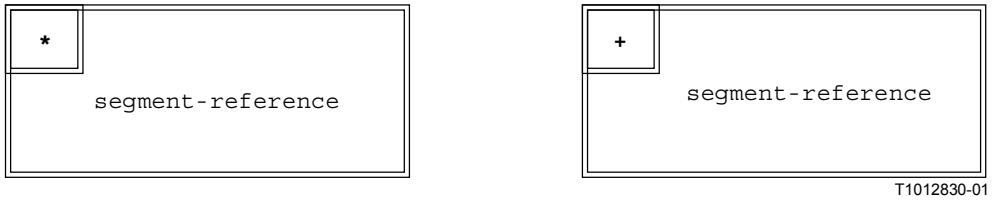


Figura B.6/Z.140 – Repetición de nodos de referencia

Se puede indicar un límite superior de posibles repeticiones de un nodo de referencia en forma de un número entero entre paréntesis que sigue al símbolo '*' o '+' en el nodo de referencia con doble marco. La referencia de segmento mostrada en la figura B.7 puede aparecer ninguna vez o hasta cinco veces.

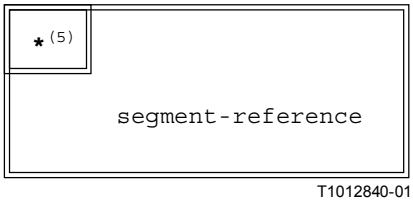
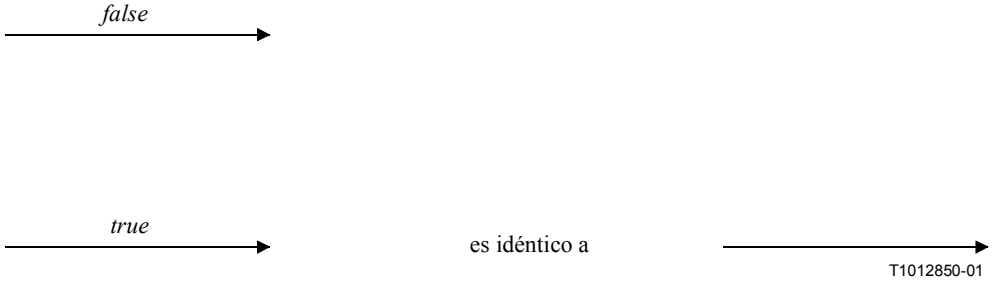


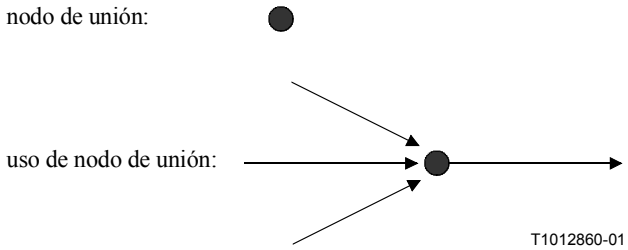
Figura B.7/Z.140 – Repetición restringida de un nodo de referencia

B.3.1.3 Líneas de flujo

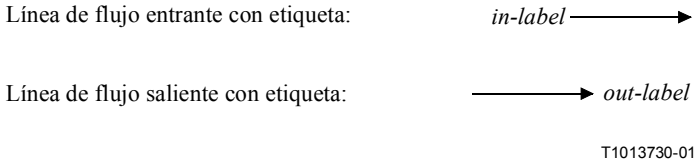
Las líneas de flujo son representadas por medio de flechas. Una línea de flujo tiene una inscripción de *true* o *false*, que indica una condición bajo la cual la línea de flujo es elegida durante la interpretación del flujograma. Como una notación abreviada, se permite omitir la inscripción *true*. A continuación se muestran ejemplos de líneas de flujo.



Para soportar la unión de varias líneas de flujo con una línea de flujo en el nivel gráfico, se introduce un nodo de unión especial. A continuación se muestra el nodo de unión y un ejemplo de su utilización.



El dibujo de líneas de flujo largas en diagramas grandes es necesario, por ejemplo, para modelar las construcciones TTCN-3 **`goto`** y **`label`**. Con este fin, se utilizan etiquetas para líneas de flujo salientes y entrantes. A continuación se muestran ejemplos.



Una línea de flujo saliente con una etiqueta está conectada con una línea de flujo entrante con una etiqueta, si las etiquetas son idénticas. Las etiquetas para las líneas de flujo entrantes serán únicas. Si hay varias líneas de flujo salientes con la misma etiqueta, esto se considera como una unión de líneas con la línea de flujo entrante que tiene una etiqueta idéntica.

B.3.1.4 Segmentos de flujograma

Los segmentos de flujograma son subflujogramas. Se hace referencia a ellos en nodos de referencia y definen el significado de ese nodo de referencia. Los segmentos de flujograma pueden incluir otros nodos de referencia.

Como se muestra en la figura B.8, los segmentos de flujograma tienen interfaces precisas que consisten en línea de flujos entrantes y salientes. Sólo hay una línea de flujo entrante sin etiqueta y una o ninguna línea de flujo saliente sin etiqueta. Además, puede haber varias líneas de flujo entrantes y salientes con etiqueta. Las líneas de flujo entrantes y salientes con etiqueta son necesarias para describir el significado de los enunciados **`goto`** de TTCN-3.

Los segmentos de flujograma se ponen en un marco y el nombre del segmento de flujograma seguirá a la palabra clave **`segment`** en la esquina izquierda superior del marco. Las líneas de flujo que describen la interfaz de segmento de flujograma atravesarán el marco del segmento de flujograma.

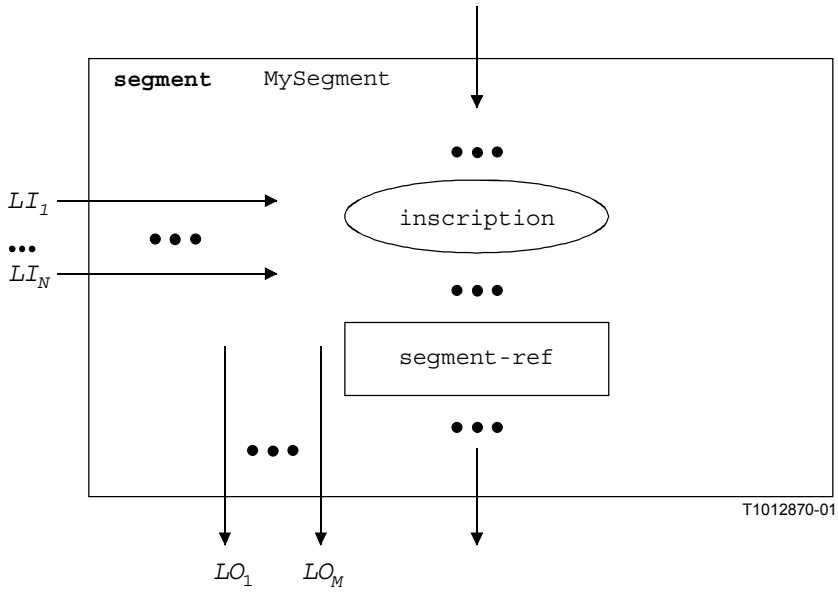
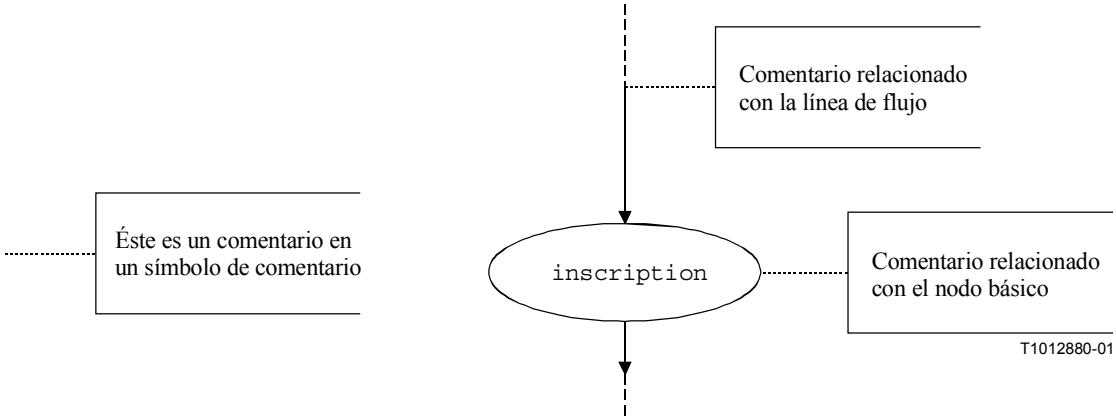


Figura B.8/Z.140 – Descripción esquemática de un segmento de flujograma

B.3.1.5 Comentarios

Para mejorar la legibilidad y la coherencia, se puede utilizar un símbolo de comentario especial para asociar comentarios con nodos de flujograma y líneas de flujo. En la figura B.9 se muestra el símbolo de comentario y su utilización.



a) Símbolo de comentario **b) Uso de símbolos de comentario**

Figura B.9/Z.140 – Representación de comentarios en un flujograma

B.3.1.6 Tratamiento de descripciones de flujograma

El procedimiento de evaluación de la semántica operacional atraviesa flujogramas que sólo consisten en nodos básicos, es decir, todos los nodos de referencia son ampliados por las correspondientes definiciones de segmento de flujograma. Para soportar esto, se requiere la función *NEXT (SIGUIENTE)*, que se define a continuación:

<actualNodeRef>.NEXT(<bool>) = <successorNodeRef>

donde:

<*actualNodeRef*> es la referencia de un nodo básico de flujograma;

<*successorNodeRef*> es la referencia de un nodo sucesor del nodo referenciado por <*actualNodeRef*>;

<*bool*> es una expresión Boolean que indica si se devuelve un sucesor *true* o *false* (véase B.3.1.3).

B.3.2 Representación mediante flujogramas de descripciones de comportamiento TTCN-3

La semántica operacional supone que se proporcionan descripciones de comportamiento TTCN-3 en forma de un conjunto de flujogramas, es decir, para cada descripción de comportamiento TTCN-3 se ha de construir un flujograma separado.

La semántica operacional interpreta las siguientes clases de definiciones TTCN-3 como descripciones de comportamiento:

a) control de módulo;

b) definiciones de casos de prueba;

c) definiciones de funciones;

d) definiciones de tipos de componentes.

El control de módulo especifica la campaña de pruebas, es decir, la orden de ejecución (posiblemente repetitiva) de los casos de prueba reales. Las definiciones de casos de prueba definen el comportamiento del MTC. Las definiciones de funciones definen el comportamiento que ha de ser ejecutado por el control de módulo o por los componentes de prueba. Se supone que las definiciones de tipos de componentes sean descripciones de comportamiento porque especifican la creación, declaración e inicialización de puertos, constantes, variables y temporizadores durante la creación de un ejemplar de componente de prueba.

B.3.2.1 Procedimiento de construcción de flujogramas

Los flujogramas presentados en las figuras B.10 y B.11 y los segmentos de flujograma presentados en B.3.6 son sólo plantillas. Incluyen símbolos *sustituidores* para la información que ha de ser proporcionada con miras a producir un flujograma o un segmento de flujograma concreto. Los sustituidores se marcan con '<' y '>'.

La construcción de una representación en flujograma de un módulo TTCN-3 se hace en tres pasos:

1) Para cada enunciado TTCN-3 en control de módulo, casos de prueba, funciones y definiciones de tipos de componentes se construye un segmento de flujograma concreto.

2) Para el control de módulo y para cada caso de prueba, función y definición de tipo de componente se construye un flujograma (con nodos de referencia).

3) En un procedimiento por pasos, todos los nodos de referencia en los flujogramas concretos son sustituidos por las correspondientes definiciones de segmento de flujograma hasta que todos los flujogramas sólo incluyen un nodo de comienzo, nodos de fin y nodos de flujograma básicos.

NOTA 1 – Los nodos de flujograma básicos describen unidades de ejecución indivisibles básicas. La semántica operacional para el comportamiento TTCN-3 se basa en la interpretación de nodos de flujograma básicos. En la cláusula B.4 se presentan los métodos de ejecución para nodos de flujogramas básicos solamente.

La sustitución de un nodo de referencia por la correspondiente definición de segmento de flujograma puede conducir a partes no conectadas en un flujograma, es decir, partes que no pueden ser alcanzadas desde el nodo de comienzo atravesando el flujograma a lo largo de las líneas de flujo. La semántica operacional pasará por alto las partes no conectadas de un flujograma.

NOTA 2 – Una parte no conectada de un flujograma es el resultado del procedimiento de sustitución mecánica. Para construir una representación en flujograma óptima hay que tomar en consideración las diferentes combinaciones de enunciados TTCN-3. Sin embargo, la finalidad de este anexo es proporcionar una semántica correcta y completa, no una representación óptima en flujograma.

B.3.2.2 Representación en flujograma de control de módulo

Esquemáticamente, la estructura sintáctica de un módulo TTCN-3 es:

```
module <identifier> (<parameter>) <module-definitions-part> control
                                                        <statement-block>
```

Para la representación del comportamiento en flujograma, sólo es pertinente la siguiente información:

```
module <identifier> <statement-block>
```

Esto es comparable a una definición de función y por consiguiente, la representación en flujograma de control de módulo es similar a la representación en flujograma de una función (véase B.3.2.4). La semántica tendrá acceso a los flujogramas que representan el control de módulo utilizando el nombre de módulo.

NOTA – El significado de la parte de definiciones de módulo está fuera del ámbito de esta semántica operacional. Los parámetros de módulo se definen como constantes globales en el tiempo de ejecución. Las referencias a parámetros de módulo tienen que ser sustituidas por sus valores concretos en un nivel sintáctico (véase B.2.3).

B.3.2.3 Representación en flujograma de casos de prueba

Esquemáticamente, la estructura sintáctica de una definición de caso de prueba TTCN-3 es:

```
testcase <identifier> (<parameter>) <testcase-interface> <statement-block>
```

La `<testcase-interface>` hace referencia a las cláusulas **`runs on`** (obligatoria) y **`system`** (facultativa) en la definición del caso de prueba. La descripción en flujograma de un caso de prueba describe el comportamiento del MTC. La información proporcionada por `<testcase-interface>` no es pertinente para el MTC y será utilizada por el enunciado **`execute`**, pero no tiene que estar representada en el flujograma de un caso de prueba. Por consiguiente, para la representación en flujograma sólo es pertinente la siguiente información:

```
testcase <identifier> (<parameter>) <statement-block>
```

Esto es comparable a una definición de función y por consiguiente, la representación en flujograma de un caso de prueba es similar a la representación en flujograma de una función (véase B.3.2.4). La semántica tendrá acceso a los flujogramas que representan casos de prueba utilizando los nombres de casos de prueba.

B.3.2.4 Representación en flujograma de funciones

Esquemáticamente, la estructura sintáctica de una función TTCN-3 es:

```
function <identifier> (<parameter>) [<function-interface>] <statement-block>
```

La `<function-interface>` facultativa hace referencia a las cláusulas **`runs on`** y **`return`** en la definición de función. La información proporcionada por la `<function-interface>` no es pertinente para la descripción de comportamiento y se utilizará para comprobaciones de semántica estática pero no tiene que estar representada en el flujograma. Por tanto, para la representación en flujograma sólo es pertinente la siguiente información:

```
function <identifier> (<parameter>) <statement-block>
```

La semántica tendrá acceso a los flujogramas que representan funciones utilizando los nombres de funciones.

El esquema de la representación de una función por medio de flujogramas se muestra en la figura B.10. El nombre de flujograma <identifier> hace referencia al nombre de la función representada (o control de módulo o caso de prueba). Los nodos del flujograma tienen comentarios asociados que describen el significado de los diferentes módulos.

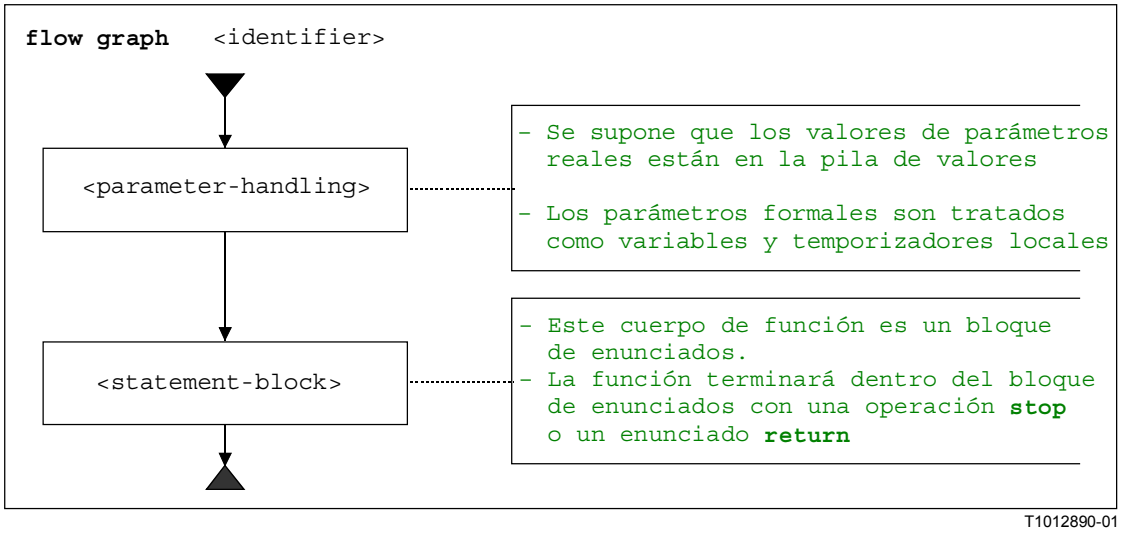


Figura B.10/Z.140 – Representación de funciones por medio de flujograma

B.3.2.5 Representación en flujograma de definiciones de tipos de componentes

Esquemáticamente, la estructura sintáctica de una definición de tipo de componente TTCN-3 es:

```
type component <identifier> <port-constant-variable-timer-declarations>
```

La semántica tendrá acceso a los flujogramas que representan tipos utilizando los nombres de tipos de componentes.

El esquema de la representación en flujograma de una definición de tipo de componente se muestra en la figura B.11. El nombre de flujograma <identifier> hace referencia al nombre del tipo de componente representado.

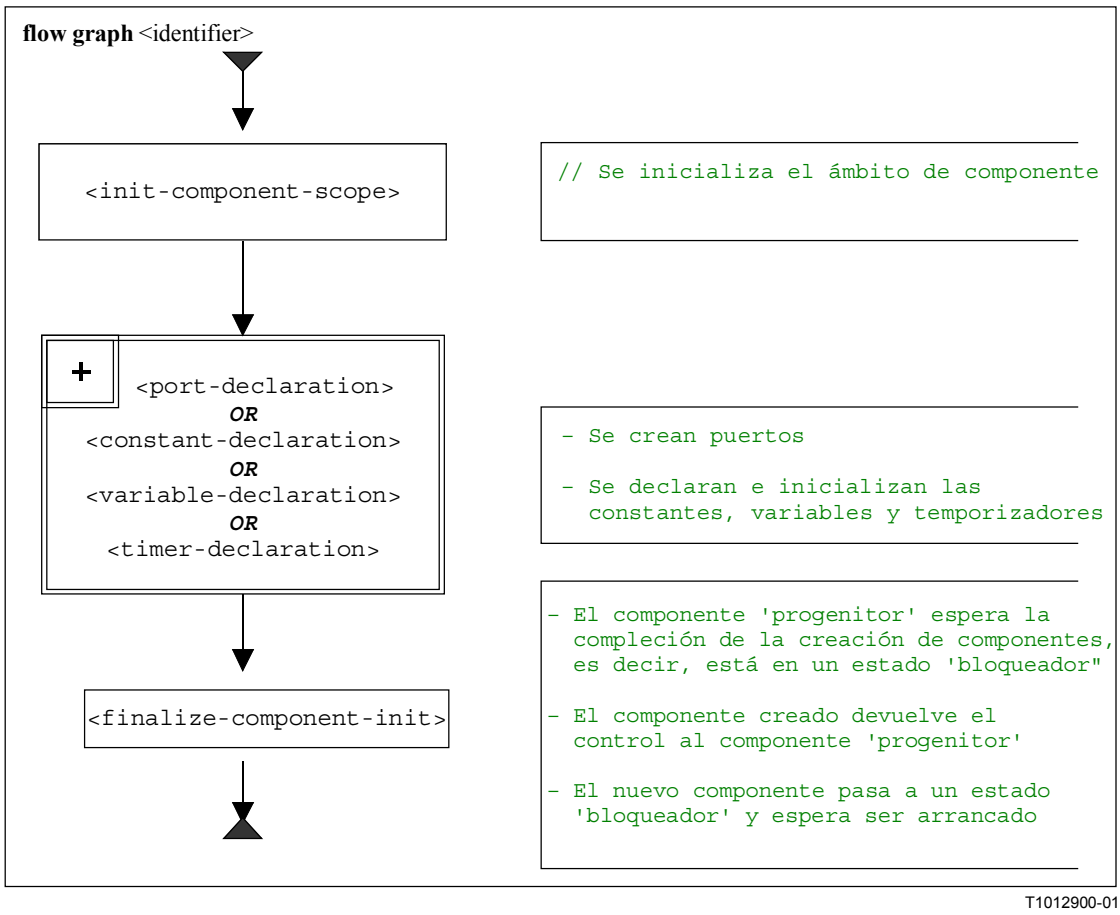


Figura B.11/Z.140 – Representación de definiciones de tipos de componente por medio de flujograma

B.3.2.6 Extracción de nodos de comienzo de flujogramas

Para extraer la referencia de nodo de comienzo de un flujograma se requiere la siguiente función:

Función GET-FLOW-GRAPH: GET-FLOW-GRAPH (<flow-graph-identifier>)

La función devuelve una referencia al nodo de comienzo de un flujograma con el nombre *<flow-graph-identifier>*, que hace referencia al nombre de módulo para el control, a nombres de caso de prueba, a nombres de funciones y a definiciones de tipos de componentes.

B.3.3 Definiciones de estados para módulos TTCN-3

Durante la interpretación de flujogramas que representan comportamiento TTCN-3, se manipulan *estados de módulo*. Un estado de módulo es un estado estructurado que consiste en varios subestados que describen los estados de los componentes y puertos de prueba. En esta cláusula se presentan los estados de módulo, estados de componentes y estados de puertos. Además, se definen las funciones para extraer información de los estados y manipularlos.

B.3.3.1 Estados de módulo

Como se muestra en la figura B.12, un estado de módulo se estructura en una *lista de estados de entidad*, una *lista de estados de puerto*, una referencia a un *MTC* y a *TC-VERDICT*. La *lista de estados de entidad* describe el estado del control de módulo y durante la ejecución de un caso de prueba, los estados de los componentes de prueba creados. La *lista de estados de puerto*, la referencia *MTC* y *TC-VERDICT* son sólo pertinentes durante la ejecución del caso de prueba. La lista de estados de puerto describe los estados de los diferentes puertos. *MTC* proporciona una

referencia al MTC, *TC-VERDICT* almacena el veredicto de prueba global real de un caso de prueba y DONE es un contador que cuenta el número de actualizaciones de *TC-VERDICT*.

NOTA 1 – El número de actualizaciones de TC-VERDICT es idéntico al número de componentes de prueba que han terminado.

El comportamiento de control de módulo (M-CONTROL en la figura B.12) se trata como un componente de prueba normal y su estado es el primer elemento en la *lista de estados de entidad* de un estado de módulo.

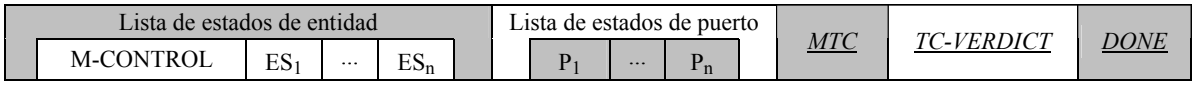


Figura B.12/Z.140 – Estructura de un estado de módulo

NOTA 2 – Los estados de puerto pueden ser considerados como parte de los estados de entidad. Sin embargo, mediante **`connect`** y **`map`** los puertos se hacen visibles a otros componentes y por tanto pueden ser tratados en el nivel más alto de un estado de módulo.

B.3.3.1.1 Acceso al estado de módulo

MTC, *SYSTEM*, *TC-VERDICT* y *DONE* son partes de un estado de módulo y se tratan como variables globales, es decir, las palabras clave *MTC* y *TC-VERDICT* pueden ser utilizadas para extraer y cambiar los valores del estado de módulo correspondiente.

NOTA 1 – Sólo existe un estado de módulo durante la interpretación de un módulo TTCN-3. Por tanto, las palabras clave *MTC* y *TC-VERDICT* pueden ser consideradas como identificadores únicos para el procedimiento de evaluación.

Para el tratamiento de la *lista de estados de entidad* y la *lista de estados de puerto*, se pueden utilizar las operaciones *append*, *delete*, *first* y *length*.

NOTA 2 – Las operaciones *append*, *delete*, *first* y *length* tienen el siguiente significado:

- <*list*>.*append*(<*item*>) anexa <*item*> como el último elemento en <*list*>;
- <*list*>.*delete*(<*item*>) suprime <*item*> de <*list*>;
- <*list*>.*first*() devuelve el primer elemento de <*list*>;
- <*list*>.*length*() devuelve la longitud de <*list*>;
- <*list*>.*next*(<*item*>) devuelve el elemento que sigue a <*item*> en la lista, o **`NULL`** si <*item*> es el último elemento en la lista.

B.3.3.2 Estados de entidad

Los estados de entidad se utilizan para describir los estados reales de control de módulo y componentes de prueba. La estructura de un estado de entidad se muestra en la figura B.13.

Figura B.13/Z.140 – Estructura de un estado de entidad

El <*identifier*> es un identificador único de una entidad, es decir, el control de módulo de componente de prueba, en el sistema de prueba. Estos identificadores únicos son creados implícitamente para el control de módulo, el **`mtc`** y el **`system`** de prueba cuando un módulo comienza la ejecución o un caso de prueba es ejecutado por medio del enunciado **`execute`**. El identificador se utiliza para identificar y direccionar entidades en el sistema de prueba, por ejemplo, en caso de operaciones **`send`** con cláusula **`to`** u operaciones **`receive`** con cláusulas **`from`**.

STATUS (*ESTADO)* describe si el módulo de control o un componente de prueba está activo o bloqueado. El control de módulo está bloqueado durante la ejecución de un caso de prueba. Los componentes de prueba pueden estar bloqueados durante la creación de otros componentes de prueba, es decir, durante la ejecución de una operación **`create`**.

CONTROL-STACK (PILA DE CONTROL) es una pila de referencias de nodos de flujogramas. El elemento de más alto nivel en *CONTROL-STACK* es el nodo de flujograma que tiene que ser interpretado a continuación. La pila tiene que modelar llamadas de funciones de manera adecuada.

Se considera que *data state* (*estado de datos*) es una lista de listas de vinculaciones de variables. La lista de estructura de listas refleja unidades de ámbito jerarquizadas debido a llamadas de funciones jerarquizadas. Cada una de las listas de vinculaciones de variables describe las vinculaciones de variables en una determinada unidad de ámbito. La entrada o salida en una unidad de ámbito corresponde con la adición o supresión de una lista de vinculaciones de variables del *estado de datos*. En B.3.3.2.2 figura una descripción más detallada de la parte de *estado de datos* de un estado de entidad.

Se considera que *timer state* (*estado de temporizador*) es una lista de listas de estados de temporizador. La lista de listas refleja las unidades de ámbito jerarquizadas debido a llamadas de funciones jerarquizadas. Cada una de las listas de estados de temporizador describe las vinculaciones de temporizador (temporizadores conocidos y sus estados) en una determinada unidad de ámbito. La entrada o salida en una unidad de ámbito corresponde a la adición o supresión de una lista de estados de temporizador del *estado de temporizador*. En B.3.3.2.3 figura una descripción más detallada de la parte de *estado de temporizador* de un estado de entidad.

VALUE-STACK (PILA DE VALORES) es una pila de valores de todos los posibles tipos que permiten el almacenamiento intermedio de resultados de operaciones finales o intermedias, funciones y enunciados. Por ejemplo, el resultado de la evaluación de una expresión o el resultado de la función **`mtc`** será introducida en *VALUE-STACK*. Además de los valores de todos los tipos de datos conocidos en un módulo, se define el valor especial **`MARK`** como parte del alfabeto de la pila. Cuando se sale de una unidad de ámbito, se utiliza **`MARK`** para despejar *VALUE-STACK*.

E-VERDICT (E-VEREDICTO) almacena el veredicto local real de un componente de prueba y es pasado por alto si un estado de entidad representa el control de módulo.

B.3.3.2.1 Acceso a estados de entidad

Las partes *STATUS* y *E-VERDICT* de un estado de entidad son tratadas como variables globales, es decir, los valores de *STATUS* y *E-VERDICT* pueden ser extraídos o cambiados utilizando la notación '`dot`' <*identifier*>.*STATUS* e <*identifier*>.*E-VERDICT*. El <*identifier*> en la notación '`dot`' hace referencia al identificador único de una entidad.

CONTROL-STACK y *VALUE-STACK* de un estado de entidad pueden ser direccionadas utilizando la notación '`dot`' <*identifier*>.*CONTROL-STACK* e <*identifier*>.*VALUE-STACK*.

CONTROL-STACK y *VALUE-STACK* pueden ser accedidas y manipuladas utilizando las operaciones de pila *push*, *pop*, *top*, *clear* y *clear-until*.

NOTA – Las operaciones de pila *push*, *pop*, *top*, *clear* y *clear-until* tienen el siguiente significado:

- <*stack*>.*push*(<*item*>) introduce <*item*> en <*stack*>;
- <*stack*>.*pop*() extrae el ítem más alto de <*stack*>;
- <*stack*>.*top*() devuelve el elemento más alto de <*stack*> o **`NULL`** si <*stack*> está vacía;
- <*stack*>.*clear*() despeja <*stack*>, es decir, extrae todos los ítems de <*stack*>;
- <*stack*>.*clear-until*(<*item*>) extrae ítems de <*stack*> hasta que <*item*> o <*stack*> está vacía.

Para crear un nuevo estado de entidad, se supone que la función *NEW-ENTITY (NUEVA ENTIDAD)* está disponible:

NEW-ENTITY (<entity-identifier>, <*flow-graph-node-reference*>)

crea un estado de nueva entidad y devuelve su referencia. Los componentes de este estado tienen los siguientes valores:

- <entity-identifier> es el identificador único;
- *STATUS* se pone a **ACTIVE**;
- <flow-graph-node-reference> es el único elemento (más alto) en *CONTROL-STACK*;
- *data state* y *timer state* son listas vacías;
- *VALUE-STACK* es una pila vacía;
- *E-VERDICT* se pone a **`none`**.

Durante la travesía por un flujograma, *CONTROL-STACK* cambia su valor a menudo de la misma manera: el elemento más alto es extraído y el nodo sucesor del nodo extraído es introducido en *CONTROL-STACK*. Esta serie de operaciones de pila se encapsula en la función *NEXT-CONTROL*:

```
<identifier>.NEXT-CONTROL(boolean <bool>) {
    FlowGraphNodeType successorNode := <identifier>.CONTROL-
    STACK.NEXT(<bool>).top();
    <identifier>.CONTROL-STACK.pop();
    <identifier>.CONTROL-STACK.push(successorNode).
}
```

B.3.3.2.2 Estado de datos y vinculación de variables

Como se muestra en la figura B.14, un *estado de datos* es una lista de listas de vinculaciones de variables. Cada lista de vinculaciones de variables define las vinculaciones de variables en una determinada unidad de ámbito. La adición de una nueva lista de vinculaciones de variables corresponde con la entrada en una nueva unidad de ámbito, por ejemplo, una función es llamada. La supresión de una lista de vinculaciones de variables corresponde con la salida de una unidad de entrada, por ejemplo, una función ejecuta un enunciado **`return`**.

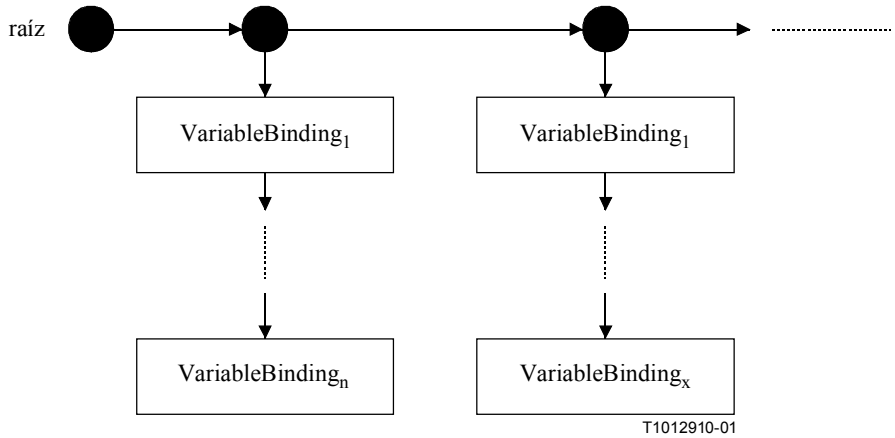


Figura B.14/Z.140 – Estructura de la parte estado de datos de un estado de entidad

En la figura B.15 se muestra la estructura de una vinculación de variable. Una variable tiene un nombre <*var-name*>, una *ubicación* y un *valor*. <*var-name*> identifica una variable en una unidad

de ámbito. La ubicación es un identificador único de la ubicación de almacenamiento del valor de la variable. La parte de valor de una vinculación de variable describe el valor real de una variable.

NOTA – Cuando se declara una variable se proporcionarán automáticamente identificadores de ubicación únicos.

<*var-name*>	**ubicación**	**valor**

Figura B.15/Z.140 – Estructura de una vinculación de variable

Se ha distinguido entre nombre y ubicación de variable para modelar llamadas de funciones y la ejecución de casos de prueba con parametrización de valores y referencias de manera apropiada:

a) Un parámetro introducido por el valor es tratado como la declaración de una nueva variable, es decir, se anexa una nueva vinculación de variable a la lista de vinculaciones de variables del ámbito de la función llamada o caso de prueba ejecutado. La nueva vinculación de variable utiliza el nombre de parámetro formal como <*var-name*>, recibe una nueva ubicación y obtiene el valor que se introduce en la función o caso de prueba.

b) Un parámetro introducido por referencia conduce también a una nueva vinculación de variable en el alcance de la función llamada o caso de prueba ejecutado. La nueva vinculación de variable utiliza también el nombre de parámetro formal como <*var-name*>, pero no recibe una nueva ubicación ni un nuevo valor. La nueva vinculación de variable obtiene una copia de *ubicación* y *valor* de la variable que se ha introducido mediante referencia.

Cuando se actualiza un valor de variable, por ejemplo, en el caso de una asignación de una variable, el nombre de variable se utiliza para identificar una ubicación y todas las vinculaciones de variables con la misma ubicación son actualizadas al mismo tiempo. De este modo, cuando se sale de la unidad de ámbito, la lista de variables perteneciente a esta unidad puede ser suprimida sin ulterior actualización. Debido al procedimiento de actualización, las variables introducidas mediante referencia automáticamente tienen el valor correcto.

B.3.3.2.3 Estados de temporizador y vinculación de temporizadores

Como se muestra en las figuras B.16 y B.17, un *estado de temporizador* y un *estado de datos* en un estado de entidad son comparables. Ambos son una lista de listas de vinculaciones y cada lista de vinculaciones define las vinculaciones válidas en un determinado ámbito. La adición de una nueva lista corresponde con la entrada en una nueva unidad de ámbito y la supresión de una lista corresponde con la salida de una unidad de ámbito.

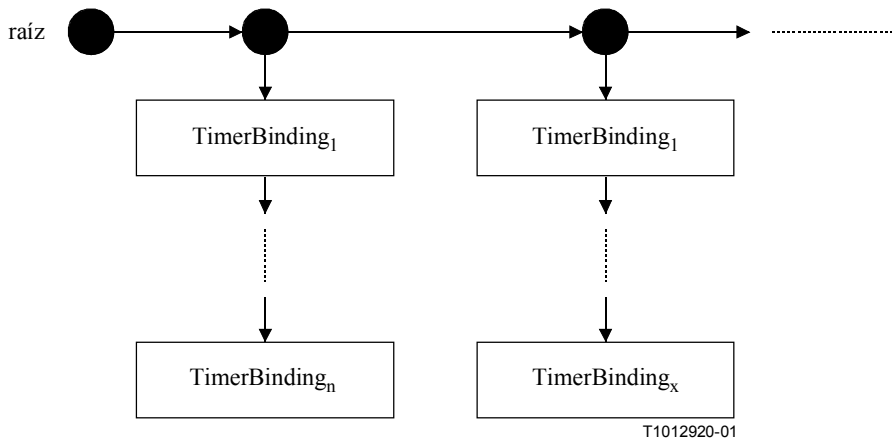


Figura B.16/Z.140 – Estructura de la parte de estado de temporizador de un estado de entidad

La estructura de una vinculación de temporizador se muestra en la figura B.17. El significado de <*timer-name*> y *location* es similar al significado de <*var-name*> y *location* para una vinculación de variables (figura B.15).

<*timer-name*>	**ubicación**	*STATUS*	*DEF-DURATION*	*ACT-DURATION*	*TIME-LEFT*

Figura B.17/Z.140 – Estructura de una vinculación de temporizador

STATUS indica si un temporizador está activo, inactivo o si ha expirado. Los valores correspondientes de *STATUS* son **`IDLE`**, **`RUNNING`** y **`TIMEOUT`**. *DEF-DURATION (DURACIÓN POR DEFECTO)* describe la duración por defecto de un temporizador. *ACT-DURATION (DURACIÓN ACTIVA)* almacena la duración real con la cual un temporizador ha sido arrancado. *TIME-LEFT (TIEMPO TRANSCURRIDO)* describe la duración real del funcionamiento de un temporizador antes de expirar.

NOTA – *DEF-DURATION* es indefinida si un temporizador es declarado sin duración por defecto. *ACT-DURATION* y *TIME-LEFT* se ponen a 0,0 si un temporizador es detenido o expira. Si un temporizador es arrancado sin duración, el valor de *DEF-DURATION* se copia en *ACT-DURATION*. Se produce un error dinámico si un temporizador es arrancado sin una duración definida.

Los temporizadores son o pueden ser introducidos en funciones mediante referencia, es decir, el mecanismo es similar al mecanismo para variables descrito en B.3.3.2.2. Esto significa que se crea una vinculación de nuevo temporizador (con el nombre de parámetro formal como <*timer-name*>) que obtiene copias de *ubicación*, *STATUS*, *DEF-DURATION*, *ACT-DURATION* y *TIME-LEFT* del temporizador que es introducido mediante referencia. Cuando se actualiza <*timer-name*>, todas las vinculaciones de temporizadores con la misma *ubicación s*on actualizadas al mismo tiempo.

B.3.3.2.4 Acceso a estados de temporizador y de datos

El *valor* de una variable puede ser extraído utilizando la notación de punto <*identifier*>.<*var-name*> donde <*identifier*> hace referencia al *identificador* único de una entidad. Para cambiar el valor de una variable, hay que utilizar la función *VAR-SET (VARIABLE FIJADA*):

<*identifier*>.*VAR-SET* (<*var-name*>, <*value*>)

fija el valor de variable <*var-name*> en el ámbito real de una entidad con el identificador único <*identifier*>. Además, el valor de todas las variables con la misma ubicación que la variable <*var-name*> se pondrá también a <*value*>.

Los valores de *STATUS*, *DEF-DURATION*, *ACT-DURATION* y *TIME-LEFT* de un temporizador <*timer-name*> pueden ser extraídos utilizando la notación de punto:

<*identifier*>.<*timer-name*>.*STATUS*;

<*identifier*>.<*timer-name*>.*DEF-DURATION*;

<*identifier*>.<*timer-name*>.*ACT-DURATION*;

<*identifier*>.<*timer-name*>.*TIME-LEFT*.

Para cambiar los valores de *STATUS*, *DEF-DURATION*, *ACT-DURATION* y *TIME-LEFT* de un temporizador <*timer-name*>, hay que utilizar una operación genérica *TIMER-SET (TEMPORIZADOR FIJADO*):

<*identifier*>.*TIMER-SET*(<*timer-name*>, *STATUS*, <*value*>)

fija el valor de *STATUS* de temporizador <*timer-name*> en el ámbito real de una entidad con el identificador único <*identifier*> para el valor <*value*>. Además, el *STATUS* de todos los temporizadores con la misma ubicación que el temporizador <*timer-name*> se pondrá también a <*value*>. La función *TIMER-SET* puede ser utilizada también para cambiar los valores de *DEF-DURATION*, *ACT-DURATION* y *TIME-LEFT.*

Para el tratamiento de variables, temporizadores y unidades de ámbito hay que definir las siguientes funciones:

a) La función *INIT-VAR*: *<identifier>.INIT-VAR (<var-name>, <value>)*

crea una vinculación de nueva variable para una variable *<var-name>* con el valor inicial *<value>* en la unidad de ámbito real de una entidad con el vinculador único *<identifier>*. El uso de la palabra clave **NONE** como *<value>* significa que se crea una variable con valor inicial indefinido.

b) La función *INIT-TIMER*: *<identifier>.INIT-TIMER (<timer-name>, <duration>)*

crea una vinculación de nuevo temporizador para un temporizador *<timer-name>* con la duración por defecto *<duration>* en la unidad de ámbito real de una entidad con el identificador único *<identifier>*. El uso de la palabra clave **NONE** como *<duration>* significa que se crea un temporizador sin duración por defecto.

c) La función *GET-VAR-LOC*: *<identifier>.GET-VAR-LOCATION (<var-name>)*

extrae la ubicación de la variable *<var-name>* poseída por una entidad con el identificador único *<identifier>*

d) La función *GET-TIMER-LOC*: *<identifier>.GET-TIMER-LOCATION (<timer-name>)*

extrae la ubicación de temporizador *<timer-name>* poseída por una entidad con el identificador único *<identifier>*

e) La función *INIT-VAR-LOC*: *<identifier>.INIT-VAR-LOC (<var-name>, <location>)*

crea una vinculación de nueva variable para una variable *<var-name>* con la ubicación *<location>* en la unidad de ámbito real de un entidad con el identificador único *<identifier>*. La variable será inicializada con el valor de otra variable con la ubicación *<location>*.

NOTA 1 – Las variables con la misma ubicación son un resultado de parametrización por referencia. Debido al tratamiento de parámetros de referencia descrito en B.3.3.2.2, todas las variables con la misma ubicación tendrán valores idénticos durante su vida.

f) La función *INIT-TIMER-LOC*: *<identifier>.INIT-TIMER-LOC (<timer-name>, <location>)*

crea una vinculación de nuevo temporizador para un temporizador *<timer-name>* con la ubicación *<location>* en la unidad de ámbito real de una entidad con el identificador único *<identifier>*. El temporizador será inicializado con los valores de *STATUS*, *DEF-DURATION*, *ACT-DURATION* y *TIME-LEFT* de otro temporizador con la ubicación *<location>*.

NOTA 2 – Los temporizadores con la misma ubicación son el resultado de parametrización por referencia. Debido al tratamiento de parámetros de referencia de temporizador descrito en B.3.3.2.3, todos los temporizadores con la misma ubicación tendrán idénticos valores para *STATUS*, *DEF-DURATION*, *ACT-DURATION* y *TIME-LEFT* durante su vida.

g) La función *INIT-VAR-SCOPE*: *<identifier>.INIT-VAR-SCOPE ()*

inicializa un nuevo ámbito de variable en el estado de datos de entidad con el identificador único *<identifier>*, es decir, se anexa una lista vacía como primera lista en la lista de listas de vinculaciones de variables.

h) La función *INIT-TIMER-SCOPE*: *<identifier>.INIT-TIMER-SCOPE ()*

inicializa un nuevo ámbito de temporizador en el estado de temporizador de entidad con el único identificador *<identifier>*, es decir, se anexa una lista vacía como primera lista en la lista de listas de vinculaciones de temporizador.

i) La función *DEL-VAR-SCOPE*: <*identifier*>.*DEL-VAR-SCOPE ()*

suprime un alcance de variables del estado de datos de entidad con el identificador único <*identifier*>, es decir, se suprime la primera lista en la lista de listas de vinculaciones de variables.

j) La función *DEL-TIMER-SCOPE*: <*identifier*>.*DEL-TIMER-SCOPE ()*

suprime un ámbito de temporizador del estado de temporizador de entidad con el identificador único <*identifier*>, es decir, se suprime la primera lista en la lista de listas de vinculaciones de temporizador.

B.3.3.3 Estados de puertos

Los estados de puertos se utilizan para describir los estados reales de puertos. En la figura B.18 se muestra la estructura de un estado de puerto. El <*port-name*> hace referencia al nombre de puerto utilizado por el componente de prueba <*owner*> que posee el puerto, para identificarlo. *STATUS* proporciona el estado real del puerto. Un puerto puede estar **`STARTED`** o **`STOPPED`**.

NOTA – Un puerto en un sistema de prueba es identificado de manera única por el componente de prueba poseedor <*owner*> y por el nombre de puerto <*port-name*> local de <*owner*>.

La parte *lista de conexiones* de un estado de puerto sigue la pista de las conexiones entre los diferentes puertos en el sistema de prueba. El mecanismo se explica en B.3.3.2.1.

La parte *cola de valores* de un estado de puerto incluye los ítems de datos recibidos en este puerto pero no consumidos aún.

<*port-name*>	<*owner*>	*STATUS*	**lista de conexiones**	**cola de valores**

Figura B.18/Z.140 – Estructura de un estado de puerto

B.3.3.3.1 Tratamiento de conexiones entre puertos

Una conexión entre dos componentes de prueba se efectúa conectando dos de sus puertos por medio de una operación **`connect`**. De este modo, un componente puede utilizar después su nombre de puerto local para direccionar la cola distante. Como se muestra en la figura B.19, *connection* se representa en los estados de ambas colas conectadas por un par de <*remote-entity*> y <*remote-port-name*>. La <*remote-entity*> es el identificador único del componente de prueba que posee el puerto distante. El <*remote-port-name*> hace referencia al nombre local utilizado por la <*remote-entity*> para direccionar la cola. TTCN-3 soporta conexiones de puertos de uno a muchos, por lo que todas las conexiones de un puerto están organizadas en una lista.

NOTA 1 – Las conexiones hechas por operaciones **`map`** son tratadas también en la lista de conexiones. La operación **`map`**: **`map`**(*PTC1*:*MyPort*, **`system`**.*PCO1*) origina a una nueva conexión (**`system`**, *PCO1*) en el estado de puerto de *MyPort* poseído por *PTC1*. El lado distante al cual está conectado *PCO1* reside dentro del SUT. Su comportamiento está fuera del ámbito de esta semántica.

NOTA 2 – La semántica operacional trata la palabra clave **`system`** como una dirección simbólica. Una conexión (**`system`**, <*port-name*>) en la lista de conexiones de un puerto indica que un puerto corresponde con el puerto <*port-name*> en la interfaz del sistema de prueba.

<*remote-entity*>	<*remote-port-name*>

Figura B.19/Z.140 – Estructura de una conexión

B.3.3.3.2 Tratamiento de estados de puerto

El tratamiento de estados de puerto es soportado por los siguientes métodos:

a) La función *NEW-PORT*: *NEW-PORT(<owner>,<port-name>)*

crea un nuevo puerto y devuelve su referencia. El nuevo puerto es poseído por <*owner*> y tiene el nombre <*port-name*> del puerto identificado por el componente de prueba <*owner*> y el nombre de puerto <*port-name*>. El estado del nuevo puerto es **STARTED** y ambos, la *lista de conexiones* y la *cola de valores* están vacías.

b) La función *GET-PORT*: *GET-PORT(<owner>, <port-name>)*

devuelve una referencia al puerto identificado por el componente de prueba <*owner*> que posee el puerto y el nombre de puerto <*port-name*>.

c) La función *GET-REMOTE-PORT*: *GET-REMOTE-PORT(<owner>, <port-name>, <remote-entity>)*

devuelve la referencia al puerto que es poseído por el componente de prueba <*remote-entity*> y está conectado a un puerto identificado por <*owner*> y <*port-name*>. Se devuelve la dirección simbólica **SYSTEM**, si el puerto distante corresponde con un puerto en la interfaz del sistema de prueba.

NOTA 1 – *GET-REMOTE-PORT* devuelve **`NULL`** si no hay puerto distante o si el puerto distante no puede ser identificado inequívocamente. Se puede utilizar el valor especial **`NONE`** como valor para el parámetro <*remote-entity*> si la entidad distante no es conocida o no se requiere, es decir, si sólo hay salida de una conexión de uno a uno para este puerto.

d) El *STATUS* de un puerto es tratado como una variable. Puede ser direccionado calificando *STATUS* con una llamada *GET-PORT*:

GET-PORT(<owner>, <port-name>).STATUS

e) La función *ADD-CON*: *ADD-CON*(<*owner*>, <*port-name*>, <*remote-entity*>, <*remote-port-name*>)

añade una conexión (<*remote-entity*>, <*remote-port-name*>) a la lista de conexiones de un puerto <*port-name*> poseído por <*owner*>.

f) La función *DEL-CON*: *DEL-CON*(<*owner*>, <*port-name*>, <*remote-entity*>, <*remote-port-name*>)

suprime una conexión (<*remote-entity*>, <*remote-port-name*>) de la lista de conexiones del puerto <*port-name*> poseído por <*owner*>.

La cola de valores en un estado de puerto puede ser accedida y manipulada utilizando las operaciones de cola conocidas *enqueue*, *dequeue*, *first* y *clear*. El uso de una función *GET-PORT* o *GET-REMOTE* hace referencia a la cola que será accedida.

NOTA 2 – Las operaciones de cola *enqueue*, *dequeue*, *first* y *clear* tienen el siguiente significado:

- <*queue*>.*enqueue*(<*item*>) pone <*item*> como el último ítem en <*queue*>;
- <*queue*>.*dequeue*() suprime el primer ítem de <*queue*>;
- <*queue*>.*first*() devuelve el primer ítem de <*queue*> o **`NULL`** si <*queue*> está vacía;
- <*queue*>.*clear*() suprime todos los elementos de <*queue*>.

B.3.3.4 Funciones generales para el tratamiento de estados de módulo

La semántica operacional supone la existencia de las siguientes funciones para el tratamiento de estados de módulo.

NOTA – Durante la interpretación de un módulo TTCN-3, sólo existe un estado de módulo. Se supone que los componentes del estado de módulo están almacenados en variables globales y no en un objeto de datos complejo. Por consiguiente, se supone que las siguientes funciones se ejecutan en variables globales y no direccionan un objeto de estado de módulo específico.

a) La función *DEL-ENTITY*: *DEL-ENTITY(<entity-identifier>)*

suprime una entidad con el identificador único <*entity-identifier*>. La supresión comprende:

- la supresión del estado de entidad de <*entity-identifier*>;
- la supresión de todos los puertos poseídos por <*entity-identifier*>;
- la supresión de todas las conexiones en las cuales participa <*entity-identifier*>.

b) La función *EXISTING*: *EXISTING(<entity-identifier>)*

devuelve *true* si existe un estado de entidad para la entidad identificada por <*entity-identifier*>. En los demás casos, devuelve *false*.

c) La función *UPDATE-REMOTE-REFERENCES*:

UPDATE-REMOTE-REFERENCES (<source-entity>, <target-entity>)

actualiza variables y temporizadores con la misma ubicación en ambas entidades. Los valores que serán utilizados para la actualización son los valores de variables y temporizadores poseídos por <*source-entity*>.

B.3.4 Mensajes, llamadas de procedimiento, respuestas y excepciones

El intercambio de información entre componentes de prueba y entre componentes de prueba y el SUT se relaciona con *mensajes*, *llamadas de procedimiento*, *respuestas a llamadas de procedimiento* y *excepciones*. A efectos de la comunicación, estos ítems tienen que ser construidos, codificados y decodificados. La codificación concreta, es decir, la correspondencia de tipos de datos TTCN-3 con bits y octetos, y la decodificación, es decir la correspondencia de bits y octetos con tipos de datos TTCN-3, está fuera del ámbito de la semántica operacional. En la presente Recomendación los *mensajes*, *procedimientos de llamada*, *respuestas a procedimientos de llamada* y *excepciones* se tratan en un nivel teórico.

B.3.4.1 Mensajes

Los mensajes se relacionan con comunicación asíncrona. Se pueden intercalar valores de todos los tipos de datos (previamente definidos y definidos por el usuario) entre las entidades que comunican. Como se muestra en la figura B.20, la semántica operacional trata un mensaje como un objeto estructurado que consiste en una parte *emisor* y una parte *valor*. La parte *emisor* identifica la entidad emisora de un mensaje y la parte *valor* define el valor del mensaje.

Figura B.20/Z.140 – Estructura de un mensaje

NOTA – La semántica operacional sólo presenta un modelo para los conceptos de TTCN-3. Si la información del *emisor* es o tiene que ser enviada y/o recibida y la manera de hacerlo dependen de la implementación del sistema de prueba; por ejemplo, en algunos casos, la información del *emisor* puede formar parte de *valor* de un mensaje y por tanto no es una parte separada de la estructura del mensaje.

B.3.4.2 Llamadas de procedimiento y respuestas

Las llamadas de procedimiento y las respuestas a llamadas de procedimiento se relacionan con llamadas síncronas. Se definen como valores de un registro con componentes que representan los parámetros. La semántica operacional trata también llamadas de procedimiento y respuestas a llamadas de procedimiento como valores en tipos estructurados. La estructura de una llamada de procedimiento y la estructura de una respuesta se presentan en las figuras B.21 y B.22.

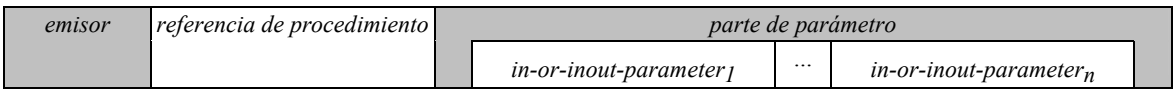


Figura B.21/Z.140 – Estructura de una llamada de procedimiento

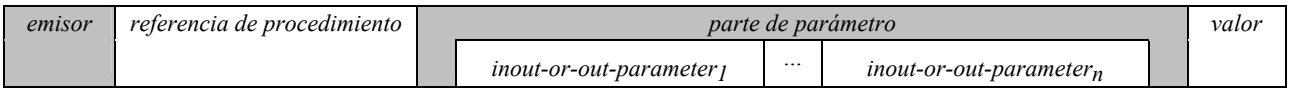


Figura B.22/Z.140 – Estructura de una respuesta a una llamada de procedimiento

La parte *emisor* y la parte *referencia de procedimiento* tienen el mismo significado en ambas figuras. La parte *emisor* hace referencia a la entidad emisora de una llamada o a la respuesta a una llamada de procedimiento. La *referencia de procedimiento* hace referencia al procedimiento al cual pertenecen la llamada y la respuesta. La *parte parámetro* de la llamada de procedimiento en la figura B.21 hace referencia a los parámetros **`in`** e **`inout`** y la *parte parámetro* de la respuesta en la figura B.22 hace referencia a los parámetros **`inout`** y **`out`** del procedimiento al cual pertenecen la llamada y la respuesta. Además, la respuesta tiene una parte *valor* para los valores devueltos en la respuesta a un procedimiento.

NOTA 1 – Como se indica en la nota anterior, la semántica operacional sólo presenta un modelo para los conceptos de TTCN-3. Si la información descrita en las figuras B.21 y B.22 es o tiene que ser enviada y/o recibida y la forma de hacerlo dependen de la realización del sistema de prueba.

NOTA 2 – Para una llamada de procedimiento, los parámetros **`out`** no son pertinentes y se omiten en la figura B.21. Para una respuesta a una llamada de procedimiento, los parámetros **`in`** no son pertinentes y se omiten en la figura B.22.

B.3.4.3 Excepciones

Las excepciones se relacionan también con comunicación síncrona. En la figura B.23 se muestra la estructura de una excepción, que consiste en tres partes. La parte *emisor* identifica al emisor de la excepción, la parte *referencia de procedimiento* hace referencia al procedimiento al cual pertenece la excepción y la parte *valor* proporciona el valor de la excepción. El tipo del valor de una excepción se define en la firma del procedimiento indicada en la parte de *referencia del procedimiento*. En general, puede ser un tipo de datos TTCN-3 definido previamente o definido por el usuario.

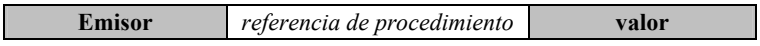


Figura B.23/Z.140 – Estructura de una excepción

B.3.4.4 Construcción de mensajes, llamadas de procedimiento, respuestas y excepciones

Las operaciones para enviar un mensaje, una llamada de procedimiento, una respuesta a una llamada de procedimiento o una excepción son **`send`**, **`call`**, **`reply`** y **`raise`**. Todas estas operaciones emisoras se construyen de la misma manera:

```
<port-name>.<sending-operation>(<send-specification>) [to <receiver>]
```

<*port-name*> y <*sending-operation*> definen el puerto y la operación utilizada para enviar un ítem. En el caso de conexiones de uno a muchos, hay que especificar una entidad <*receiver*>. El ítem que se ha de enviar se construye utilizando <*send-specification*>. La especificación de envío puede utilizar valores concretos, referencias de plantilla, valores de variables, constantes, expresiones, funciones, etc., para construir y codificar el ítem que se ha de enviar.

La semántica operacional supone que existe una función genérica *CONSTRUCT-ITEM (CONSTRUIR ÍTEM)*:

- *CONSTRUCT-ITEM(<sender>, <sending-operation>, <send-specification>)*

  devuelve un mensaje, una llamada de procedimiento, una respuesta a una llamada de procedimiento o una excepción, dependiendo de *<sending-operation>* y *<send-specification>*. Se supone también que la información *<sender>* forma parte del ítem que se ha de enviar (figuras B.20 a B.23).

B.3.4.5 Concordancia de mensajes, procedimientos de llamada, respuestas y excepciones

Las operaciones para recibir un mensaje, una llamada de procedimiento, una respuesta a una llamada de procedimiento o una excepción son **`receive`**, **`getcall`**, **`getreply`** y **`catch`**. Todas estas operaciones receptoras se construyen de la misma manera.

```
<port-name>.<receiving-operation>(<matching-part>) [from <sender>]
                                                   [<assignment-part>]
```

<port-name> y *<receiving-operation>* definen el puerto y la operación utilizados para la recepción de un ítem. En el caso de conexiones de uno a muchos, se puede utilizar una cláusula **`from`** para seleccionar una entidad emisora específica *<sender>*. El ítem que se ha de recibir tiene que cumplir las condiciones especificadas en *<matching-part>*, es decir, tiene que concordar. La *<matching-part>* puede utilizar valores concretos, referencias de plantilla, valores de variable, constante, expresiones, funciones, etc., para especificar las condiciones de concordancia.

La semántica operacional supone que existe una función genérica *MATCH-ITEM (CONCORDAR ÍTEM)*:

- *MATCH-ITEM(<item-to-check>, <matching-part>, <sender>)*

  devuelve *true* si *<item-to-check>* cumple las condiciones de *<matching-part>* y si *<item-to-check>* ha sido enviado por *<sender>*; en los demás casos, devuelve *false*.

B.3.4.6 Extracción de información de ítems recibidos

La información de mensajes recibidos, llamadas de procedimiento, respuestas a llamadas de procedimiento y excepciones puede ser extraída en la *<assignment-part> (parte de asignación)* (véase B.3.4.3) de las funciones receptoras **`receive`**, **`getcall`**, **`getreply`** y **`catch`**. La *<assignment-part>* describe cómo los parámetros de llamadas de procedimiento y respuestas, valores devueltos codificados en respuestas, mensajes, excepciones y el identificador de la entidad *<sender>* son asignados a variables.

La semántica operacional supone que existe una función genérica *RETRIEVE-INFO (EXTRAER INFORMACIÓN)*:

- *RETRIEVE-INFO(<item-received>, <assignment-part>, <receiver>)*

  todos los valores que han de ser extraídos de acuerdo con la *<assignment-part>* son extraídos y asignados a las variables enumeradas en la parte de asignación. Las asignaciones se hacen por medio de la operación VAR-SET, es decir, las variables con la misma ubicación son actualizadas al mismo tiempo.

B.3.5 Registros de llamada para funciones y casos de prueba

Las funciones y casos de prueba son llamadas (o ejecutadas) por su nombre y una lista de parámetros reales. Los parámetros reales proporcionan referencias para parámetros de referencias y valores concretos para el parámetro de valor definido por los parámetros formales en la definición de función o de caso de prueba. La semántica operacional trata llamadas de funciones y llamadas de casos de prueba utilizando *registros de llamada* como se muestra en la figura B.24. El valor de *BEHAVIOUR-ID (ID DE COMPORTAMIENTO)* es el nombre de una función o caso de prueba, los

parámetros de valor proporcionan valores concretos <*parId*$_1$> … <*parId*$_n$> para los parámetros formales <*parId*$_1$> … <*parId*$_n$>. Los parámetros de referencia proporcionan referencias a ubicaciones de variables y temporizadores existentes. Antes de poder ejecutar una función o caso de prueba, hay que construir un registro de llamada apropiado.

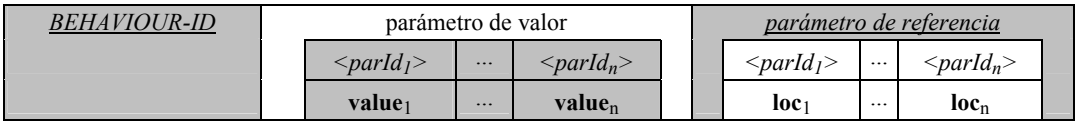


Figura B.24/Z.140 – Estructura de un registro de llamada

B.3.5.1 Tratamiento de registros de llamada

El nombre de función o de caso de prueba y los valores de parámetros reales pueden ser extraídos utilizando la notación de punto, por ejemplo, <*myRecord*>.<*parId*$_n$> o <*myRecord*>.*BEHAVIOUR-ID* donde <*myRecord*> es un puntero a un registro de llamada.

Para la construcción de una llamada se supone que la función *NEW-CALL-RECORD (NUEVO REGISTRO DE LLAMADA*) esté disponible:

- *NEW-CALL-RECORD(<behaviour-name>)*

  crea un nuevo registro de llamada para la función o caso de prueba <*behaviour-name*> y devuelve un puntero al nuevo registro. Los campos de parámetros del nuevo registro de llamada tienen valores indefinidos.

- *<call-record>.INIT-CALL-RECORD()*

  crea variables y temporizadores para el tratamiento de parámetros de valor y de referencia en el ámbito real de una función o caso de prueba. Las variables para el tratamiento de parámetros de valores son inicializadas con los correspondientes valores proporcionados en el registro de llamada. Las variables y temporizadores para el tratamiento de parámetros de referencia obtienen la ubicación proporcionada. Además, obtienen un valor de una variable o temporizador existente en otra unidad de ámbito del componente en la cual se creó el registro de llamada.

B.3.6 Procedimiento de evaluación para un módulo TTCN-3

B.3.6.1 Fases de evaluación

El procedimiento de evaluación para un módulo TTCN-3 se estructura en:

1) fase de inicialización;
2) fase de actualización;
3) fase de selección; y
4) fase de ejecución.

Las fases 2), 3) y 4) se repiten hasta que termina el control de módulo. El procedimiento de evaluación se describe por medio de una mezcla de texto informal, pseudocódigo y funciones introducidas en las cláusula previas.

B.3.6.1.1 Fase I: Inicialización

La fase de inicialización incluye las siguientes acciones:

a) **Declaración e inicialización de variables:**

   – *INIT-FLOW-GRAPHS*(); // Inicialización de tratamiento de flujograma.
   // *INIT-FLOW- GRAPHS* se explica en B.3.5.1.

– *Entity* := **NULL**; *Entity* se utilizará para hacer referencia a un estado // de entidad. Un estado representa un control de // módulo o un componente de prueba.

– *AllEntities*:= **NULL**; // *AllEntities* será una lista de estados de entidad

– *AllPorts*:= **NULL**; // *AllPorts* será una lista de estados de puerto

– *MTC*:= **NULL**; // *MTC* hará referencia al MTC cuando se ejecuta // un caso de prueba

– *TC-VERDICT*:= **none**; // *TC-VERDICT* almacenará el veredicto de caso de // prueba real cuando se ejecuta un caso de prueba

– *DONE*:= 0; // Durante la ejecución de un caso de prueba *DONE* // cuenta el número de componentes de prueba que // han terminado.

NOTA – Las variables globales *AllEntities*, *AllPorts*, *MTC*, *TC-VERDICT* y *DONE* forman el estado de módulo que es manipulado durante la interpretación de un módulo TTCN-3.

b) **Creación e inicialización de control de módulo**

– *Entity:= NEW-ENTITY (GET-UNIQUE-ID(),GET-FLOW-GRAPH (<moduleId>));*
// Un nuevo estado de entidad es creado e // inicializado con el nodo de comienzo del // flujograma que representa el // comportamiento del control de módulo // con el nombre // *<moduleId>*. *GET-* // *UNIQUE-ID* se explica en B.3.5.1.

– *Entity.INIT-VAR-SCOPE*(); // Nuevo ámbito de variable

– *Entity.INIT-TIMER-SCOPE*(); // Nuevo ámbito de temporizador

– *Entity.VALUE-STACK.push*(**MARK**); // Una marca es introducida en la pila de // valores

– *AllEntities.append*(*Entity*); // La nueva entidad es introducida en el // estadode módulo.

B.3.6.1.2 Fase II: Actualización

La fase de actualización se relaciona con todas las acciones que están fuera del ámbito de la semántica operacional, pero que influyen en la interpretación de un módulo TTCN-3. La fase de actualización comprende las siguientes acciones:

a) **Progresión temporal**: Todos los temporizadores en funcionamiento son actualizados, es decir, los valores *TIME-LEFT* de temporizadores en funcionamiento son (posiblemente) disminuidos, y si debido a la actualización un temporizador expira, se actualizan las correspondientes vinculaciones de temporizador, es decir, *TIME-LEFT* se pone a 0,0 y *STATUS* se pone a **`TIMEOUT`**;

b) **Comportamiento del SUT**: Los mensajes, llamadas de procedimientos distantes, respuestas a llamadas de procedimientos distantes y excepciones (posiblemente) recibidas del SUT se introducen en las colas de puerto en las cuales se efectuarán las recepciones correspondientes.

NOTA – Esta semántica operacional no establece hipótesis sobre la progresión temporal y el comportamiento del SUT.

B.3.6.1.3 Fase III: Selección

La fase de selección consiste en las dos acciones siguientes:

a) **Selección**: Seleccionar una entidad no bloqueada, es decir una entidad que tiene el valor *STATUS* **ACTIVE**;

b) **Almacenamiento**: Almacenar el identificador de la entidad seleccionada en la variable global *Entity*.

B.3.6.1.4 Fase IV: Ejecución

La fase de ejecución consiste en las dos acciones siguientes:

a) **Paso de ejecución de la entidad seleccionada**: Ejecutar el nodo de flujograma en la cima de la *CONTROL-STACK* de *Entity*;

b) **Comprobación de criterio de terminación**: Detener la ejecución si el control de módulo ha terminado, es decir, la lista de estados de entidad está vacía; en los demás casos, continuar con la fase II.

NOTA – El paso de ejecución de la entidad seleccionada puede ser considerado como una llamada de procedimiento. La comprobación del criterio de terminación se hace cuando termina el paso de ejecución, es decir, se devuelve el control.

B.3.6.2 Funciones globales

El procedimiento de evaluación utiliza las funciones globales *INIT-FLOW-GRAPHS* y *GET-UNIQUE-ID*:

a) se supone que *INIT-FLOW-GRAPHS (INICIAR FLUJOGRAMA)* es la función que inicializa el tratamiento de flujogramas. El tratamiento puede incluir la creación de los flujogramas y el tratamiento de los punteros a los flujogramas y nodos de flujogramas.

b) se supone que *GET-UNIQUE-ID (OBTENER ID ÚNICO)* es una función que devuelve un identificador único cada vez que es llamada. El identificador único puede ser implementado en forma de una variable de contador que es aumentada y devuelta cada vez que se invoca *GET-UNIQUE-ID*.

El pseudocódigo utiliza las siguientes cláusulas para describir la ejecución de nodos de flujograma que utilizan las funciones *CONTINUE-COMPONENT*, *RETURN*, `***DYNAMIC-ERROR***`:

c) *CONTINUE-COMPONENT (CONTINUAR COMPONENTE)* hace que el componente de prueba real continúe su ejecución con el nodo que está en la cima de la pila de control, es decir, el control no es devuelto al procedimiento de evaluación de módulo descrito en esta subcláusula.

d) *RETURN (DEVOLVER)* devuelve el control al procedimiento de evaluación de módulo descrito en esta cláusula. *RETURN* es la última acción del '*paso de ejecución de la entidad seleccionada*' de la fase de *ejecución*.

e) `***DYNAMIC-ERROR***` hace referencia a la ocurrencia de un error dinámico. El procedimiento de tratamiento de errores en sí mismo está fuera del ámbito de la semántica operacional. Si se produce un error dinámico, se considera que todo el comportamiento siguiente del módulo es indefinido.

NOTA – La ocurrencia de un error dinámico se relaciona con el comportamiento de prueba. Un error dinámico especificado por la semántica operacional indica un problema en la utilización de TTCN-3, por ejemplo, utilización errónea o condiciones de competencia.

B.3.7 Definiciones de segmentos de flujograma para construcciones TTCN-3

La semántica operacional representa el comportamiento TTCN-3 en forma de flujogramas. El algoritmo de construcción para los flujogramas que representan comportamiento se describe en B.3.2.1. Se basa en plantillas para flujogramas y segmentos de flujogramas que se han de utilizar

para construir flujogramas concretos para el control de módulo, casos de prueba, funciones y definiciones de tipo de componentes definidos en un módulo TTCN-3. Las definiciones de las plantillas para los segmentos de flujograma se explican en esta cláusula y se presentan en orden alfabético (inglés) y no en un orden lógico.

Las definiciones de segmentos de flujograma se proporcionan en forma de figuras. Los nodos de flujogramas se presentan en el lado izquierdo de la figura y los comentarios asociados con los nodos y líneas de flujo se muestran en el lado derecho. Se presentan comentarios descriptivos para los nodos de referencia y los comentarios en forma de pseudocódigo se asocian con nodos básicos. El pseudocódigo describe cómo se interpreta un nodo básico, es decir, cambios del estado de módulo. Utiliza las funciones definidas en las partes previas de B.3 y las variables globales declaradas e inicializadas en el procedimiento de evaluación para módulos TTCN-3 (en B.3.6). En esta cláusula se da una visión general de todas las funciones y palabras clave utilizadas por el pseudocódigo.

B.3.7.1 Enunciado Alt

La representación en flujograma del enunciado **`alt`** en la figura B.25 distingue entre enunciados **`alt`** que tienen una rama **`else`** y enunciados **`alt`** que no tienen rama **`else.`**

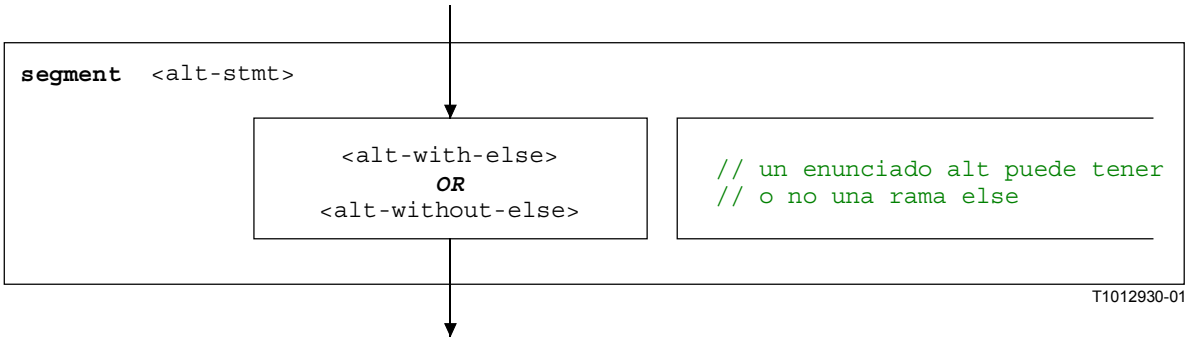


Figura B.25/Z.140 – Segmento de flujograma <alt-stmt >

Los segmentos de flujograma `<alt-with-else>` y `<alt-without-else>` se muestran en las figuras B.26 y B.27. La rama **`else`** es un bloque de enunciados que no necesita más explicaciones. Sin embargo, ambos segmentos de flujogramas son muy similares, con la diferencia de que la rama **`else`** proporciona una salida definida para el enunciado **`alt`**, mientras que un enunciado **`alt`** sin rama **`else`** puede establecer un bucle.

Ambos segmentos de flujograma tienen un nodo de entrada y además una línea de flujo entrante, una línea de flujo adicional con una etiqueta <altId>. Ésta es una etiqueta simbólica para el enunciado **`alt`** e identifica el objetivo de enunciados **`goto alt`** y define también el bucle en el segmento de flujograma `<alt-without-else>`. Ambos segmentos de flujograma tienen también un punto de salida definido por medio de la etiqueta <altIdExit> y el nodo `alt-exit`.

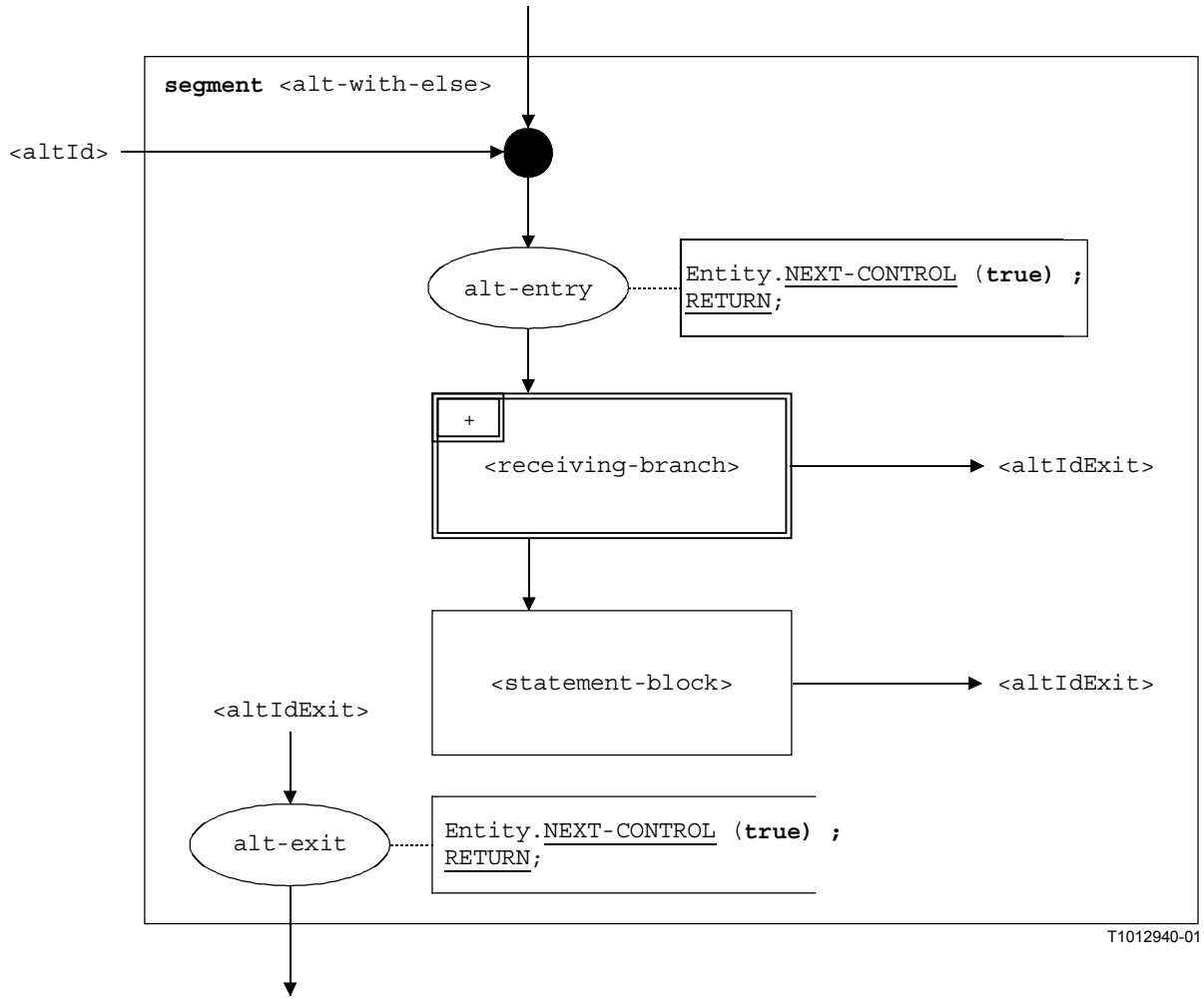


Figura B.26/Z.140 – Segmento de flujograma <alt-with-else>

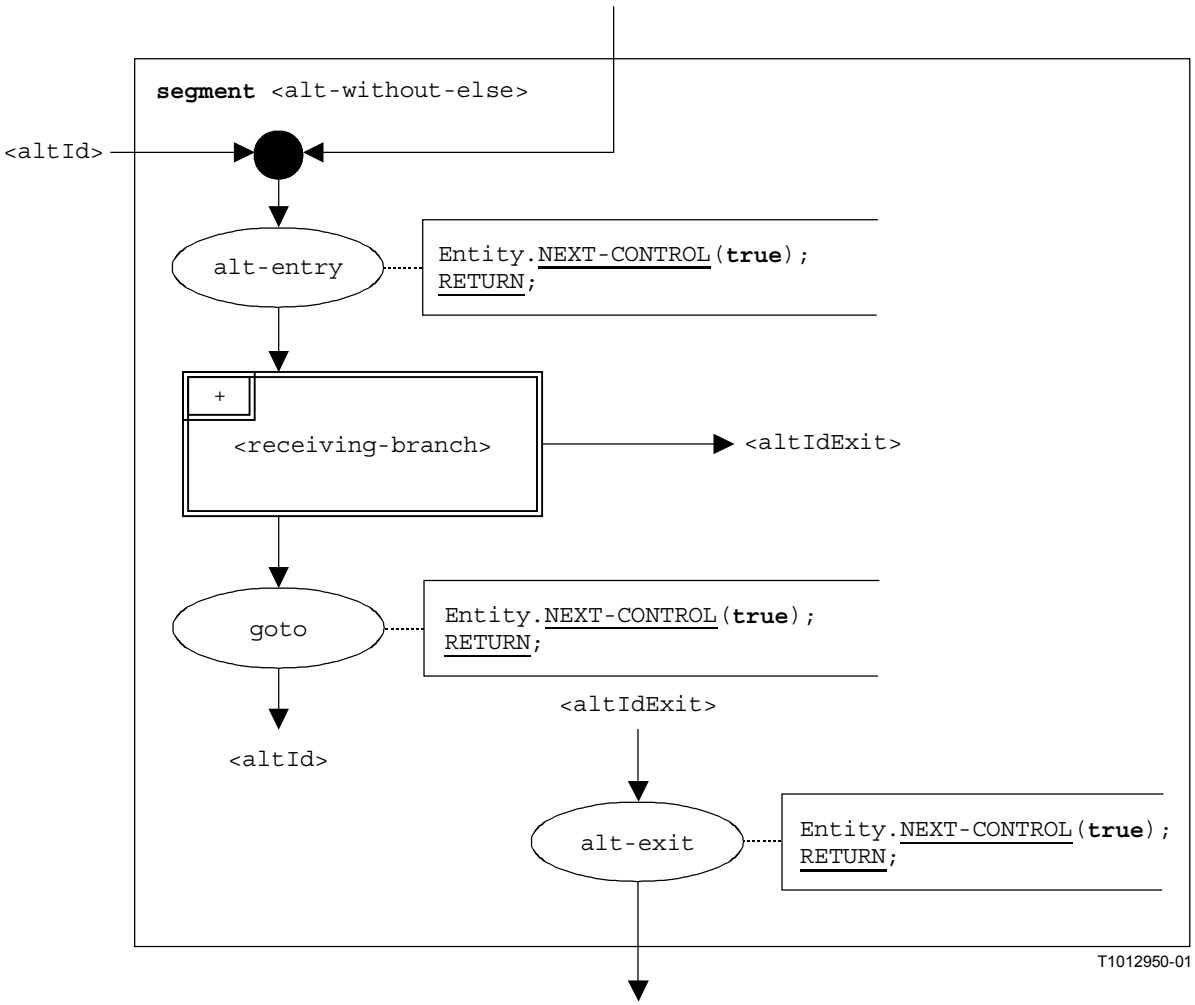


Figura B.27/Z.140 – Segmento de flujograma <alt-without-else>

B.3.7.1.1 Segmento de flujograma <receiving-branch>

La ejecución del segmento de flujograma `<receiving-branch>` se muestra en la figura B.28.

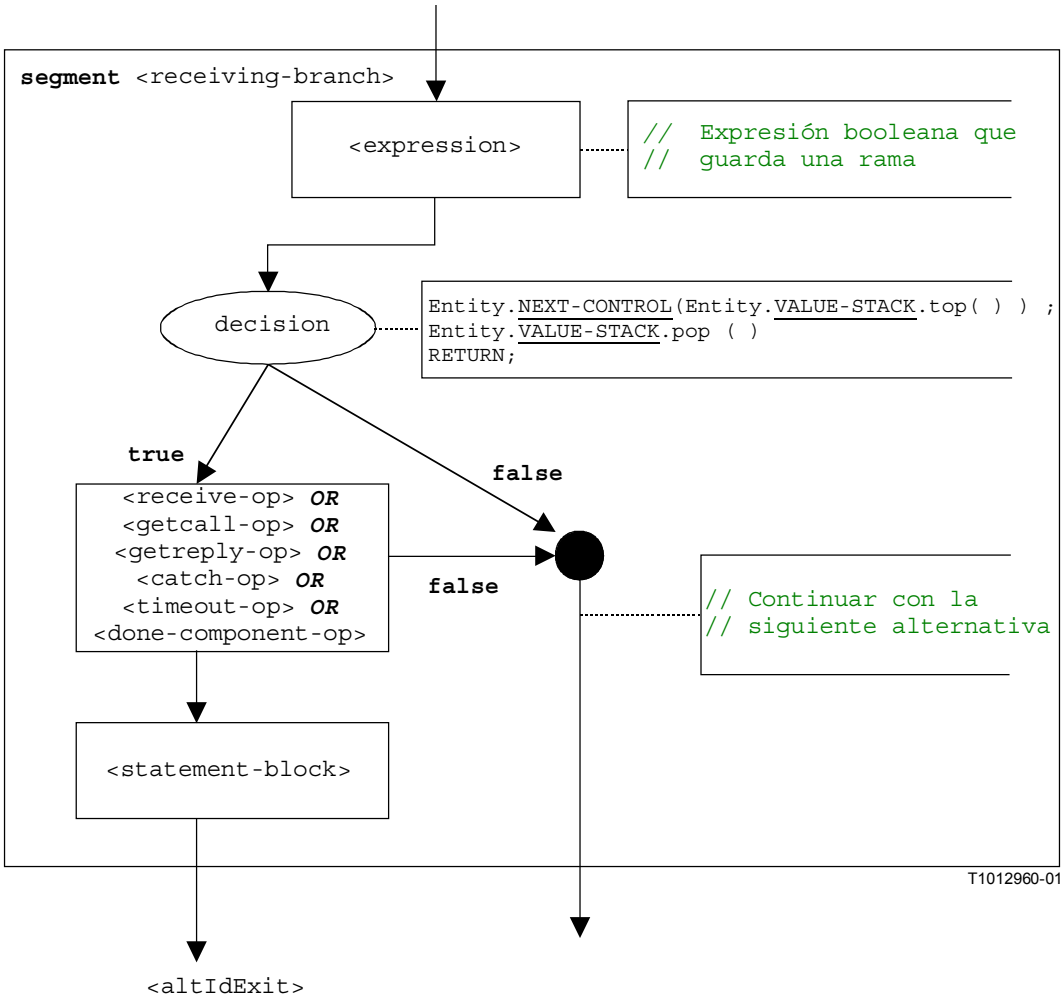


Figura B.28/Z.140 – Segmento de flujograma <receiving-branch>

B.3.7.2 Enunciado Assignment

La estructura sintáctica del enunciado **`assignment`** es:

```
<varId> := <expression>
```

El valor de la expresión <expression> se asigna a la variable <varId>. La ejecución de un enunciado de asignación es definido por el segmento de flujograma <assignment-stmt> en la figura B.29.

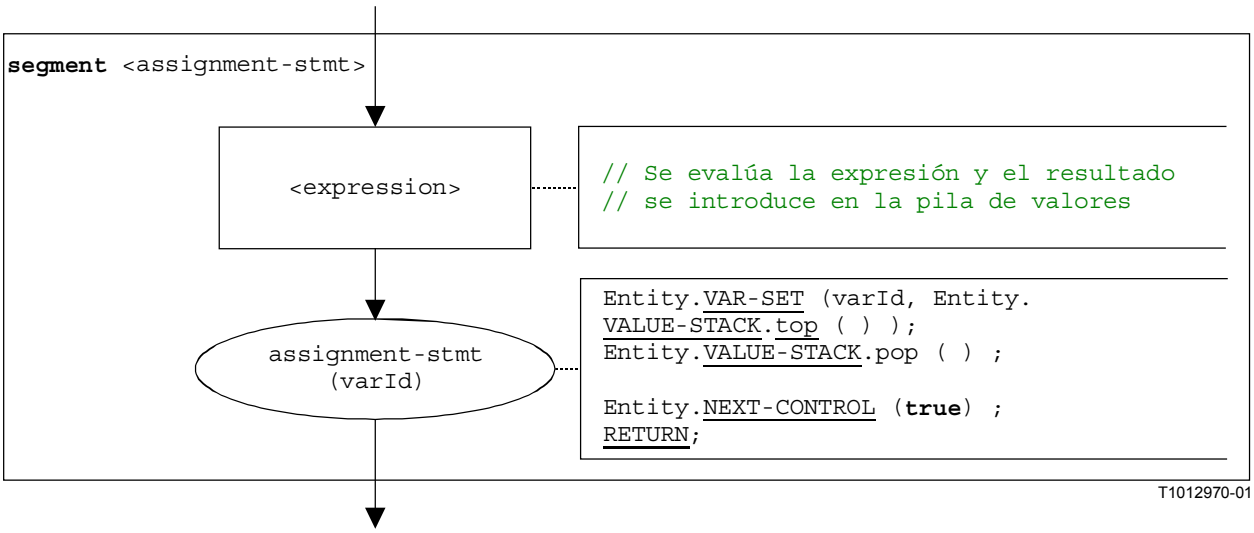


Figura B.29/Z.140 – Segmento de flujograma <assignment-stmt>

B.3.7.3 Operación Call

La estructura sintáctica de la operación **call** es:

```
<portId>.call (<callSpec> [<blocking-info>]) [to <component_expression>]
                                                    [<call-reception-part>]
```

La <blocking-info> facultativa consiste en la palabra clave **nonblocking** o en una duración de una excepción de temporización. La <component_expression> facultativa en la cláusula **to** hace referencia a la entidad receptora. Se puede proporcionar en forma de un valor de variable o el valor devuelto de una función. La <call-reception-part> facultativa indica las recepciones alternativas en caso de una operación **call** bloqueadora.

La semántica operacional distingue entre operaciones **call** *blocking* y *non-blocking*. Una operación **call** no es bloqueadora si no se prevén respuestas o si se utiliza la palabra clave **nonblocking**. Una operación **call** tiene una <call-reception-part>.

El segmento de flujograma <call-op> en la figura B.30 define la ejecución de una operación **call** y refleja la distinción entre llamadas bloqueadoras y no bloqueadoras.

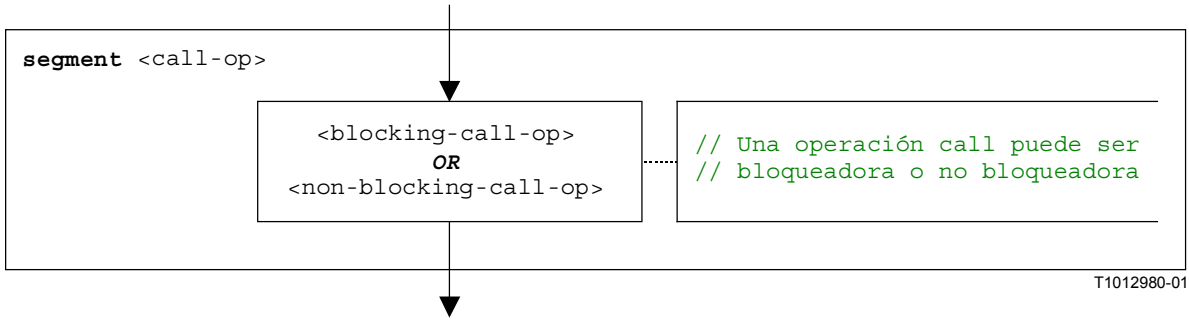


Figura B.30/Z.140 – Segmento de flujograma <call-op>

Para las operaciones **call** bloqueadoras y no bloqueadoras, se puede especificar una entidad receptora en forma de una expresión. Las posibilidades se muestran en las figuras B.31 y B.32.

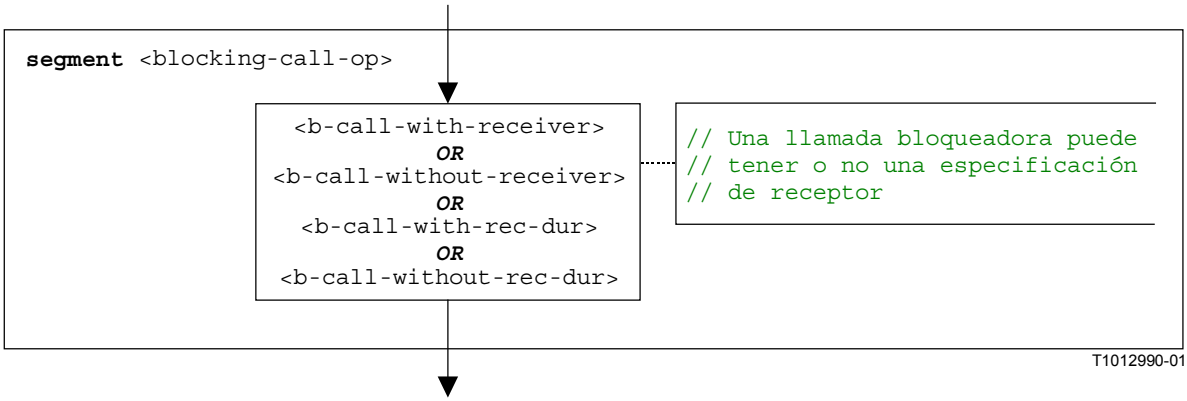


Figura B.31/Z.140 – Segmento de flujograma <blocking-call-op>

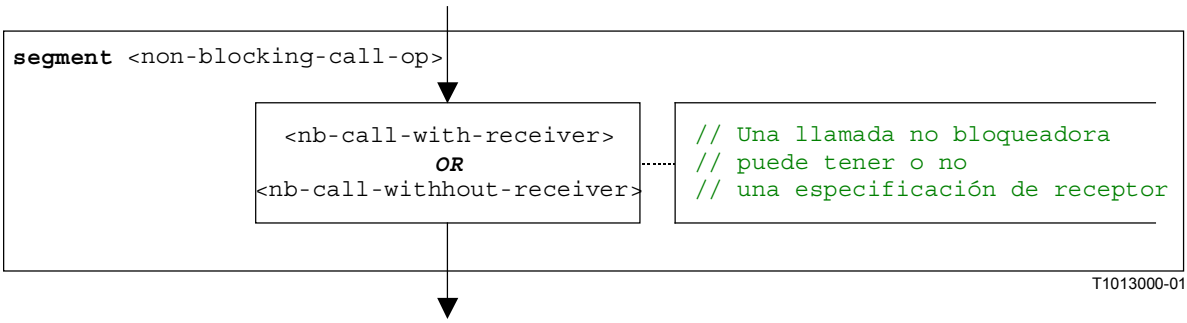


Figura B.32/Z.140 – Segmento de flujograma <non-blocking-call-op>

B.3.7.3.1 Segmento de flujograma <nb-call-with-receiver>

El segmento de flujograma `<nb-call-with-receiver>` de la figura B.33 define la ejecución de una operación **call** no bloqueadora, cuando el receptor se especifica en forma de una expresión.

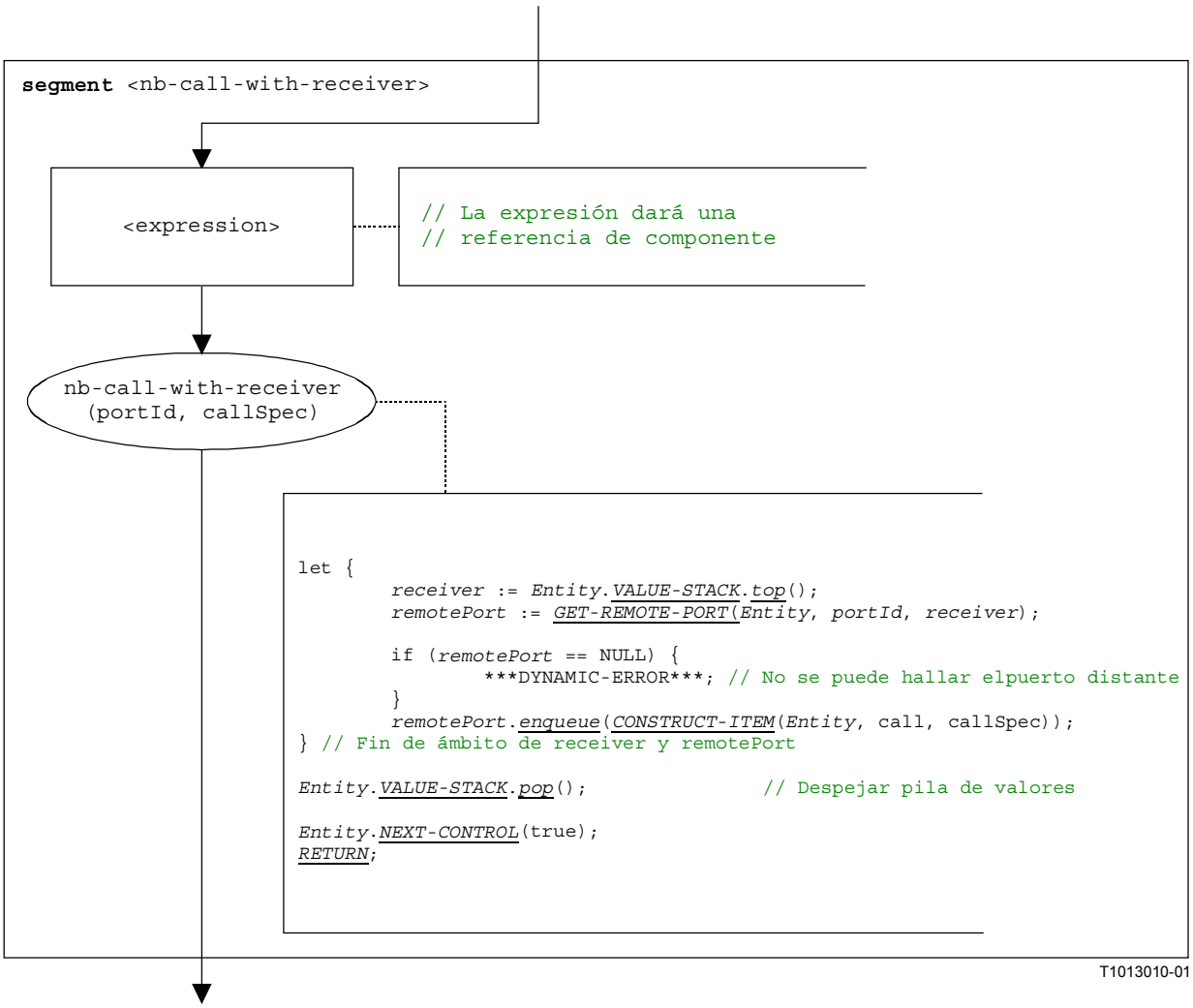


Figura B.33/Z.140 – Segmento de flujograma <nb-call-with-receiver>

B.3.7.3.2 Segmento de flujograma <nb-call-without-receiver>

El segmento de flujograma `<nb-call-without-receiver>` de la figura B.34 define la ejecución de una operación **`call`** no bloqueadora sin una cláusula **`to`**.

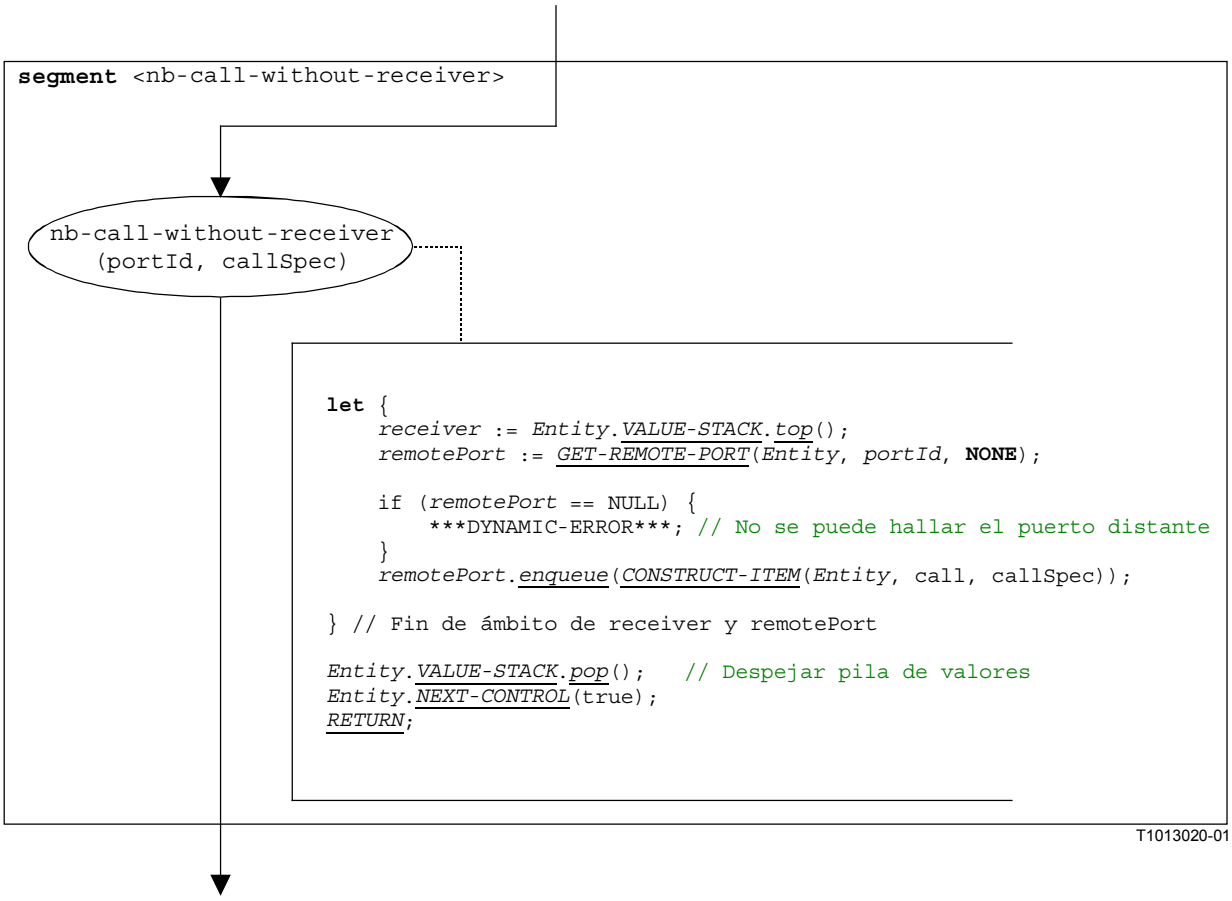


Figura B.34/Z.140 – Segmento de flujograma <nb-call-without-receiver>

B.3.7.3.3 Segmento de flujograma <b-call-with-receiver>

Las llamadas bloqueadoras son modeladas por una llamada no bloqueadora seguida por un enunciado **alt**. El segmento de flujograma <b-call-with-receiver> describe la ejecución de una llamada bloqueadora, sin una duración como guarda de temporizador, pero con una descripción de receptor para la llamada. El segmento de flujograma se muestra en la figura B.35.

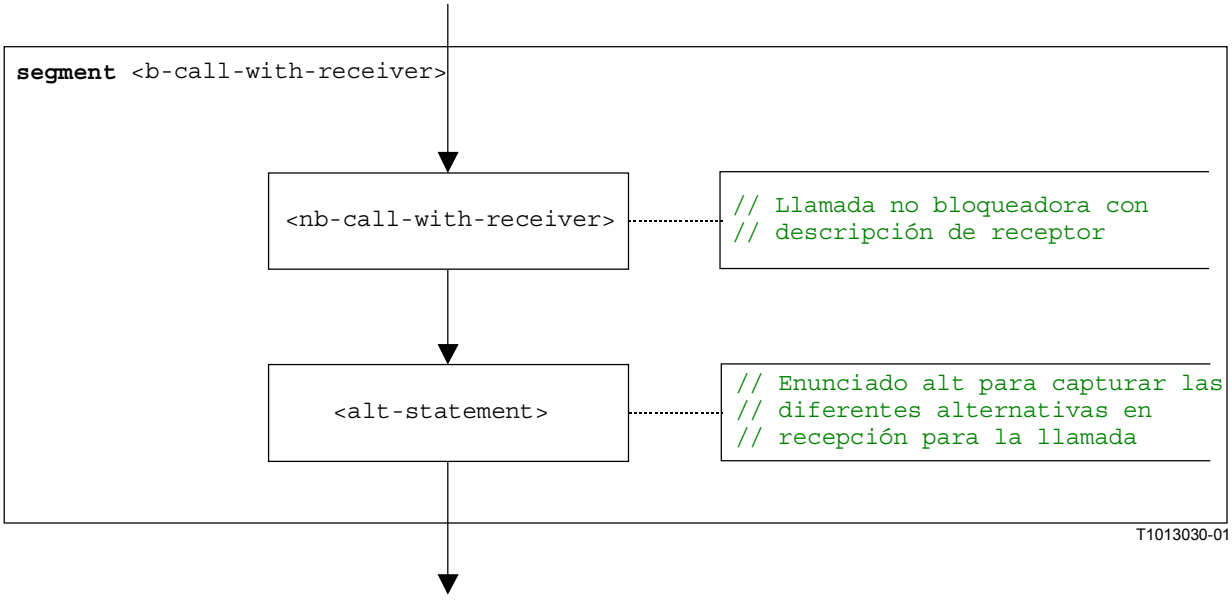


Figura B.35/Z.140 – Segmento de flujograma <b-call-with-receiver>

B.3.7.3.4 Segmento de flujograma <b-call-without-receiver>

El segmento de flujograma `<b-call-without-receiver>` describe la ejecución de una llamada bloqueadora, sin una duración como guarda de temporizador y sin una especificación de receptor para la llamada. El segmento de flujograma se muestra en la figura B.36.

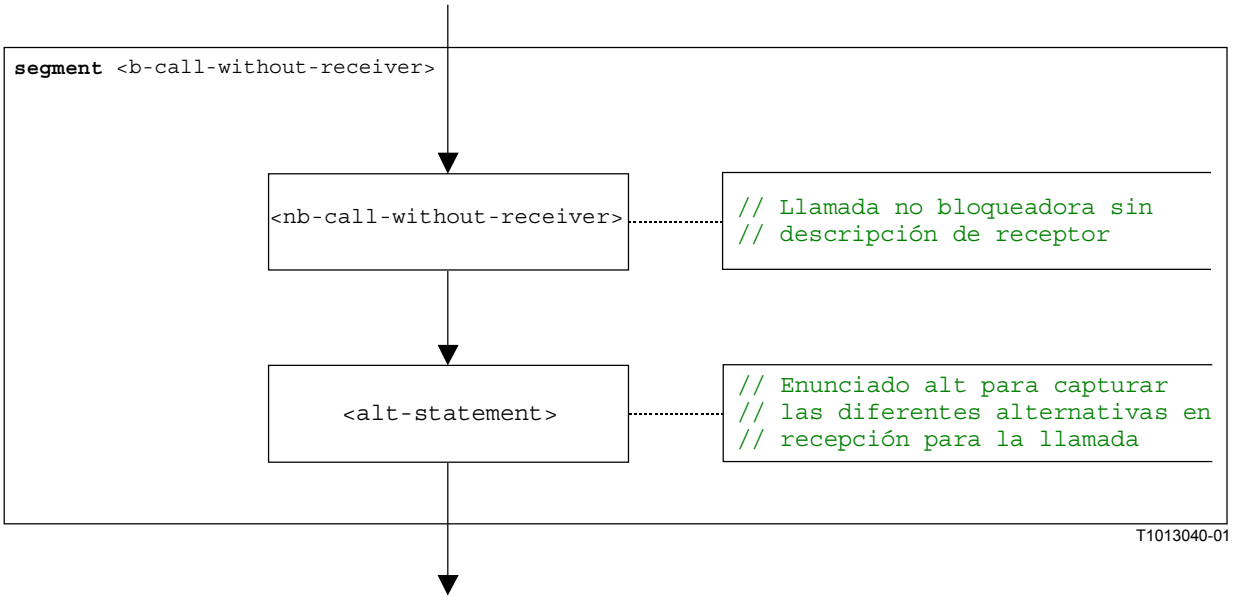


Figura B.36/Z.140 – Segmento de flujograma <b-call-without-receiver>

B.3.7.3.5 Segmento de flujograma <b-call-with-rec-dur>

Las llamadas bloqueadoras guardadas por temporizadores son modeladas por una llamada no bloqueadora seguida por un enunciado **`alt`**. Para la duración, se arranca un temporizador de sistema especial SYS-TI. La rama de temporización de captura en el enunciado **`alt`** hace referencia al temporizador de sistema. El segmento de flujograma `<b-call-with-rec-dur>` describe la ejecución de una llamada bloqueadora con una duración como guarda de temporizador y una descripción de receptor para la llamada. El segmento de flujograma se muestra en la figura B.37.

NOTA – El tratamiento del temporizador de sistema se trata sólo de manera informal. La implementación es una patente del equipo de prueba.

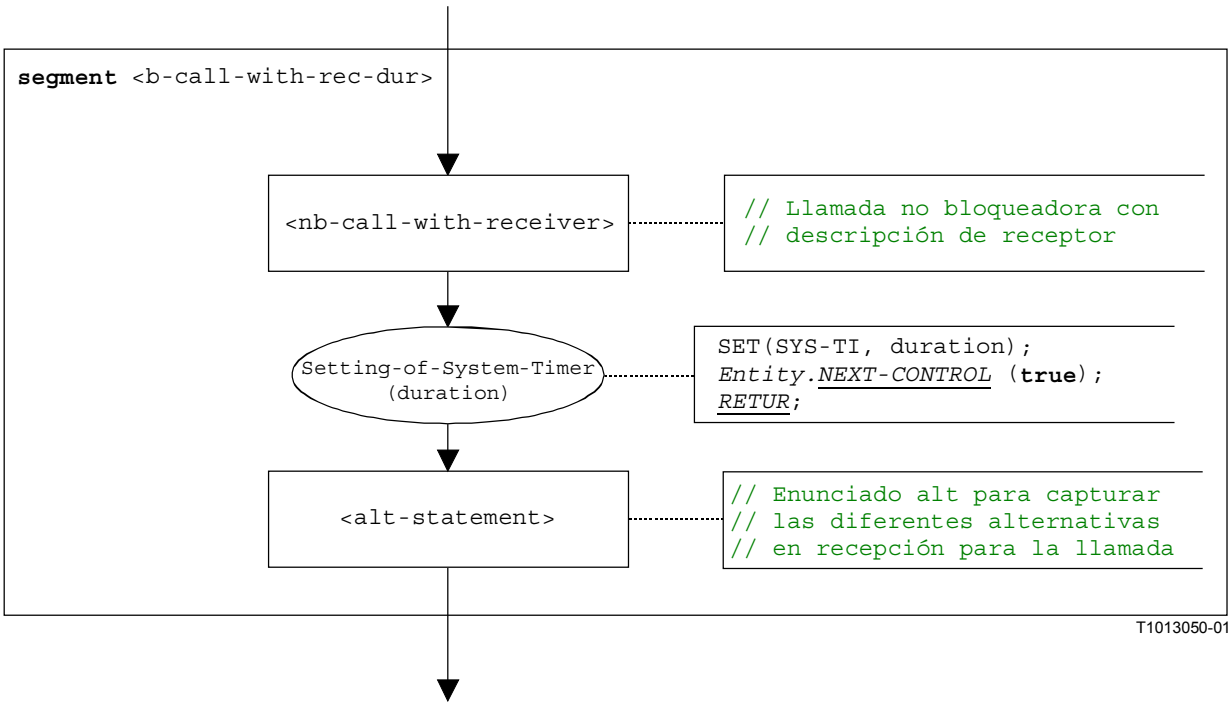


Figura B.37/Z.140 – Segmento de flujograma <b-call-with-rec-dur>

B.3.7.3.6 Segmento de flujograma <b-call-without-rec-dur>

El segmento de flujograma `<b-call-without-rec-dur>` describe la ejecución de una llamada bloqueadora con una duración como guarda de temporizador y sin una descripción de receptor para la llamada. El segmento de flujograma se muestra en la figura B.38.

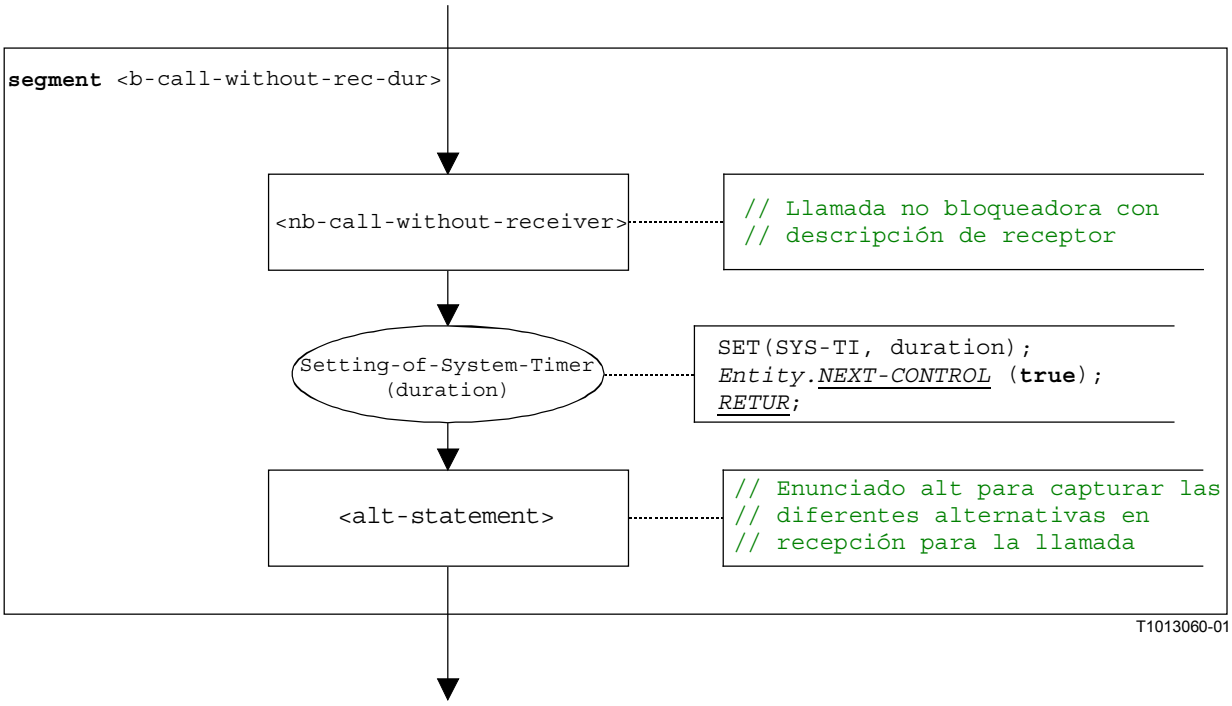


Figura B.38/Z.140 – Segmento de flujograma <b-call-without-rec-dur>

B.3.7.4 Operación Catch

La estructura sintáctica de la operación **catch** es:

```
<portId>.catch (<matchingSpec>) [from <component_expression>] ->
                                                    [<assignmentPart>]
```

La <component_expression> facultativa en la cláusula **from** hace referencia al emisor de la excepción. Se puede proporcionar en forma de un valor de variable o el valor devuelto de una función, es decir, se supone que sea una expresión. La <assignmentPart> facultativa indica la asignación de información capturada si la excepción capturada concuerda con la especificación de concordancia <matchingSpec> y con la cláusula **from** (facultativa).

El segmento de flujograma <catch-op> de la figura B.39 define la ejecución de una operación **catch**.

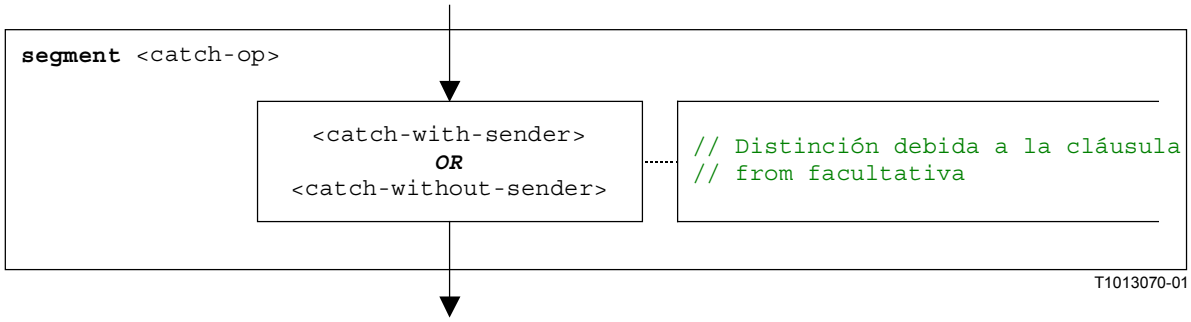


Figura B.39/Z.140 – Segmento de flujograma <catch-op>

B.3.7.4.1 Segmento de flujograma <catch-with-sender>

El segmento de flujograma <catch-with-sender> de la figura B.40 define la ejecución de una operación **catch** cuando el emisor es especificado en forma de una expresión.

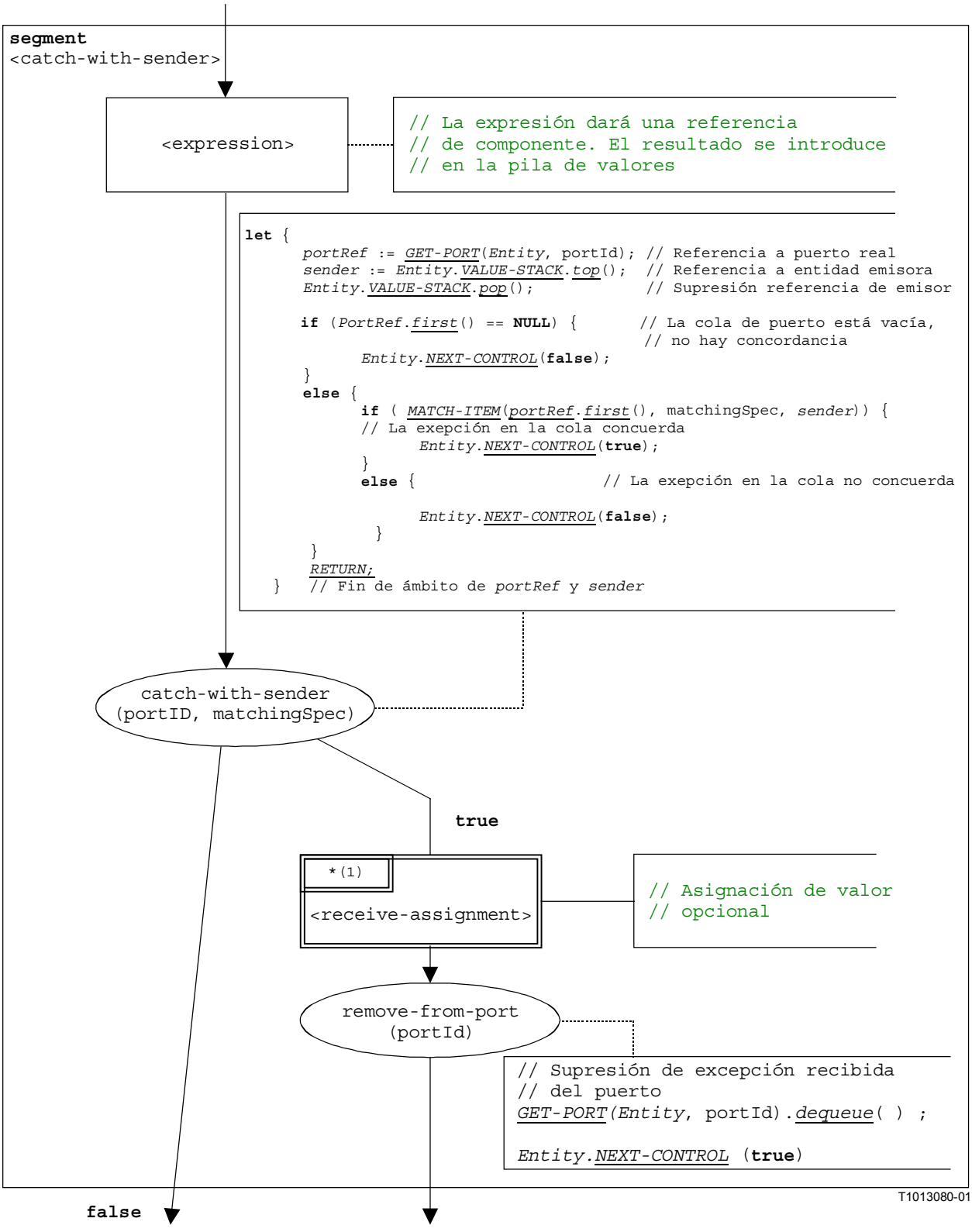


Figura B.40/Z.140 – Segmento de flujograma <catch-with-sender>

B.3.7.4.2 Segmento de flujograma <catch-without-sender>

El segmento de flujograma `<catch-without-sender>` de la figura B.41 define la ejecución de una operación **`catch`** sin una cláusula **`from`**.

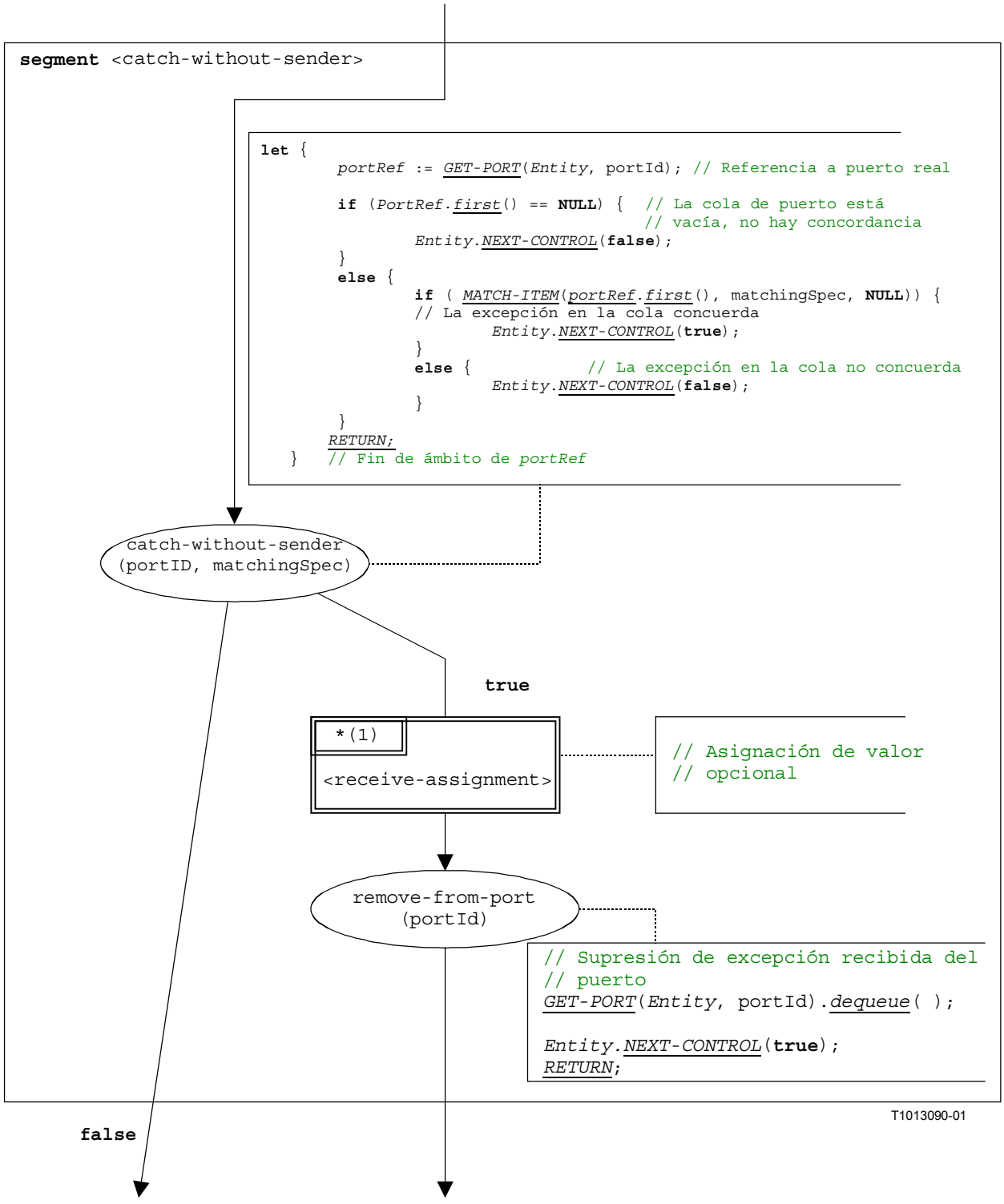


Figura B.41/Z.140 – Segmento de flujograma <catch-without-sender>

B.3.7.5 Operación Clear port

La estructura sintáctica de la operación **clear** port es:

<portId>**.clear**

El segmento de flujograma <clear-port-op> de la figura B.42 define la ejecución de la operación **clear** port.

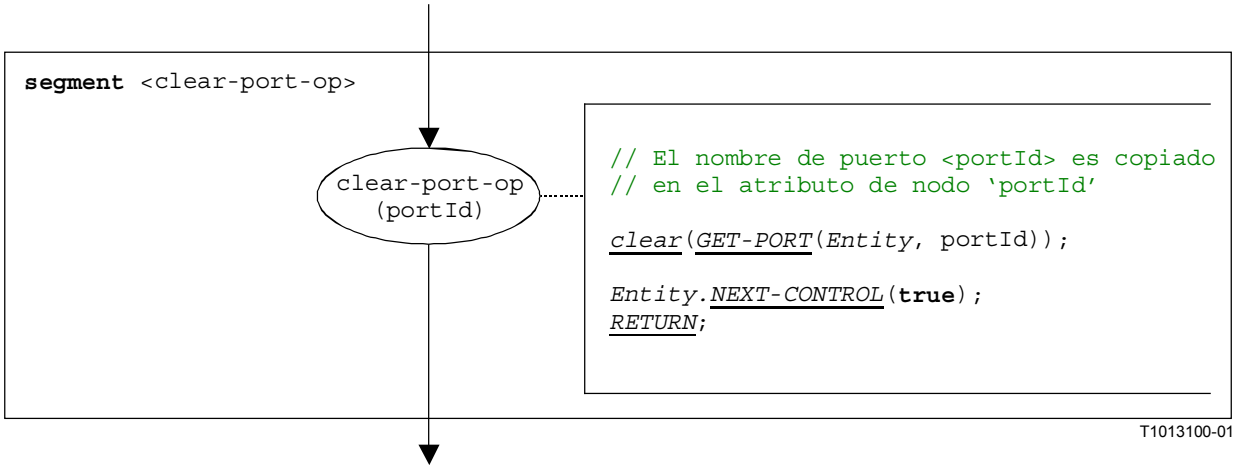


Figura B.42/Z.140 – Segmento de flujograma <clear-port-op>

B.3.7.6 Operación Connect

La estructura sintáctica de una operación **connect** es:

connect(<component_expression$_1$>.<portId1>,<component_expression$_2$>.<portId2>)

Se considera que los identificadores <portId1> y <portId2> son identificadores de puerto de los componentes de prueba correspondientes. Se hace referencia a los componentes a los cuales pertenecen los puertos por medio de las referencias de componentes <component_expression$_1$> y <component_expression$_2$>. Las referencias pueden ser almacenadas en variables o ser devueltas por una función. Para simplificar, son consideradas como expresiones que dan una referencia de componente. De este modo, la pila de valores se utiliza para almacenar las referencias de componentes.

La ejecución de la operación **connect** es definida por el segmento de flujograma <connect-op> mostrado en la figura B.43. En la descripción del flujograma la primera expresión que se ha de evaluar hace referencia a <component_expression$_1$> y la segunda expresión a <component_expression$_2$>, es decir, <component_expression$_2$> está en la cima de la pila de valores cuando se ejecuta el nodo connect-op.

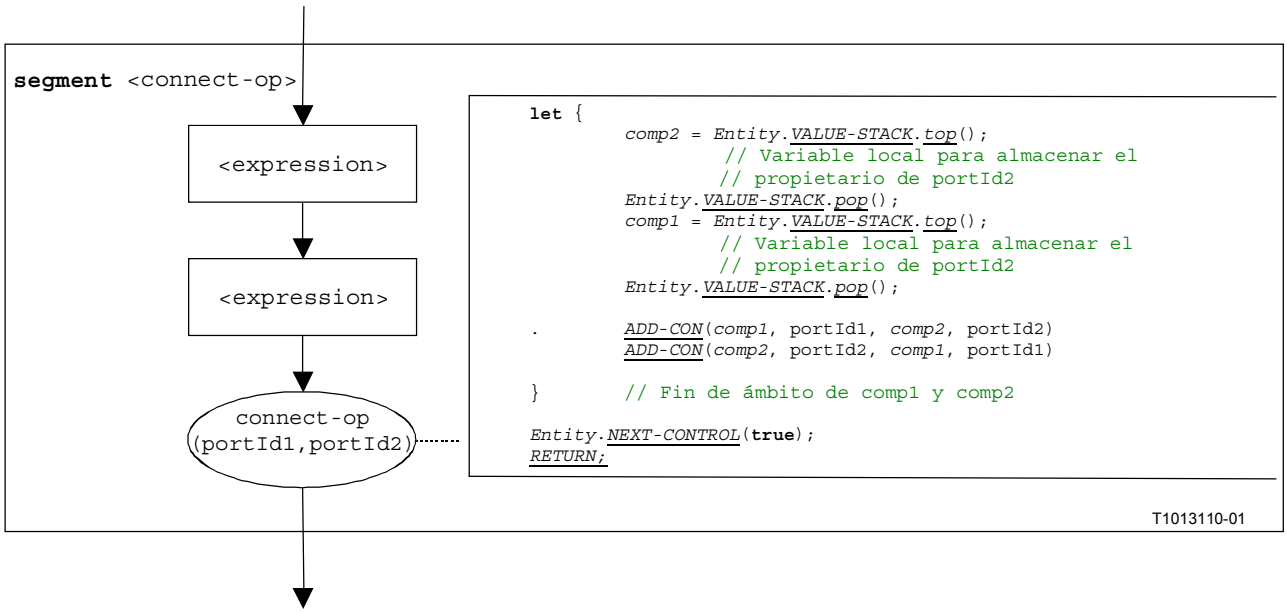


Figura B.43/Z.140 – Segmento de flujograma <connect-op>

B.3.7.7 Declaración de una constante

La estructura sintáctica de una declaración de constante es:

```
const <constType> <constId> := <constType-expression>
```

Se considera que el valor de una constante es una expresión que da un valor del tipo de la constante.

NOTA – Las constantes globales son sustituidas por sus valores en un paso previo al procesamiento antes de aplicar esta semántica (véase B.2.3). Las constantes locales son tratadas como declaraciones de variables con inicialización. Durante el análisis semántico estático de un módulo TTCN-3 se comprobará el uso correcto de constantes, es decir, las constantes nunca se producirán en el lado izquierdo de una asignación.

El segmento de flujograma <constant-declaration> de la figura B.44 define la ejecución de una declaración de constante cuando el valor de la constante se proporciona en forma de una expresión.

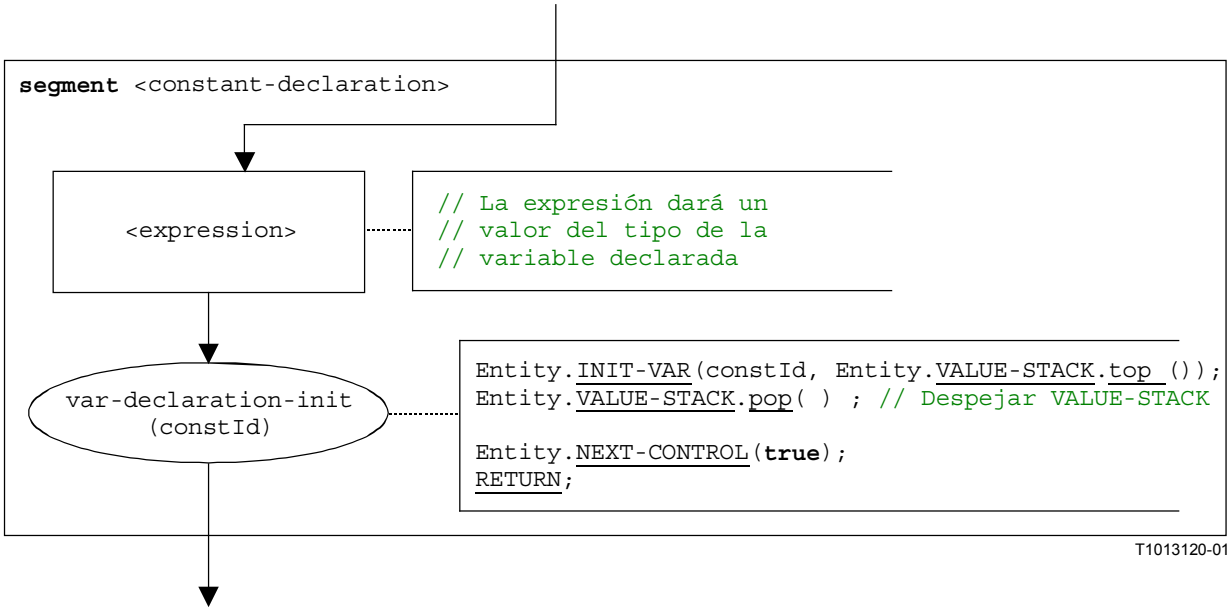


Figura B.44/Z.140 – Segmento de flujograma <constant-declaration>

B.3.7.8 Operación Create

La estructura sintáctica de la operación **create** es:

```
<componentTypeId>.create
```

El segmento de flujograma <create-op> de la figura B.45 define la ejecución de la operación **create**.

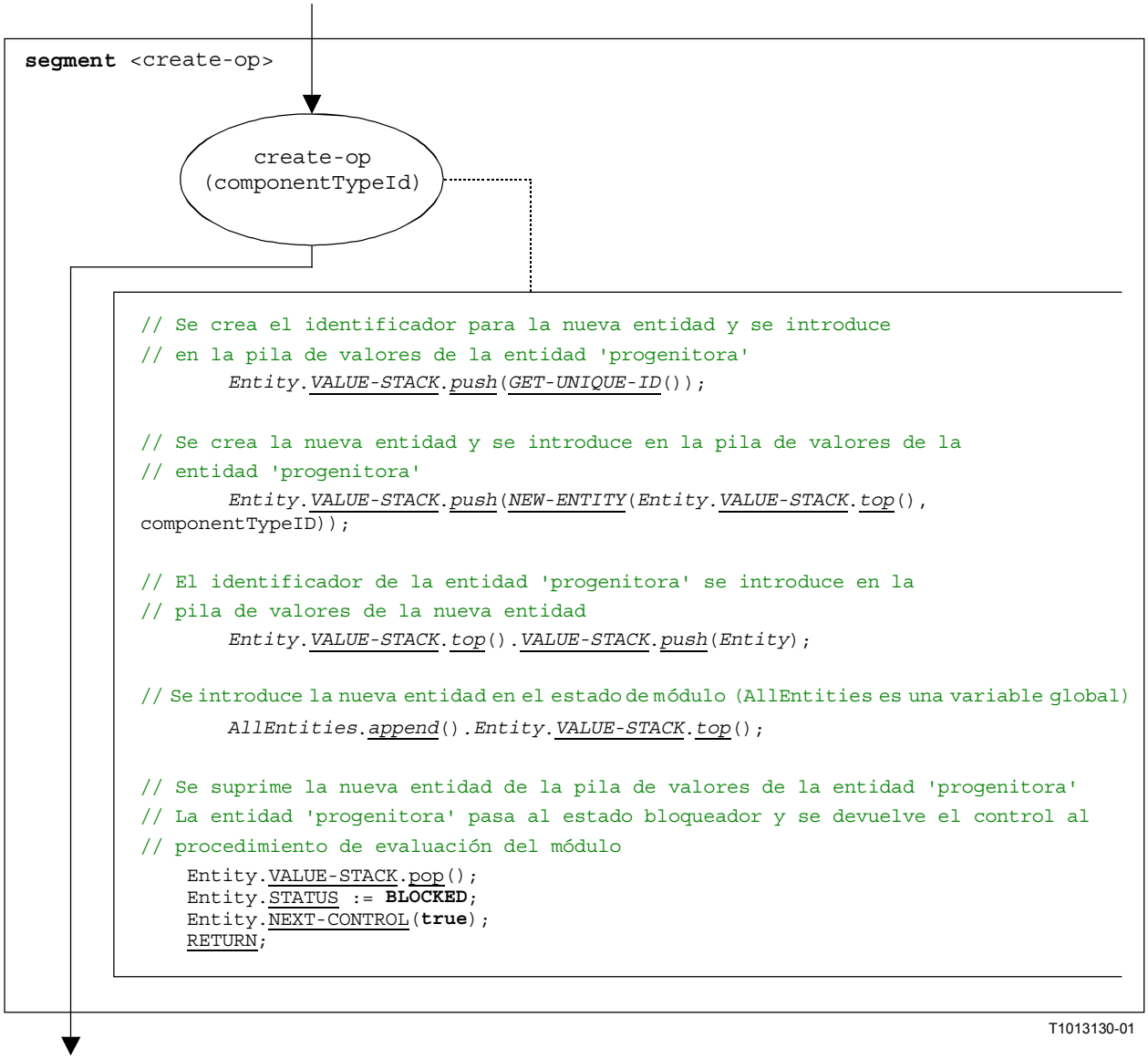


Figura B.45/Z.140 – Segmento de flujograma <create-op>

B.3.7.9 Declaración de un puerto

La estructura sintáctica de una declaración de puerto es:

```
<portType> <portName>
```

Las declaraciones de puerto pueden hallarse en definiciones de tipos de componentes. El efecto de una declaración de puerto es la creación de un nuevo puerto. El segmento de flujograma <port-declaration> de la figura B.46 define la ejecución de una declaración de puerto.

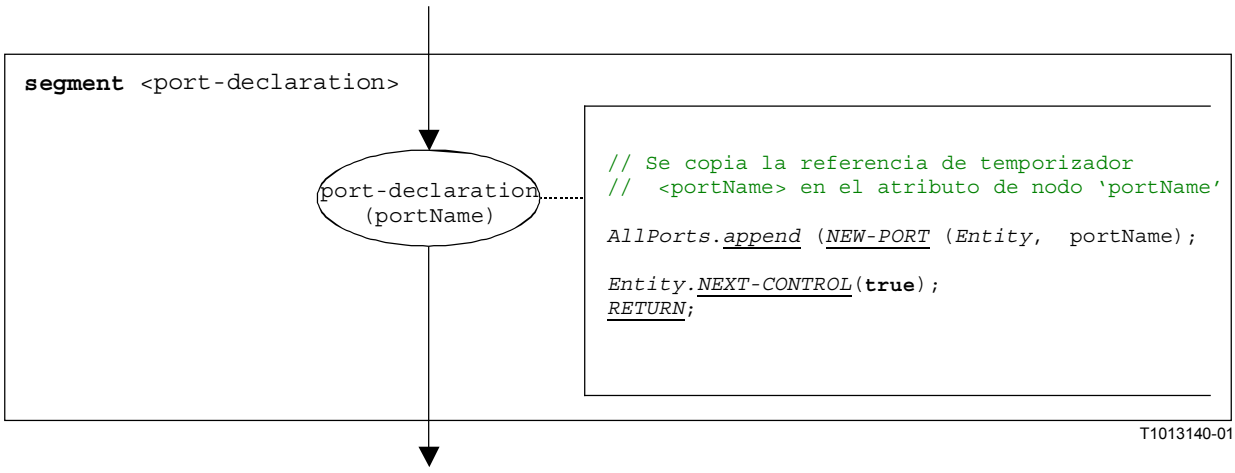


Figura B.46/Z.140 – Segmento de flujograma <port-declaration>

B.3.7.10 Declaración de un temporizador

La estructura sintáctica de una declaración de temporizador es:

```
timer <timerId> [:= <float_expression>]
```

El efecto de una declaración de temporizador es la creación de una nueva vinculación de temporizador. La declaración de una variable con una duración por defecto es facultativa. Se considera que el valor por defecto es una expresión que da un valor del tipo **float**.

El segmento de flujograma <timer-declaration> de la figura B.47 define la ejecución de la declaración de un temporizador.

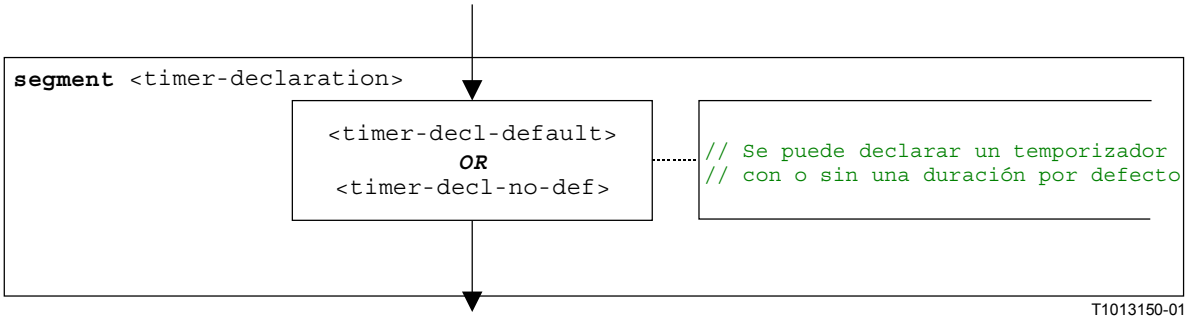


Figura B.47/Z.140 – Segmento de flujograma <timer-declaration>

B.3.7.10.1 Segmento de flujograma <timer-decl-default>

El segmento de flujograma <timer-decl-default> de la figura B.48 define la ejecución de una declaración de temporizador cuando se proporciona una duración por defecto en forma de una expresión.

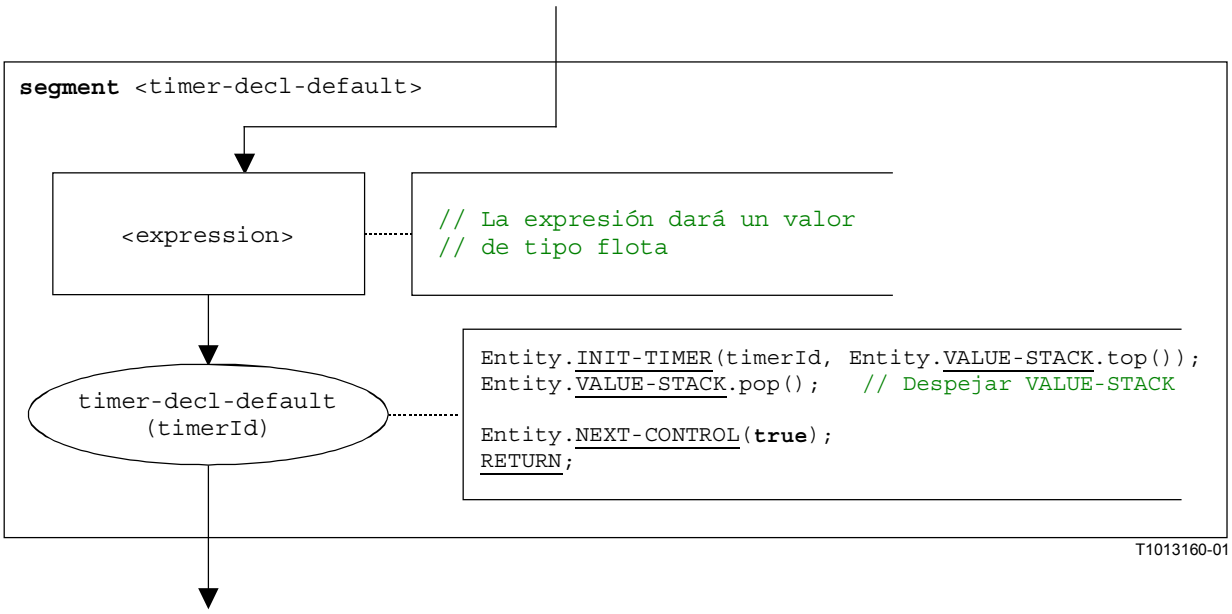


Figura B.48/Z.140 – Segmento de flujograma <timer-decl-default>

B.3.7.10.2 Segmento de flujograma <timer-decl-no-def>

El segmento de flujograma `<timer-decl-no-def>` de la figura B.49 define la ejecución de una declaración de temporizador cuando no se proporciona duración por defecto, es decir, la duración por defecto del temporizador es indefinida.

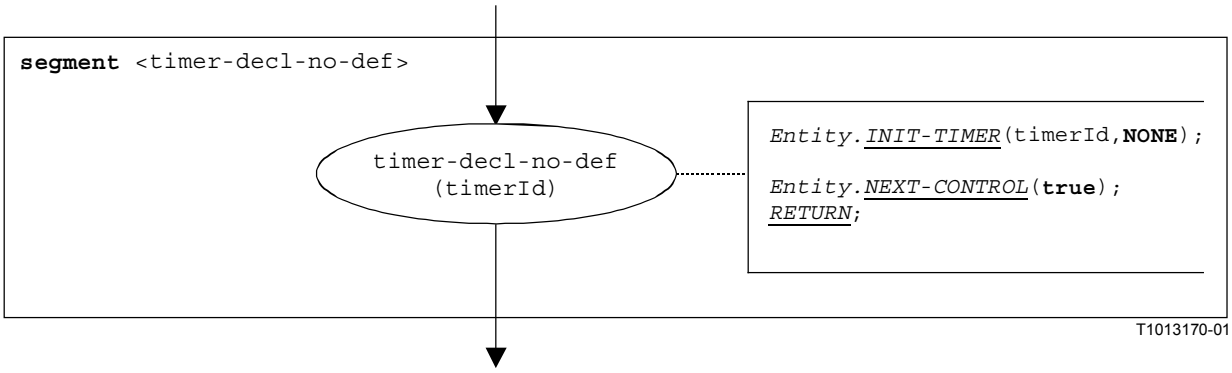


Figura B.49/Z.140 – Segmento de flujograma <timer-decl-no-def>

B.3.7.11 Declaración de una variable

La estructura sintáctica de una declaración de variable es:

```
var <varType> <varId> [:= <varType_expression>]
```

La inicialización de una variable con un valor inicial es facultativa. Se considera que el valor inicial es una expresión que da un valor del tipo de la variable.

El segmento de flujograma `<variable-declaration>` de la figura B.50 define la ejecución de la declaración de una variable.

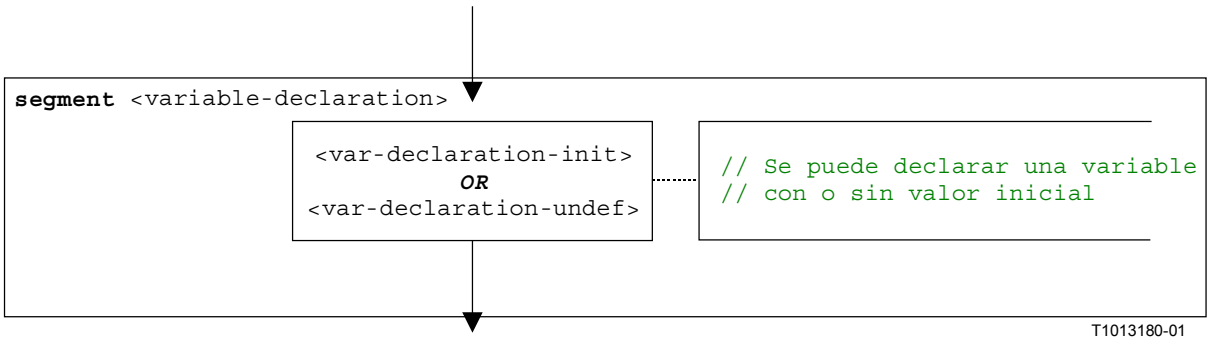


Figura B.50/Z.140 – Segmento de flujograma <variable-declaration>

B.3.7.11.1 Segmento de flujograma <var-declaration-init>

El segmento de flujograma `<var-declaration-init>` de la figura B.51 define la ejecución de una declaración de variable cuando se proporciona un valor inicial en forma de una expresión.

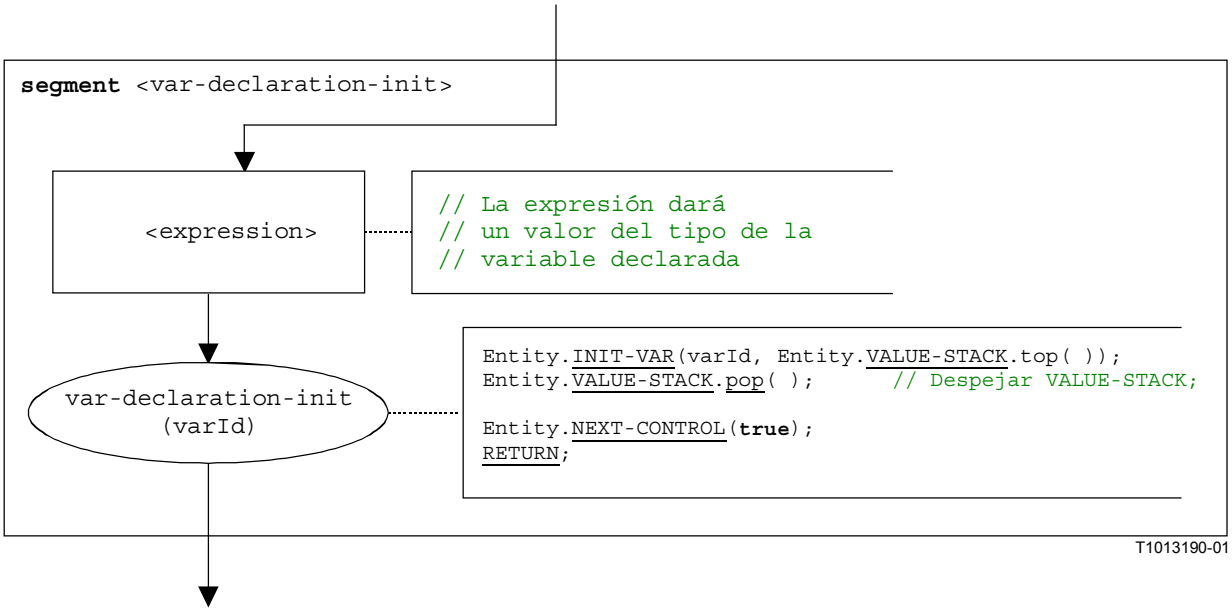


Figura B.51/Z.140 – Segmento de flujograma <var-declaration-init>

B.3.7.11.2 Segmento de flujograma <var-declaration-undef>

El segmento de flujograma `<var-declaration-undef>` de la figura B.52 define la ejecución de una declaración de variable cuando no se proporciona el valor inicial, es decir, el valor de la variable es indefinido.

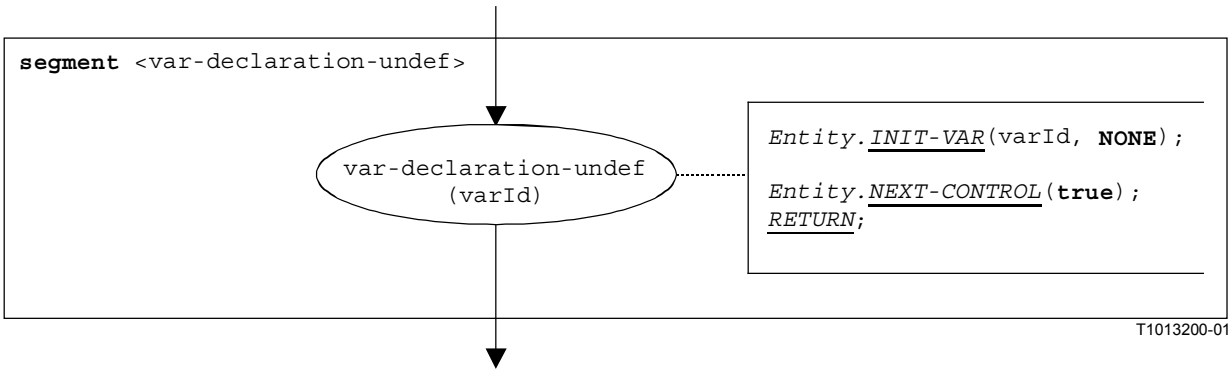


Figura B.52/Z.140 – Segmento de flujograma <var-declaration-undef >

B.3.7.12 Operación Disconnect

La estructura sintáctica de una operación **disconnect** es:

disconnect(<component_expression$_1$>.<portId1>,<component_expression$_2$>.<portId2>)

Se considera que los identificadores <portId1> y <portId2> son identificadores de puerto de los componentes de prueba correspondientes. Los componentes a los cuales pertenecen los puertos son referenciados por medio de las referencias de componentes <component_expression$_1$> y <component_expression$_2$>. Las referencias pueden ser almacenadas en variables o devueltas por una función. Para simplificar, se considera que son expresiones que dan una referencia de componente. De este modo, la pila de valores se utiliza para almacenar las referencias de componentes.

La ejecución de una operación **disconnect** es definida por el segmento de flujograma <disconnect-op> mostrado en la figura B.53. En el segmento de flujograma, la primera expresión que ha de ser evaluada hace referencia a <component_expression$_1$> y la segunda expresión a <component_expression$_2$>, es decir, <component_expression$_2$> está en la cima de la pila de valores cuando se ejecuta el nodo disconnect-op.

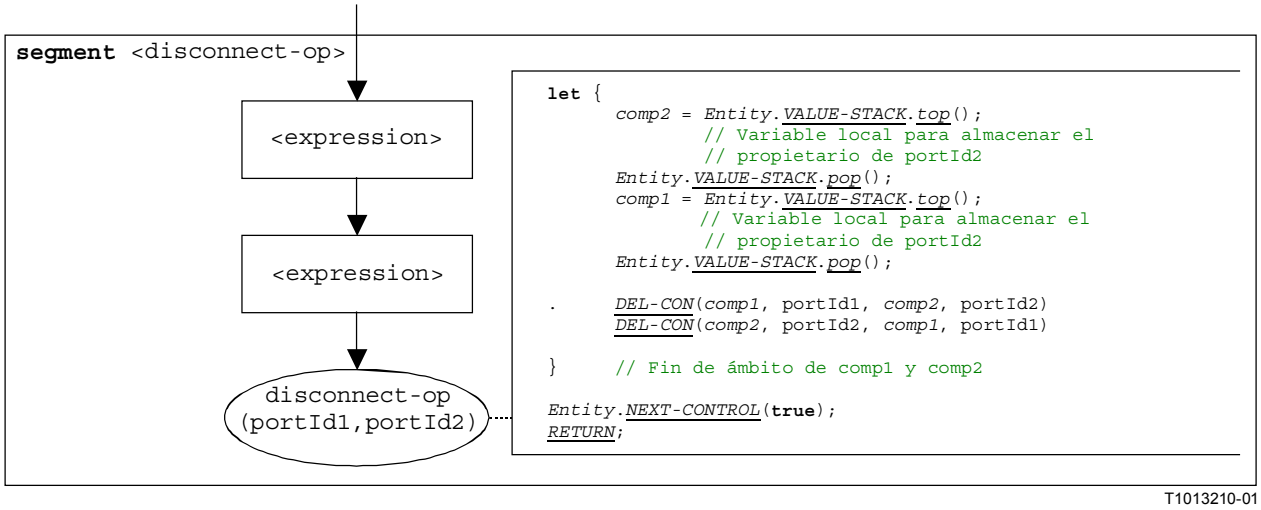


Figura B.53/Z.140 – Segmento de flujograma <disconnect-op>

B.3.7.13 Enunciado Do-while

La estructura sintáctica del enunciado **do-while** es:

```
do <statement-block>
while (<boolean_expression>)
```

La ejecución de un enunciado **do-while** es definido por el segmento de flujograma <do-while-stmt> mostrado en la figura B.54.

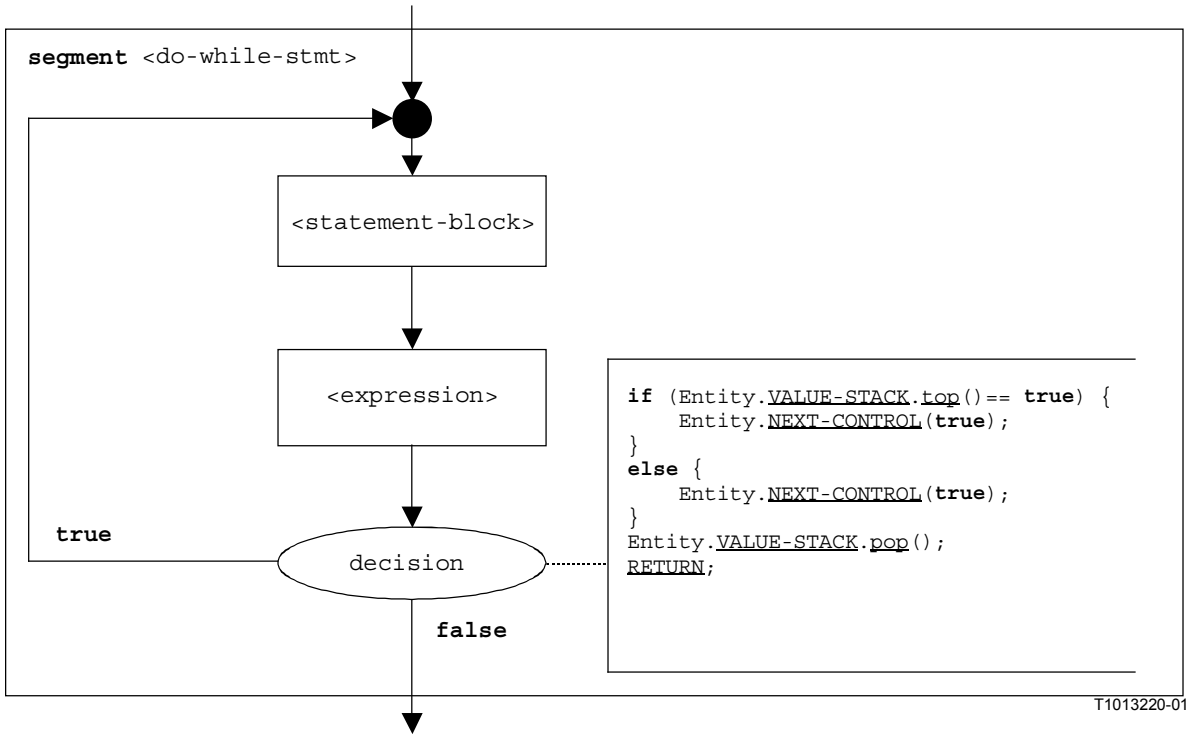


Figura B.54/Z.140 – Segmento de flujograma <do-while-stmt>

B.3.7.14 Operación Done-all-components

En la operación **done-all-components** se hace referencia a la utilización de las palabras clave **all component** en la operación **done** (véase B.7.16). La operación **done-all-components** sólo puede ser invocada por el **mtc**. Permite comprobar si todos los componentes de prueba paralelos de un caso de prueba han terminado. La estructura sintáctica de esta operación es:

```
all component.done;
```

La ejecución de la operación **done-all-components** es definida por el segmento de flujograma <done-all-comp-op> de la figura B.55.

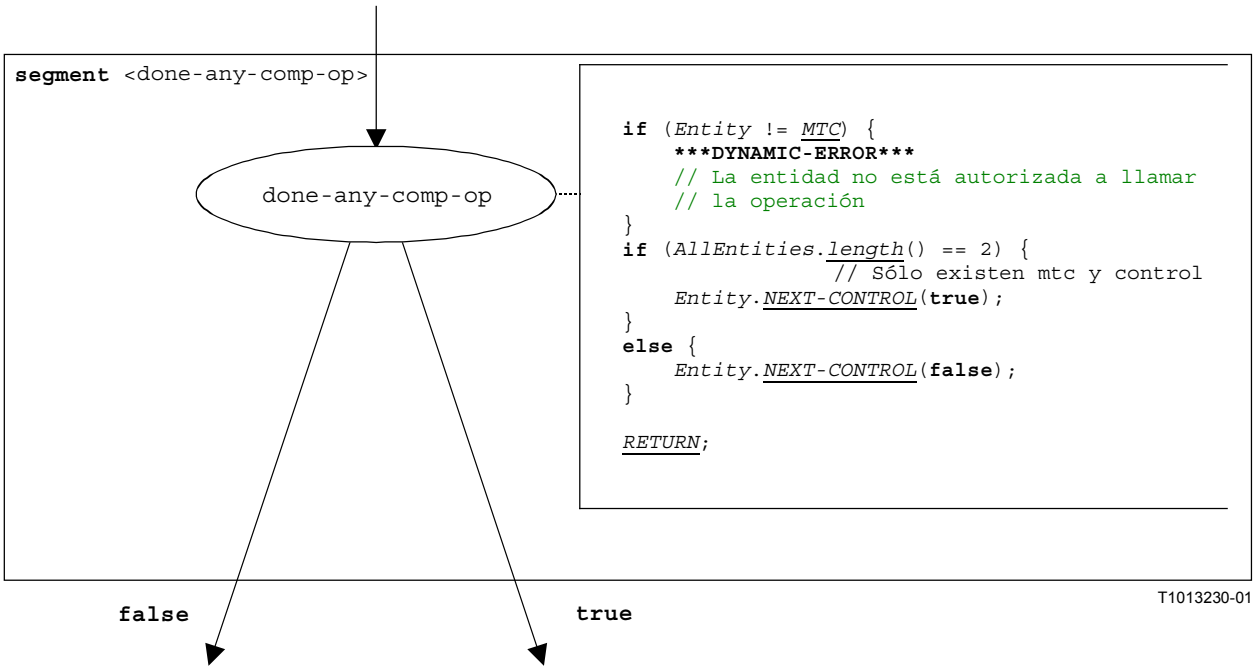


Figura B.55/Z.140 – Segmento de flujograma <done-all-comp-op>

B.3.7.15 Operación Done-any-component

La operación **done-any-component** hace referencia a la utilización de las palabras clave **any component** en la operación **done** (véase B.3.7.16). Esta operación sólo puede ser invocada por el **mtc**. Permite comprobar si un componente de prueba paralelo de un caso de prueba ha terminado ya. La estructura sintáctica de la operación **done-any-component** es:

```
any component.done;
```

La ejecución de la operación **done-any-component** es definida por el segmento de flujograma <done-any-comp-op> de la figura B.56.

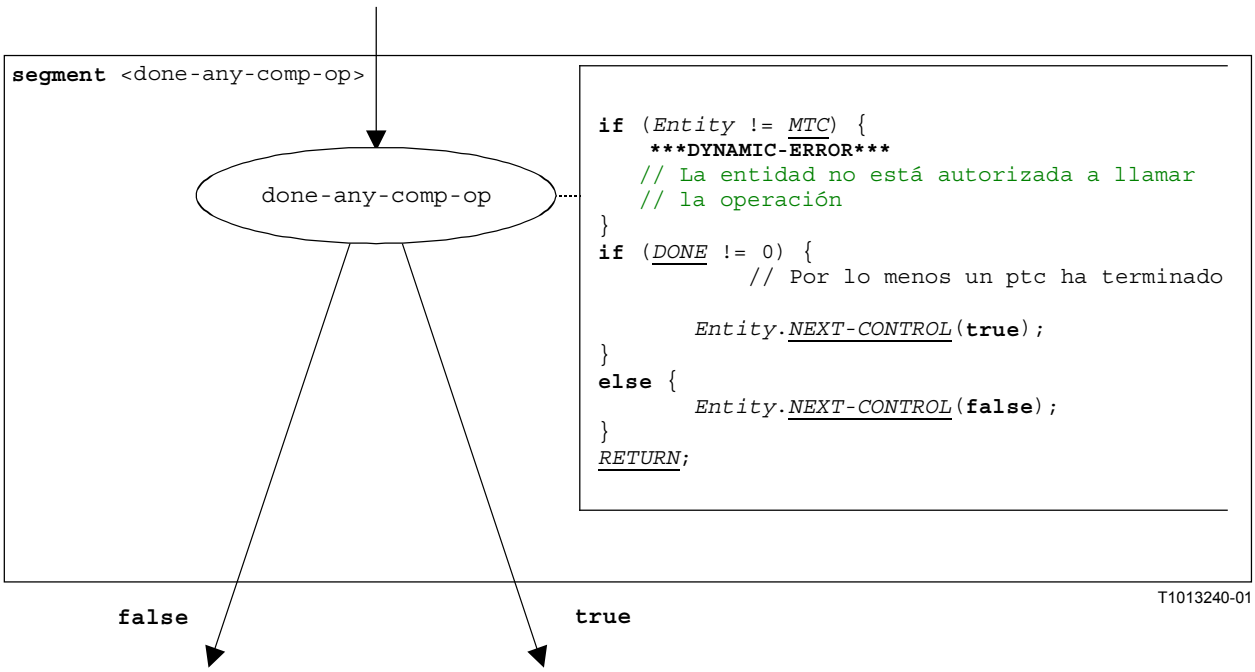


Figura B.56/Z.140 – Segmento de flujograma <done-any-comp-op>

B.3.7.16 Operación Done component

La estructura sintáctica de la operación **done** component es:

<component_expression>.**done**

La operación **done** component verifica si un componente está en funcionamiento o detenido. Si un componente comprobado está en funcionamiento o detenido, la operación **done** decide cómo continúa el flujo de control. Se utiliza una referencia de componente para identificar al componente que ha de ser comprobado. La referencia puede ser almacenada en una variable o devuelta por una función. Para simplificar, se considera como una expresión que da una referencia de componente.

El segmento de flujograma <done-component-op> de la figura B.57 define la ejecución de la operación **done** component.

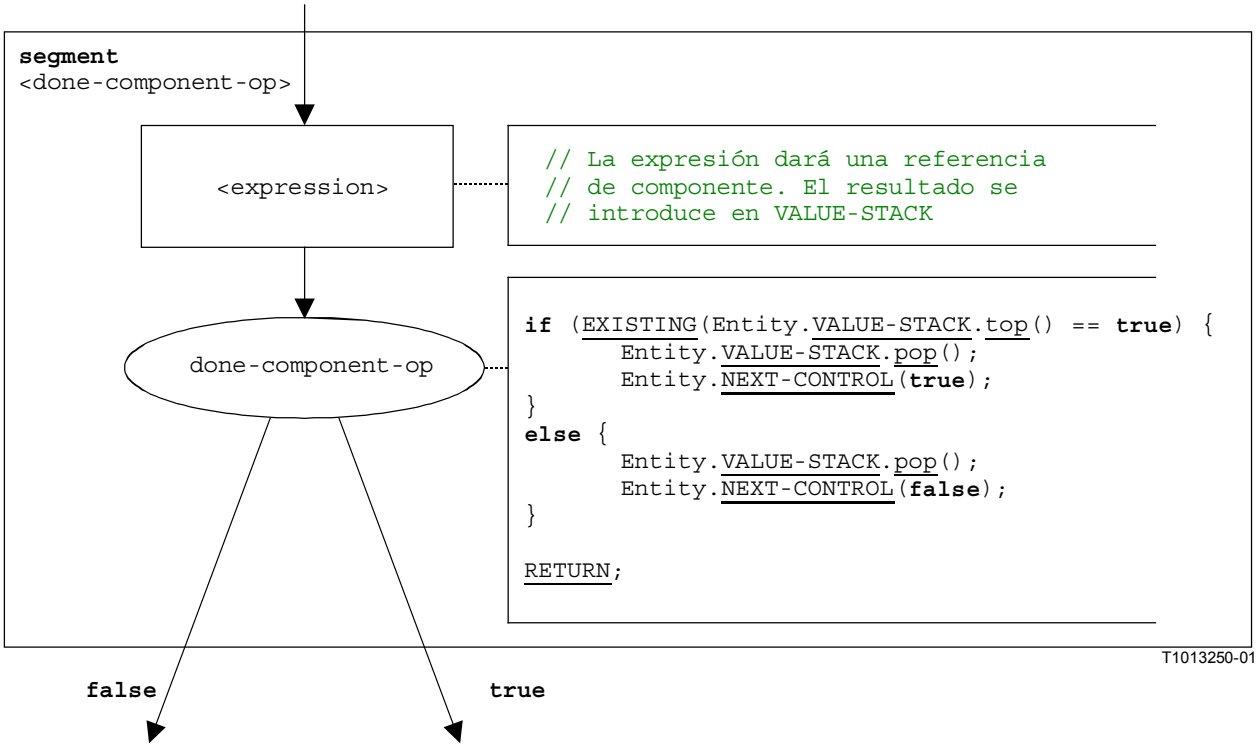


Figura B.57/Z.140 – Segmento de flujograma <done-component-op>

B.3.7.17 Enunciado Execute

La estructura sintáctica del enunciado **execute** es:

execute(<testCaseId>([<act-par$_1$>, ... , <act-par$_n$>)]) [, <float_expression>])

El enunciado **execute** describe la ejecución de un caso de prueba <testCaseId> con los parámetros reales (facultativos) <act-par$_1$>, ... , <act-par$_n$>. Facultativamente, el enunciado **execute** puede ser guardado durante un tiempo cuya duración es proporcionada en forma de una expresión que da un valor **float**. Si dentro de la duración especificada, el caso de prueba no devuelve un veredicto, se produce una excepción de temporización, el caso de prueba es detenido y se devuelve un veredicto **error**. Sin embargo, TTCN-3 no tiene semántica en tiempo real y, por tanto, la decisión de si se produce o no una excepción de temporización es modelada en forma de una elección no determinística.

NOTA – La semántica operacional sólo modela la elección no determinística. No se evalúa la <float_expression>.

Si debido a la elección no determinística no se produce ninguna excepción de temporización, se crea el **mtc**, el caso de control (que representa la parte de control del módulo TTCN-3) es bloqueado hasta que termina el caso de prueba, y para la ulterior ejecución de caso de prueba se da el flujo de control al **mtc**. El flujo de control es devuelto al caso de control cuando el **mtc** termina.

El segmento de flujograma <execute-stmt> de la figura B.58 define la ejecución de un enunciado **execute**.

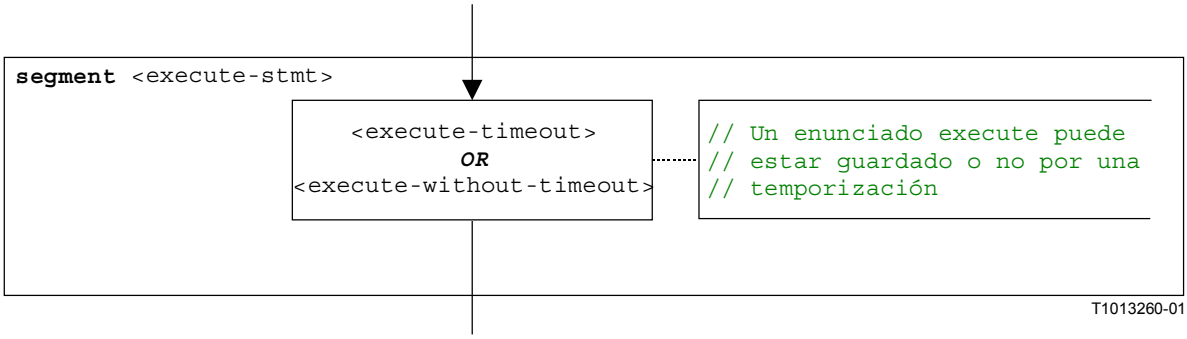


Figura B.58/Z.140 – Segmento de flujograma <execute-stmt>

B.3.7.17.1 Segmento de flujograma <execute-timeout>

El segmento de flujograma <execute-timeout> de la figura B.59 define la ejecución de un enunciado **execute** que tiene un valor de temporización.

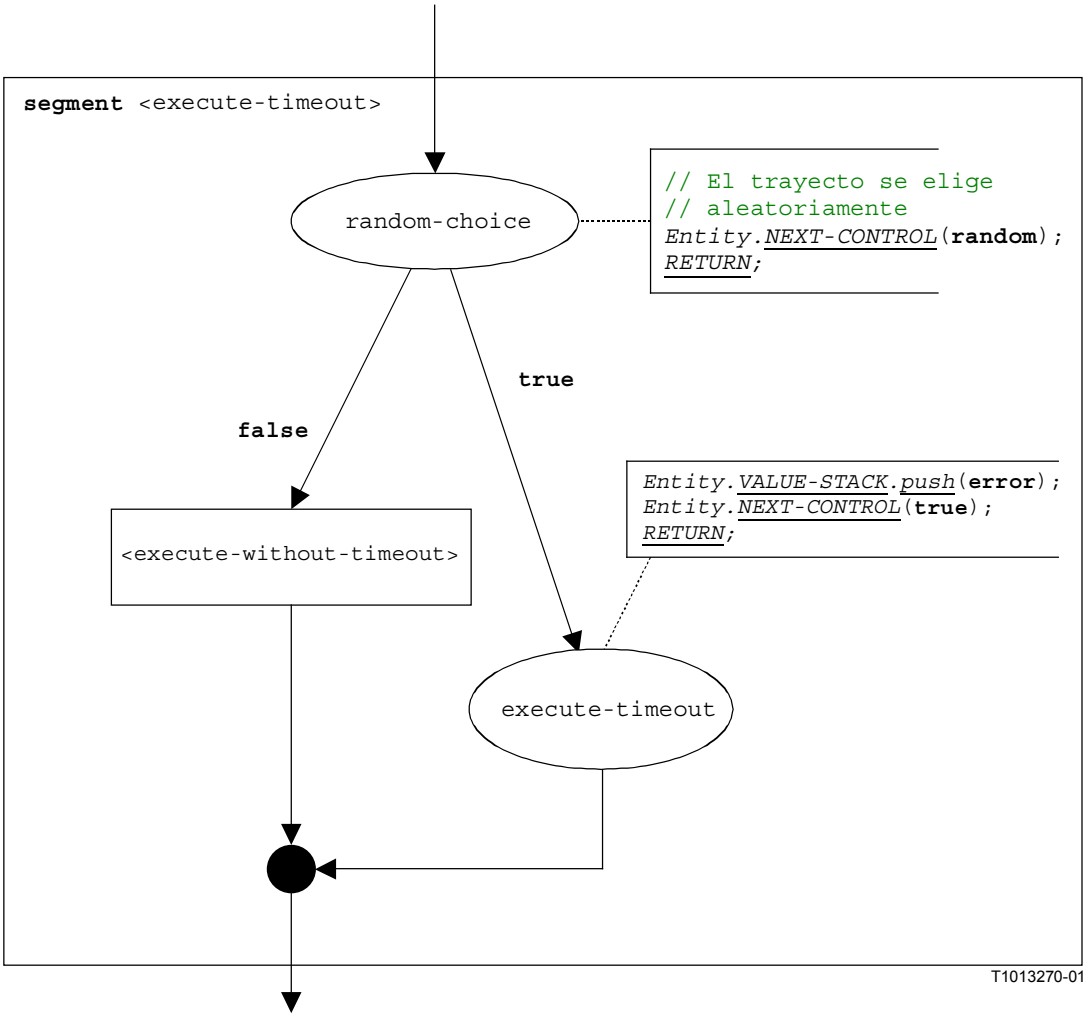


Figura B.59/Z.140 – Segmento de flujograma <execute-timeout>

B.3.7.17.2 Segmento de flujograma <execute-without-timeout>

La ejecución de un caso de prueba comienza con la creación del **mtc**. Después, el **mtc** es arrancado con el comportamiento indicado en la definición del caso de prueba. A continuación, el control de módulo espera hasta que el caso de prueba termina. Se puede describir la creación y el comienzo del **mtc** utilizando los enunciados **create** y **start**:

```
mtcType MyMTC := mtcType.create;
MyMTC.start(TestCaseName(P1...Pn);
```

El segmento de flujograma <execute-without-timeout> de la figura B.60 define la ejecución de un enunciado **execute** sin que ocurra una excepción de temporización utilizando los segmentos de flujograma de las operaciones **create** y **start**.

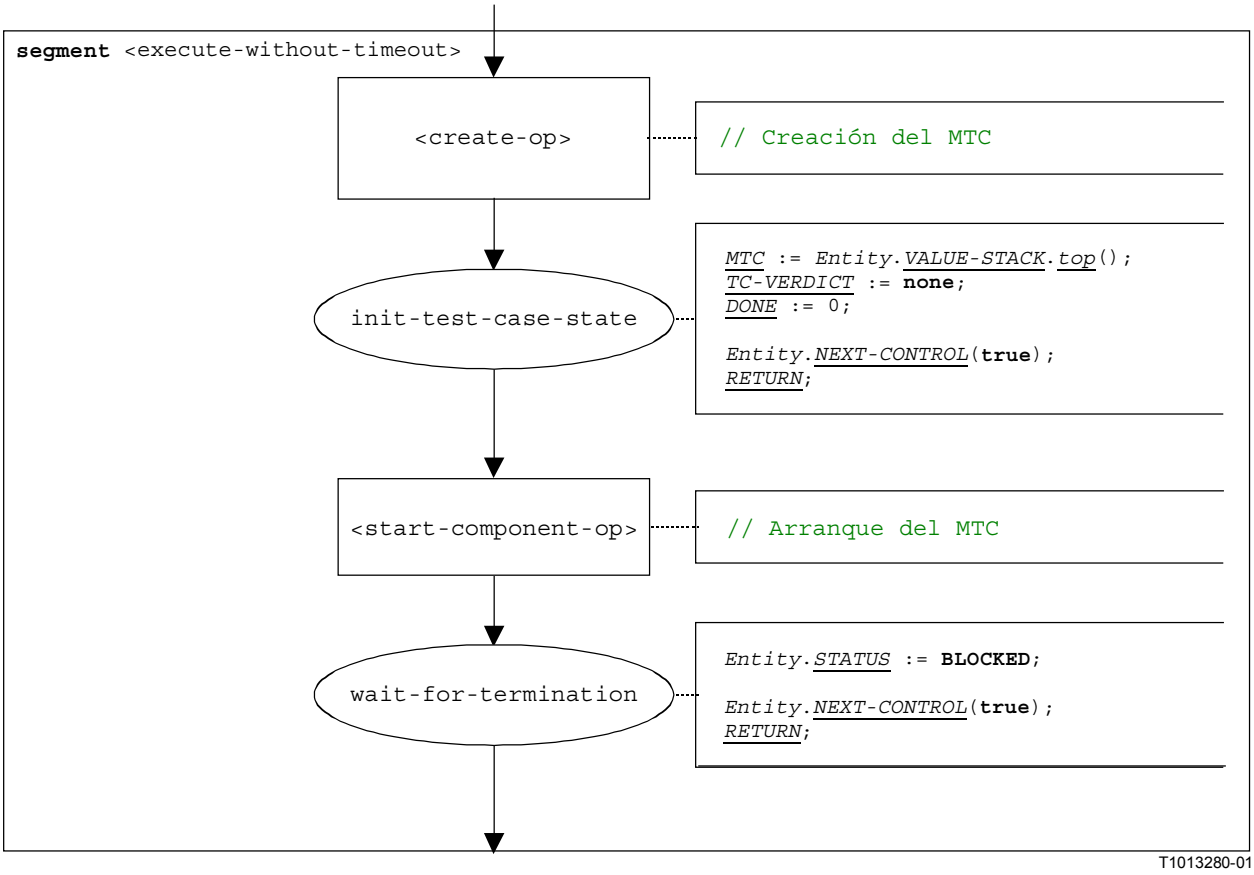


Figura B.60/Z.140 – Segmento de flujograma <execute-without-timeout>

B.3.7.18 Expresión

Para el tratamiento de expresiones, hay que distinguir los cuatro casos siguientes:

a) la expresión es un valor literal (o una constante);

b) la expresión es una variable;

c) la expresión es un operador aplicado a uno o más operandos;

d) la expresión es una llamada de función o de operación.

La estructura sintáctica de una expresión es:

```
<lit-val> | <var-val> | <func-op-call> | <operand-appl>
```

donde:

<lit-val> indica un valor literal;

`<var-val>` indica un valor de variable;

`<func-op-call>` indica una llamada de función o de operación;

`<operator-appl>` indica la aplicación de operadores aritméticos como `+`, `–`, `not`, etc.

La ejecución de una expresión se define mediante el segmento de flujograma `<expression>` mostrado en la figura B.61.

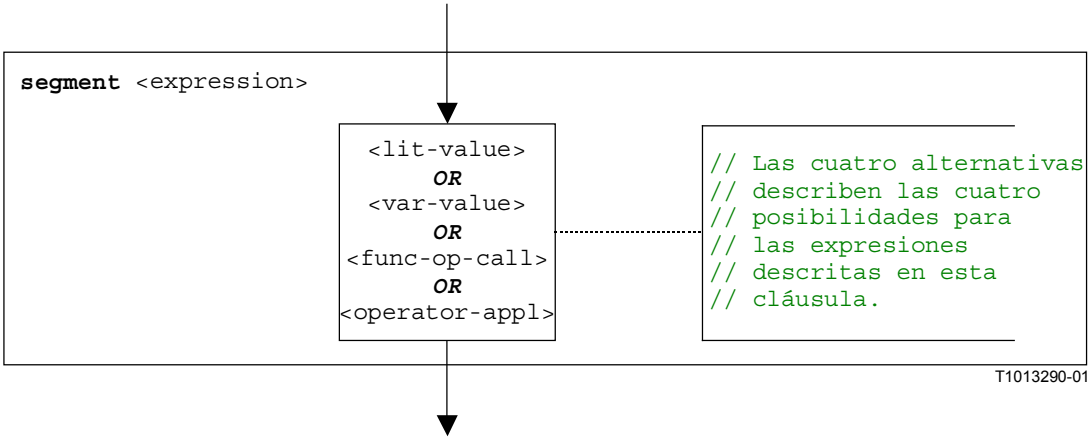


Figura B.61/Z.140 – Segmento de flujograma <expression>

B.3.7.18.1 Segmento de flujograma <lit-value>

El segmento de flujograma `<lit-value>` de la figura B.62 introduce un valor literal en la pila de valores de una entidad.

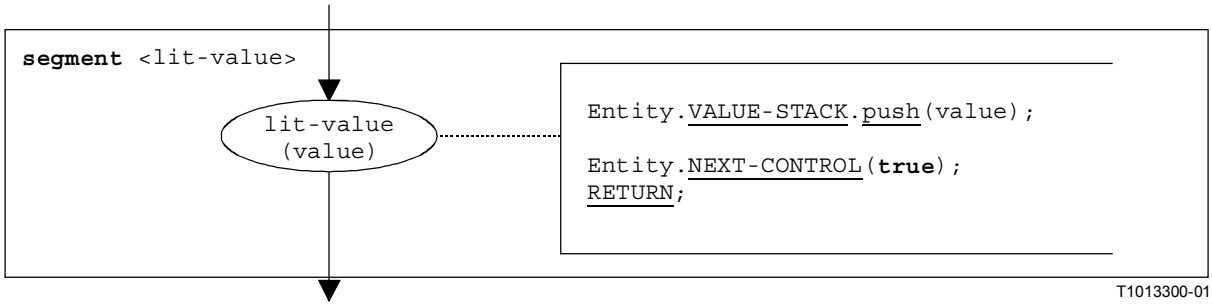


Figura B.62/Z.140 – Segmento de flujograma <lit-value>

B.3.7.18.2 Segmento de flujograma <var-value>

El segmento de flujograma `<var-value>` de la figura B.63 introduce el valor de una variable en la pila de valores de una entidad.

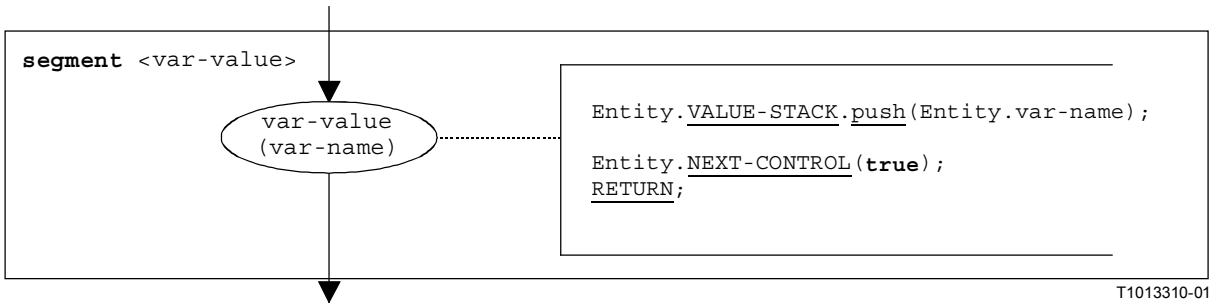


Figura B.63/Z.140 – Segmento de flujograma <var-value>

B.3.7.18.3 Segmento de flujograma <func-op-call>

El segmento de flujograma `<func-op-call>` de la figura B.64 hace referencia a llamadas de funciones y operaciones, que devuelven un valor que es introducido en la pila de valores de una entidad. Se considera que todas estas llamadas son expresiones.

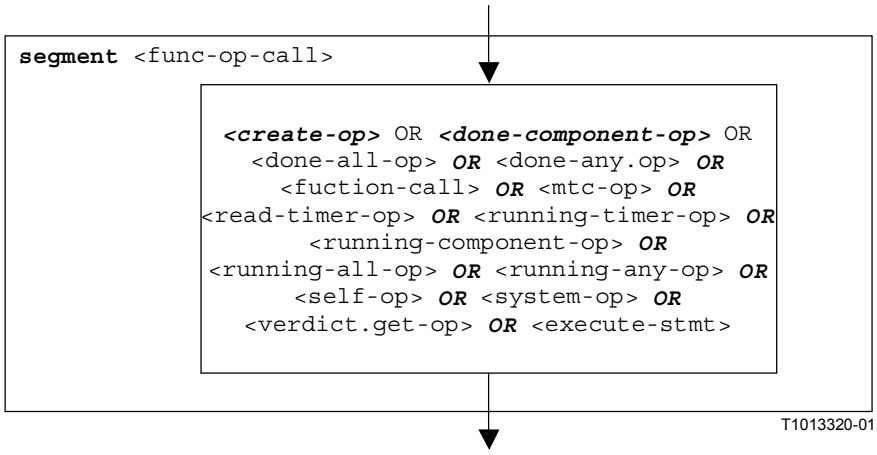


Figura B.64/Z.140 – Segmento de flujograma <func-op-call>

B.3.7.18.4 Segmento de flujograma <operator-appl>

La representación del flujograma de la figura B.65 hace referencia directamente a la hipótesis de que se utiliza notación inversa para evaluar expresiones de operador. Los operandos del operador se calculan e introducen en la pila de evaluación. Para la aplicación del operador, los operandos son extraídos de la pila de evaluación y se aplica el operador. El resultado de la aplicación del operador se introduce finalmente en la pila de evaluación.

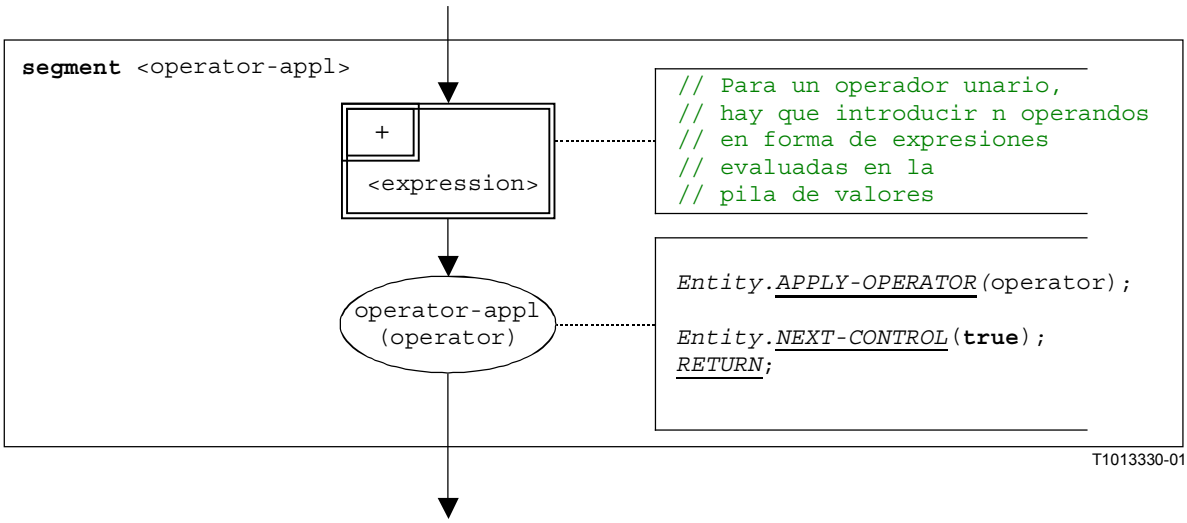


Figura B.65/Z.140 – Segmento de flujograma <operator-appl>

B.3.7.19 Segmento de flujograma <finalize-component-init>

El segmento de flujograma `<finalize-component-init>` forma parte del flujograma que representa el comportamiento de una definición de tipo de componente. Su ejecución se define en la figura B.66.

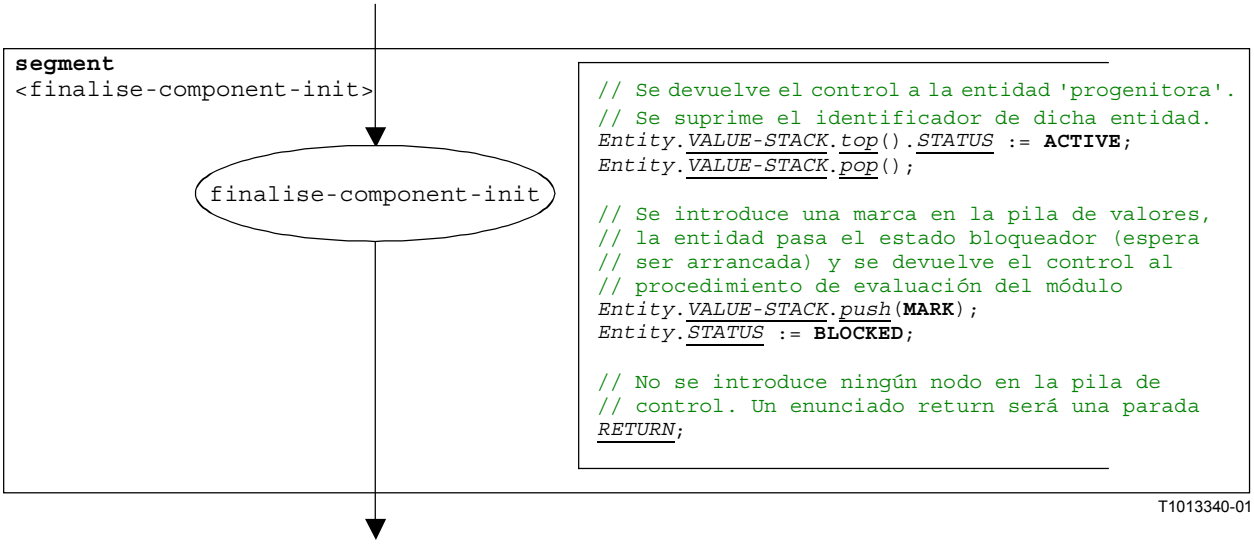


Figura B.66/Z.140 – Segmento de flujograma <finalize-component-init>

B.3.7.20 Segmento de flujograma <init-component-scope>

El segmento de flujograma `<init-component-scope>` forma parte del flujograma que representa el comportamiento de una definición de tipo de componente. Su ejecución se define en la figura B.67.

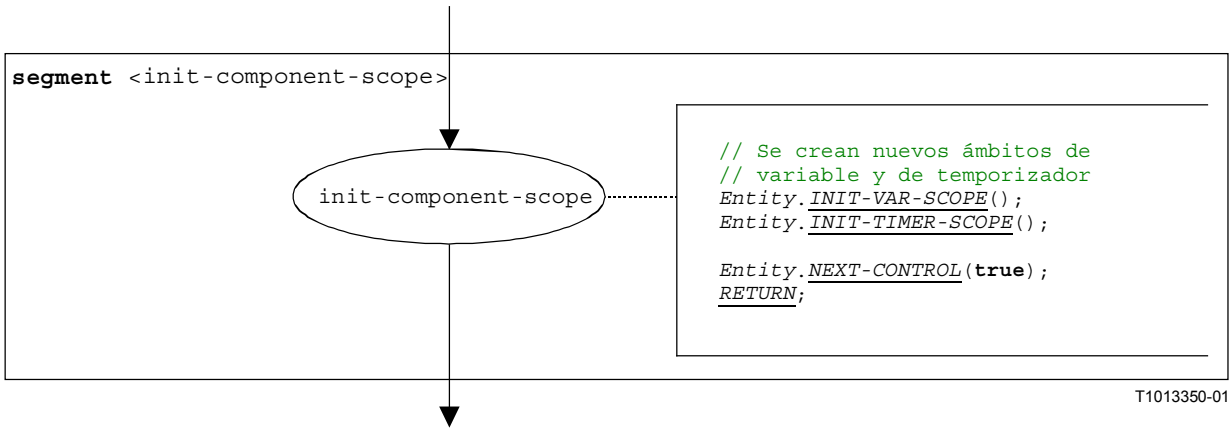


Figura B.67/Z.140 – Segmento de flujograma <init-component-scope>

B.3.7.21 Enunciado For

La estructura sintáctica del enunciado **for** es:

for (<assignment>, <boolean_expression>, <assignment>) <statement-block>

La inicialización de la variable de índice y la correspondiente manipulación de la variable de índice se consideran asignaciones de la variable de índice. La <boolean_expression> describe el criterio de terminación del bucle especificado por el enunciado **for** y el <statement-block> describe el cuerpo del bucle.

La ejecución del enunciado **for** se define mediante el segmento de flujograma <for-stmt> mostrado en la figura B.68. La <assignment> inicial describe la inicialización de la variable de índice. La <assignment> en la rama **true** del nodo de **decisión** describe la manipulación de la variable de índice.

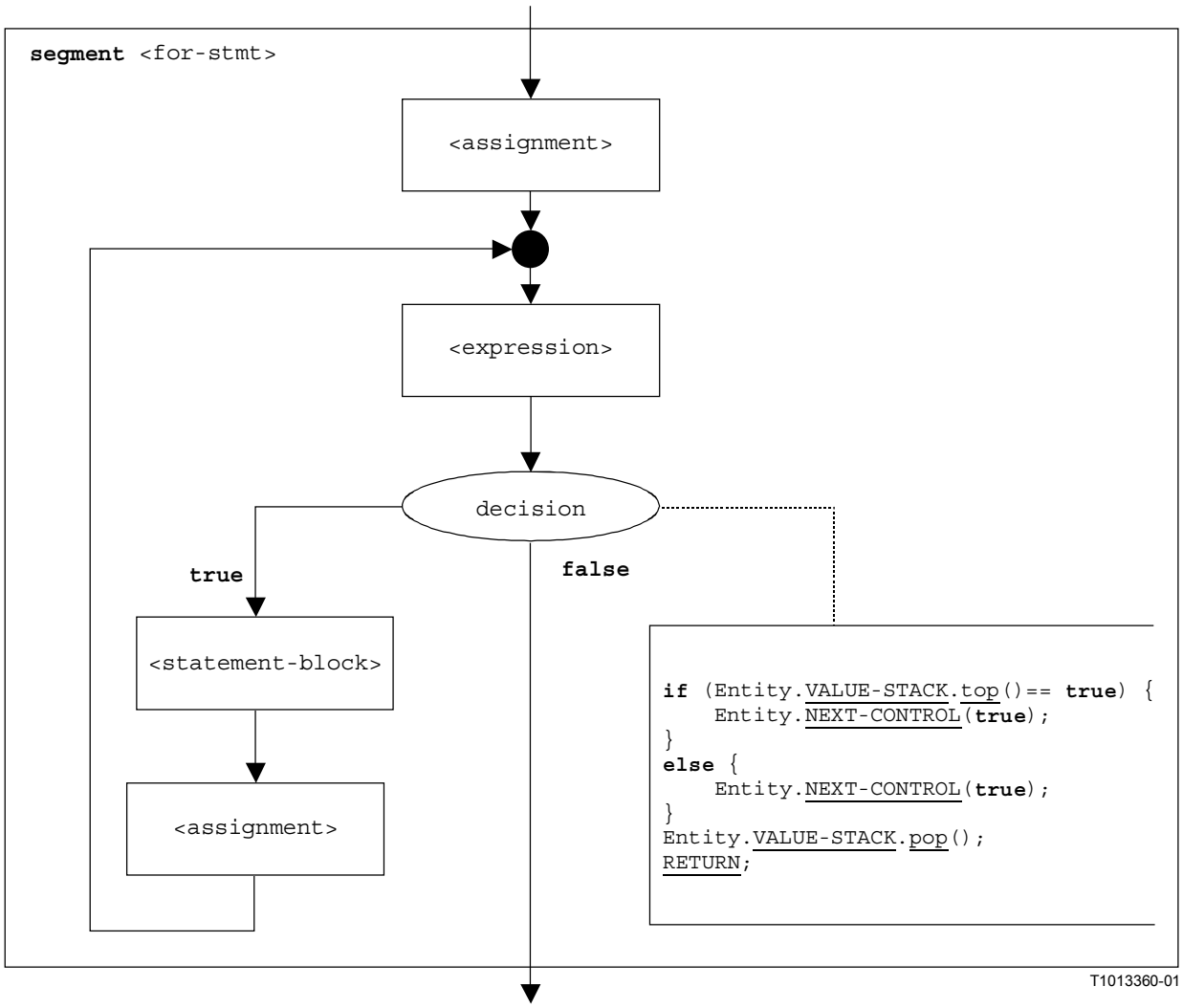


Figura B.68/Z.140 – Segmento de flujograma <for-stmt>

B.3.7.22 Llamada de función

La estructura sintáctica de una llamada de función es:

<function-name>([<act-par-desc$_1$>, ... , <act-par-desc$_n$>])

El <function-name> indica el nombre de una función y <act-par-descr$_1$>, ... , <act-par-descr$_n$> es la descripción de los valores de parámetros reales de la llamada de función. En el caso de un parámetro de valor, la descripción de un parámetro real puede ser proporcionada en forma de una expresión que tiene que ser evaluada antes de poder ejecutar la llamada.

Se supone que para cada <act-par-desc$_1$>, se conoce el correspondiente identificador de parámetro formal <f-par-Id$_1$>, es decir, la estructura sintáctica anterior se puede ampliar a:

<function-name>((<f-par-Id$_1$>, <act-par-desc$_1$>), ... , (<f-par-Id$_n$>, <act-par-desc$_n$>))

El segmento de flujograma <function-call> de la figura B.69 define la ejecución de una llamada de función. La ejecución se estructura en tres pasos. En el primer paso, se crea un registro de llamada para la función <function-name>. En el segundo caso, se calculan los valores de parámetros reales y se asignan al campo correspondiente en el registro de llamada. En el tercer paso, se transfiere el control del comportamiento que llama a la función.

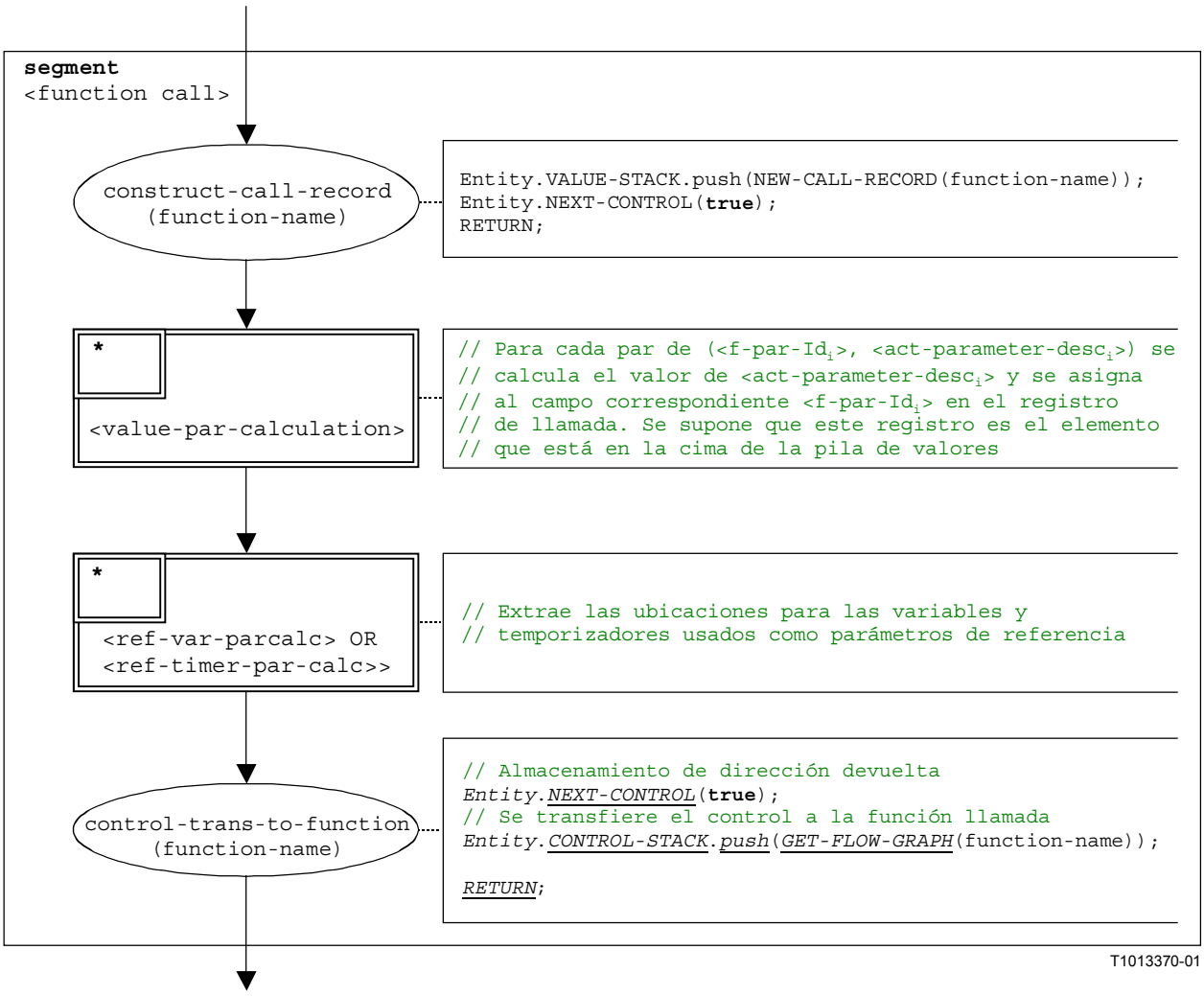


Figura B.69/Z.140 – Segmento de flujograma <function-call>

B.3.7.23 Segmento de flujograma <value-par-calculation>

El segmento de flujograma `<value-par-calculation>` se utiliza para calcular valores de parámetros reales y asignarlos a campos correspondientes en registros de llamada para funciones y casos de prueba.

Se supone que un registro de llamada es el elemento en la cima de la pila de valores y que hay que tratar un par de:

`(<f-par-Id`$_i$`>, <act-parameter-desc`$_i$`>)`

Hay que evaluar `<act-parameter-desc`$_i$`>` y `<f-par-Id`$_i$`>` es el identificador de un parámetro formal que tiene un campo correspondiente en el registro de llamada en la pila de valores.

La ejecución del segmento de flujograma `<value-par-calculation>` se muestra en la figura B.70.

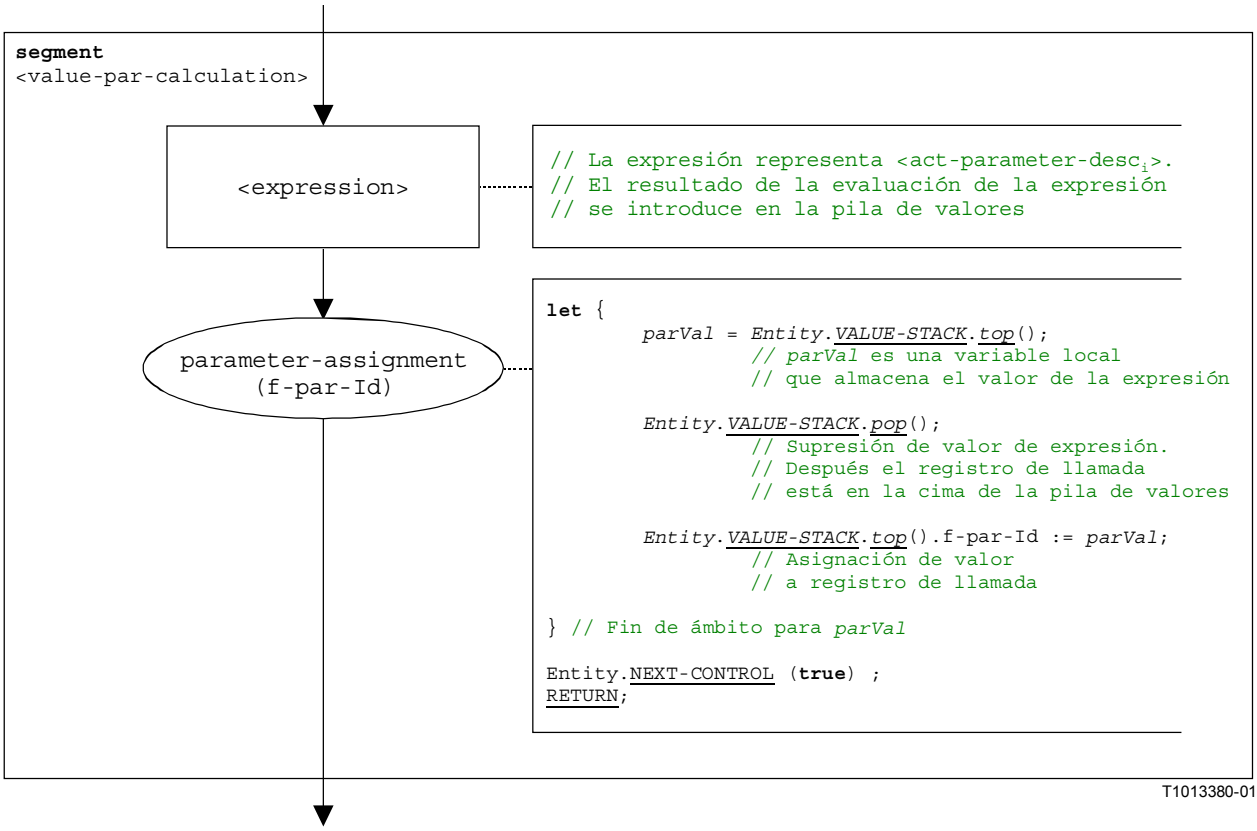


Figura B.70/Z.140 – Segmento de flujograma <value-par-calculation>

B.3.7.24 Segmento de flujograma <ref-par-var-calc>

El segmento de flujograma `<ref-par-var-calc>` se utiliza para extraer las ubicaciones de variables utilizadas como parámetros de referencias reales y asignarlas a los campos correspondientes en los registros de llamada para funciones y casos de prueba.

Se supone que un registro de llamada es el elemento en la cima de la pila de valores y que hay que tratar un par de:

(<f-par-Id$_i$>, <act-par$_i$>)

<act-par$_i$> es el parámetro real cuya ubicación tiene que ser extraída y <f-par-Id$_i$> es el identificador de un parámetro formal que tiene un campo correspondiente en el registro de llamada en la pila de valores.

La ejecución del segmento de flujograma `<ref-par-var-calc>` se muestra en la figura B.71.

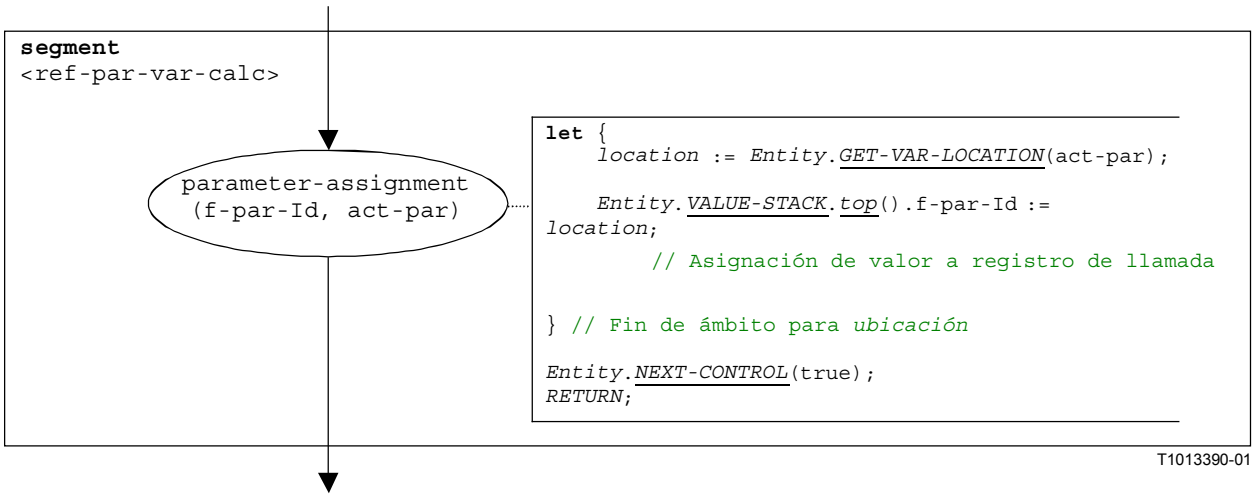


Figura B.71/Z.140 – Segmento de flujograma <ref-par-var-calc>

B.3.7.25 Segmento de flujograma <ref-par-timer-calc>

El segmento de flujograma `<ref-par-timer-calc>` se utiliza para extraer las ubicaciones de temporizadores utilizados como parámetros de referencia reales y asignarlos a los campos correspondientes en registros de llamada para funciones y casos de prueba.

Se supone que un registro de llamada es el elemento en la cima de la pila de valores y que hay que tratar un par de:

(<f-par-Id$_i$>, <act-par$_i$>)

<act-par$_i$> es el parámetro real cuya ubicación tiene que ser extraída y <f-par-Id$_i$> es el identificador de un parámetro formal que tiene un campo correspondiente en el registro de llamada en la pila de valores.

La ejecución del segmento de flujograma `<ref-par-timer-calc>` se muestra en la figura B.72.

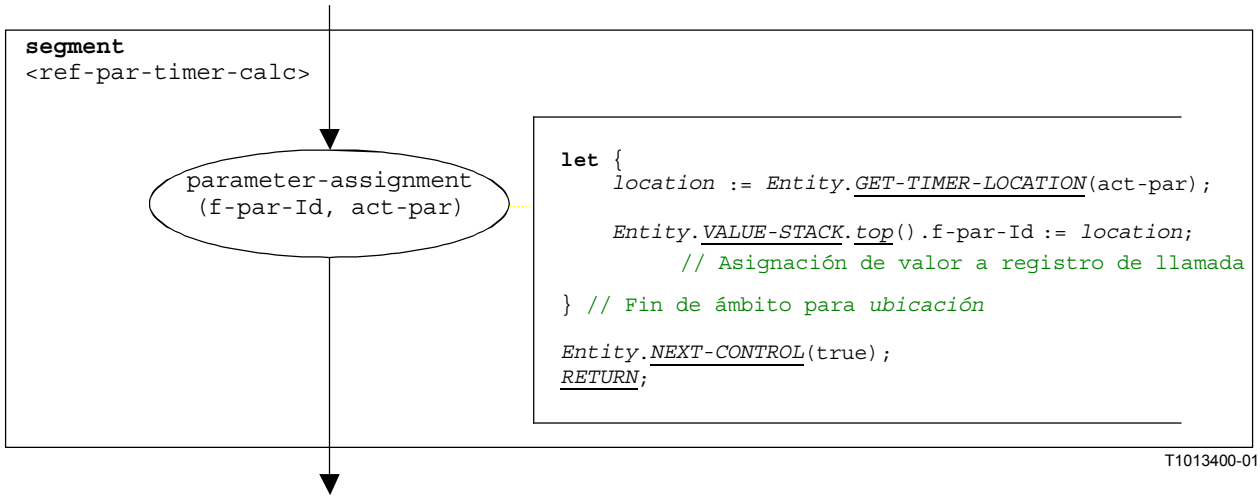


Figura B.72/Z.140 – Segmento de flujograma <ref-par-timer-calc>

B.3.7.26 Segmento de flujograma <parameter-handling>

El segmento de flujograma <parameter-handling> se utiliza en el comienzo de llamadas de funciones. Inicializa un nuevo ámbito y crea variables y temporizadores para el tratamiento de parámetros. Se supone que el registro de llamada de la función llamada está en la cima de la pila de valores.

La ejecución del segmento de flujograma <parameter-handling> se muestra en la figura B.73.

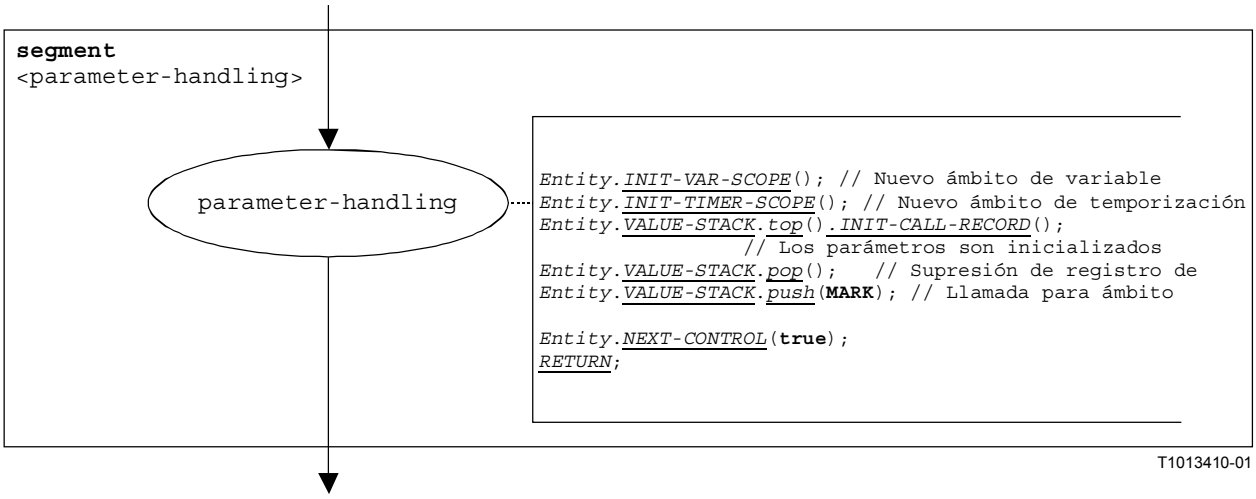


Figura B.73/Z.140 – Segmento de flujograma <parameter-handling>

B.3.7.27 Operación Getcall

La estructura sintáctica de la operación **getcall** es:

```
<portId>.getcall (<matchingSpec>) [from <component_expression>] ->
                                                    [<assignmentPart>]
```

La componente <component_expression> en la cláusula **from** hace referencia al emisor de la llamada que es tratado por la operación **getcall**. Se puede proporcionar en forma de un valor de variable o el valor devuelto de una función, es decir, se supone que es una expresión. <assignmentPart> facultativa indica la asignación de información recibida si la llamada recibida concuerda con la especificación de concordancia <matchingSpec> y con la cláusula **from** (facultativa).

El segmento de flujograma <getcall-op> de la figura B.74 define la ejecución de una operación **getcall**.

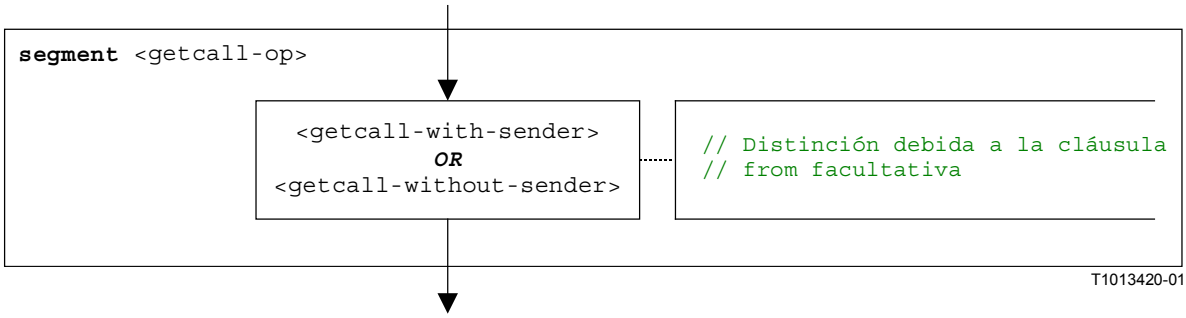


Figura B.74/Z.140 – Segmento de flujograma <getcall-op>

B.3.7.27.1 Segmento de flujograma <getcall-with-sender>

El segmento de flujograma `<getcall-with-sender>` de la figura B.75 define la ejecución de una operación **`getcall`** cuando el emisor se especifica en forma de una expresión.

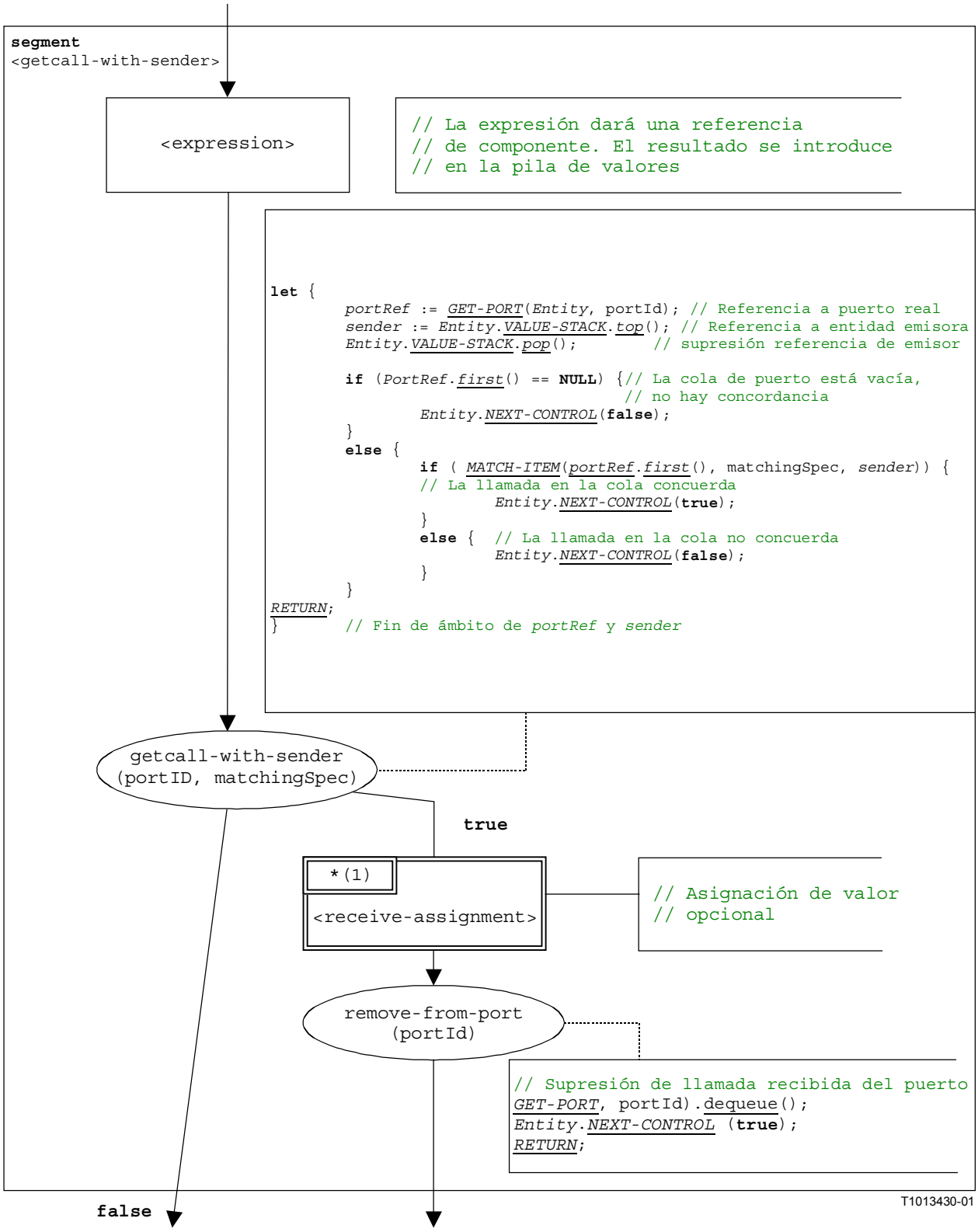


Figura B.75/Z.140 – Segmento de flujograma <getcall-with-sender>

B.3.7.27.2 Segmento de flujograma <getcall-without-sender>

El segmento de flujograma `<getcall-without-sender>` de la figura B.76 define la ejecución de una operación **`getcall`** sin una cláusula **`from`**.

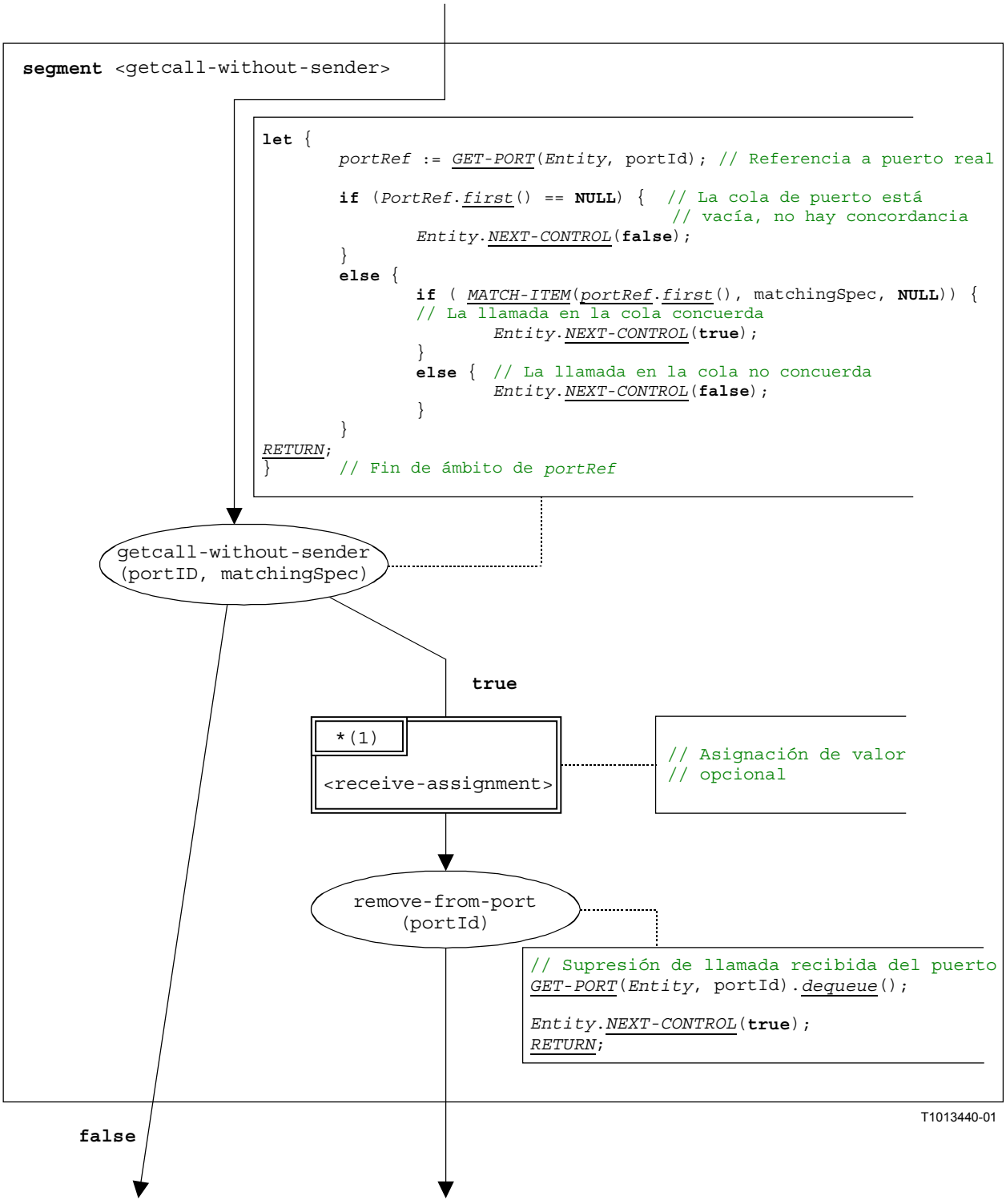


Figura B.76/Z.140 – Segmento de flujograma <getcall-without-sender>

B.3.7.28 Operación Getreply

La estructura sintáctica de la operación **getreply** es:

```
<portId>.getreply (<matchingSpec>) [from <component_expression>] ->
                                                        [<assignmentPart>]
```

La <component_expression> facultativa en la cláusula **from** hace referencia al emisor de la respuesta que es tratado por la operación **getreply**. Se puede proporcionar en forma de un valor de variable o el valor de retorno de una función, es decir, se supone que es una expresión. La <assignmentPart> facultativa indica la asignación de la información recibida si la respuesta concuerda con la especificación de concordancia <matchingSpec> y con la cláusula **from** (facultativa).

El segmento de flujograma <getreply-op> de la figura B.77 define la ejecución de una operación **getreply**.

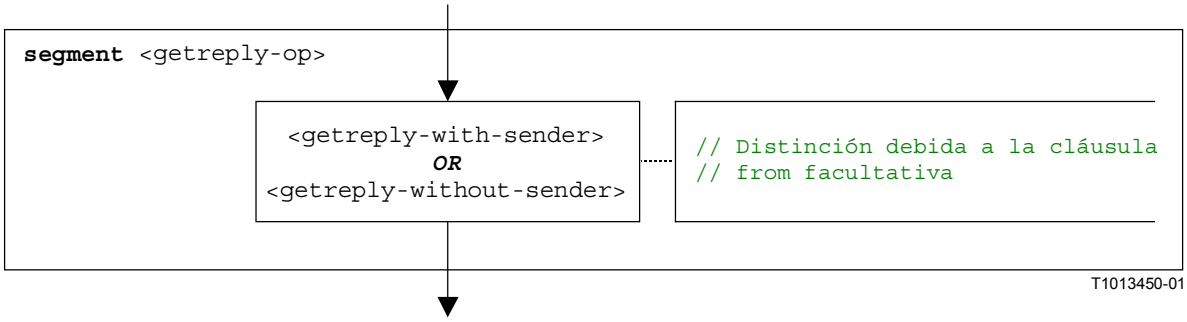


Figura B.77/Z.140 – Segmento de flujograma <getreply-op>

B.3.7.28.1 Segmento de flujograma <getreply-with-sender>

El segmento de flujograma `<getreply-with-sender>` de la figura B.78 define la ejecución de una operación **`getreply`** cuando el emisor se especifica en forma de una expresión.

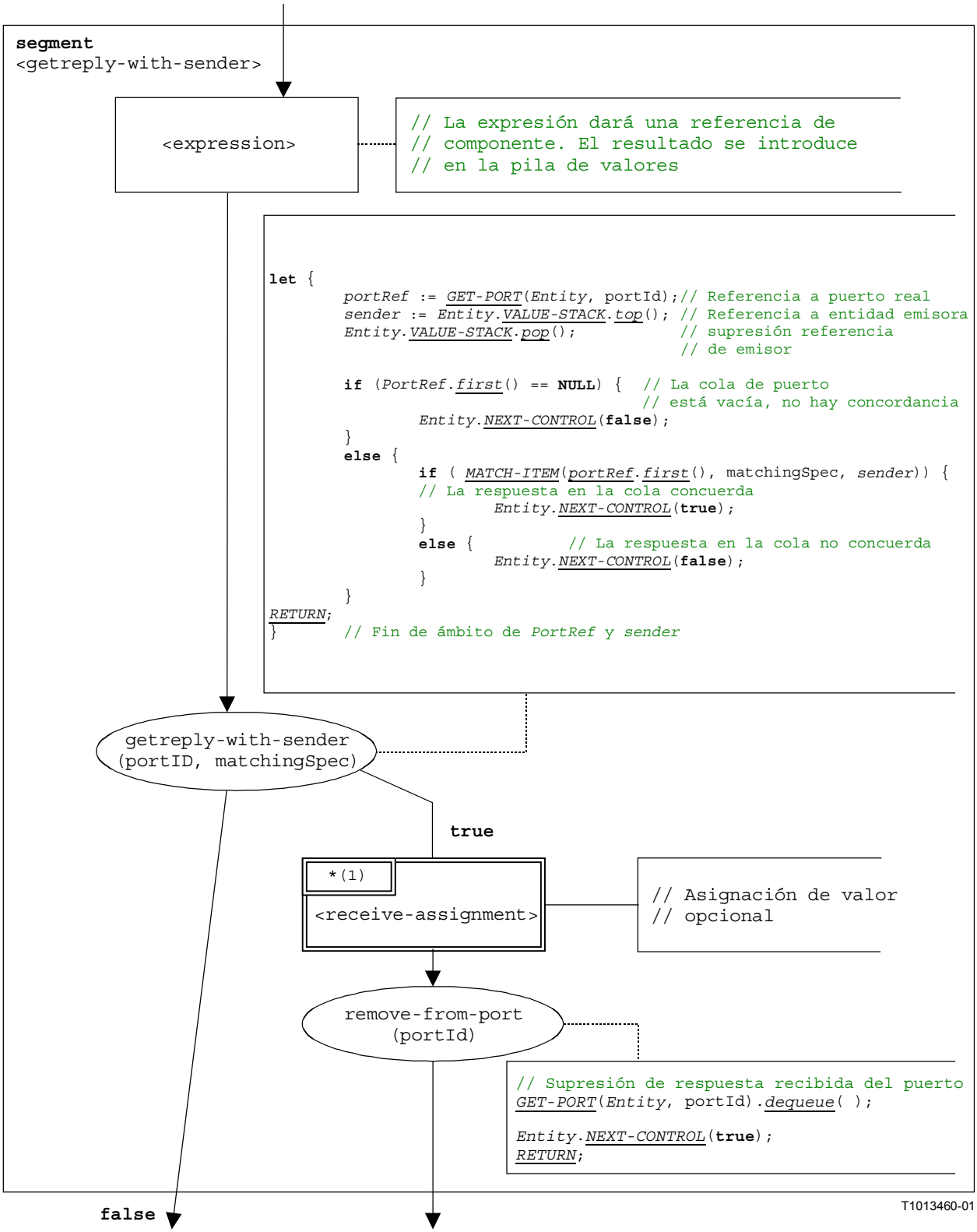


Figura B.78/Z.140 – Segmento de flujograma <getreply-with-sender>

B.3.7.28.2 Segmento de flujograma <getreply-without-sender>

El segmento de flujograma `<getreply-without-sender>` de la figura B.79 define la ejecución de una operación **`getreply`** sin una cláusula **`from`**.

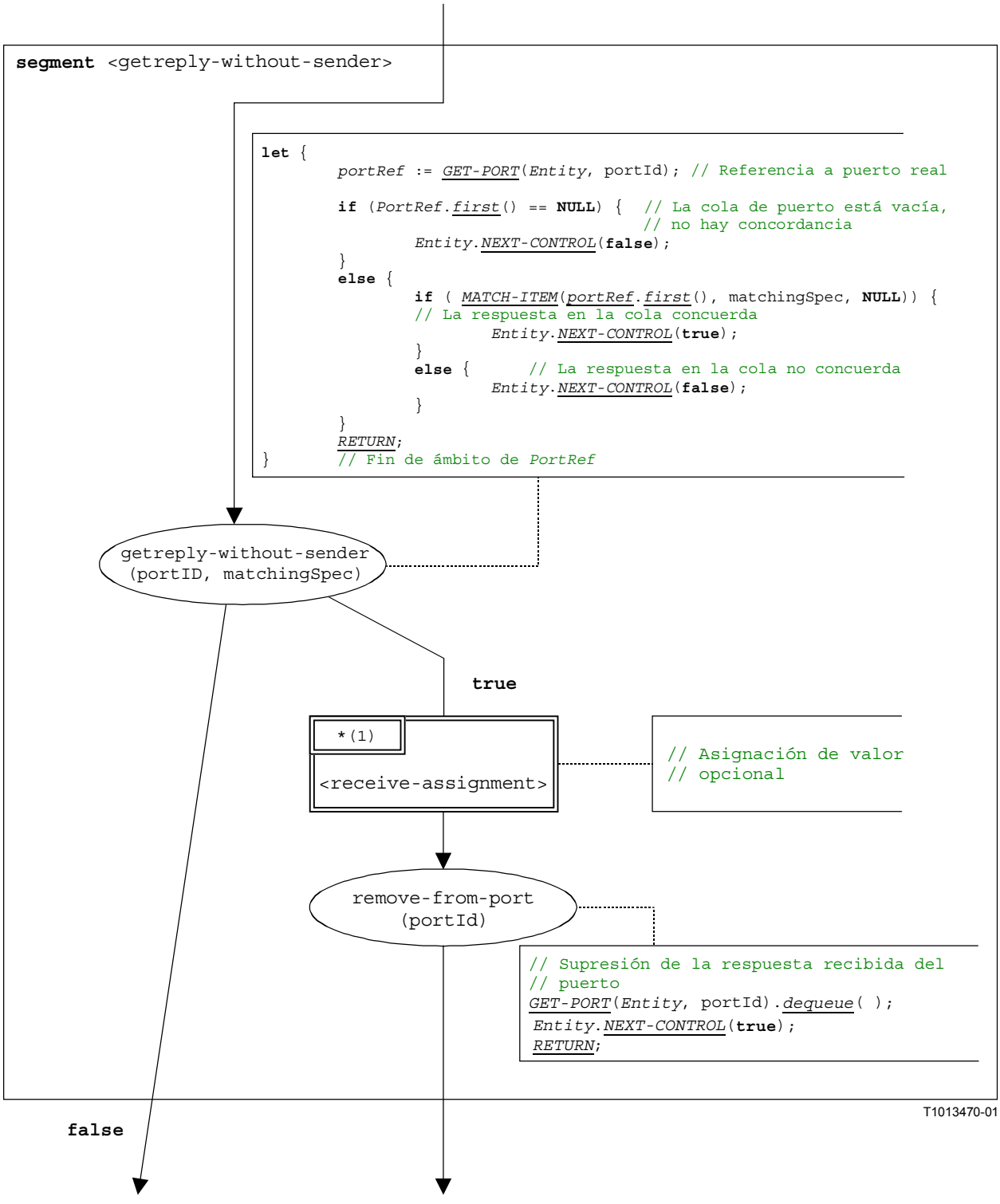


Figura B.79/Z.140 – Segmento de flujograma <getreply-without-sender>

B.3.7.29 Enunciado Goto

La estructura sintáctica del enunciado **goto** es:

```
goto <labelId>
```

El segmento de flujograma <goto-stmt> de la figura B.80 define la ejecución del enunciado **goto**.

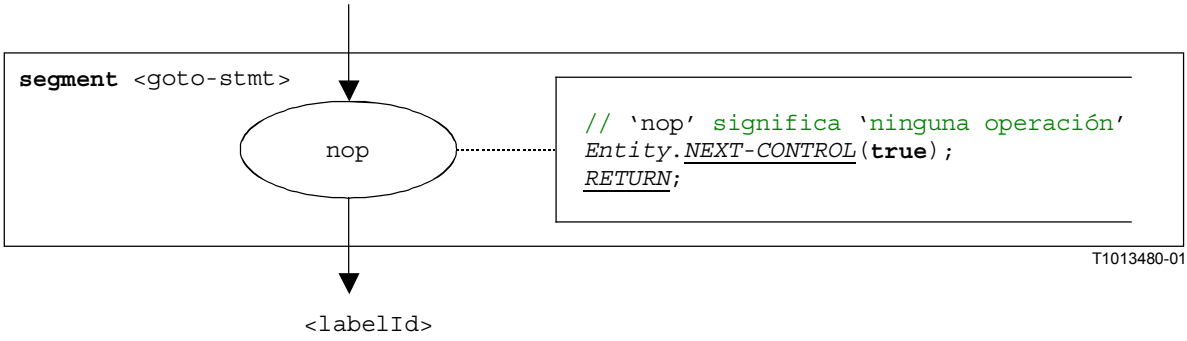


Figura B.80/Z.140 – Segmento de flujograma <goto-stmt>

NOTA – El parámetro <labelId> del enunciado **goto** indica la transferencia de control al lugar en el cual se define una etiqueta <*labelId*> (véase también B.3.7.31).

B.3.7.30 Enunciado If-else

La estructura sintáctica del enunciado **if-else** es:

```
if (<boolean_expression>) <statement-block1>
                [else <statement-block2>]
```

La parte **else** del enunciado **if-else** es facultativa.

El segmento de flujograma <if-else-stmt> en la figura B.81 define la ejecución del enunciado **if-else**.

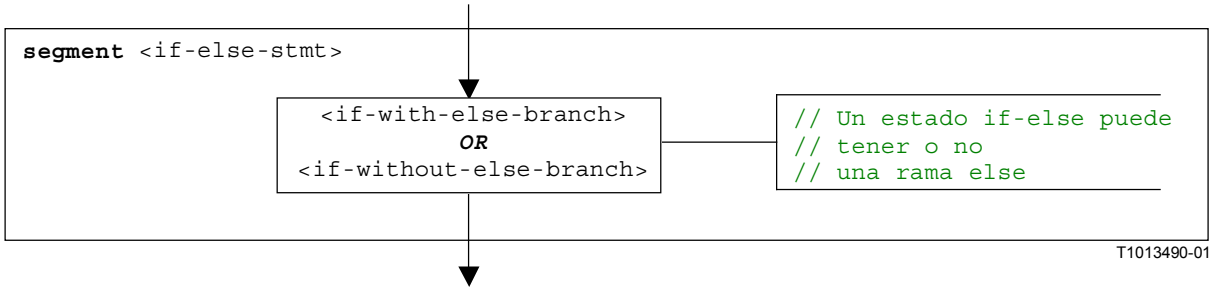


Figura B.81/Z.140 – Segmento de flujograma <if-else-stmt>

B.3.7.30.1 Segmento de flujograma <if-with-else-branch>

La figura B.82 describe la ejecución de un enunciado **if-else** que incluye una rama **else**. El <statement-block> en la rama **true** del nodo de decisión de la figura B.82 corresponde con el <statement-block$_1$> en la estructura sintáctica anterior. El otro <statement-block> corresponde con el <statement-block$_2$> anterior.

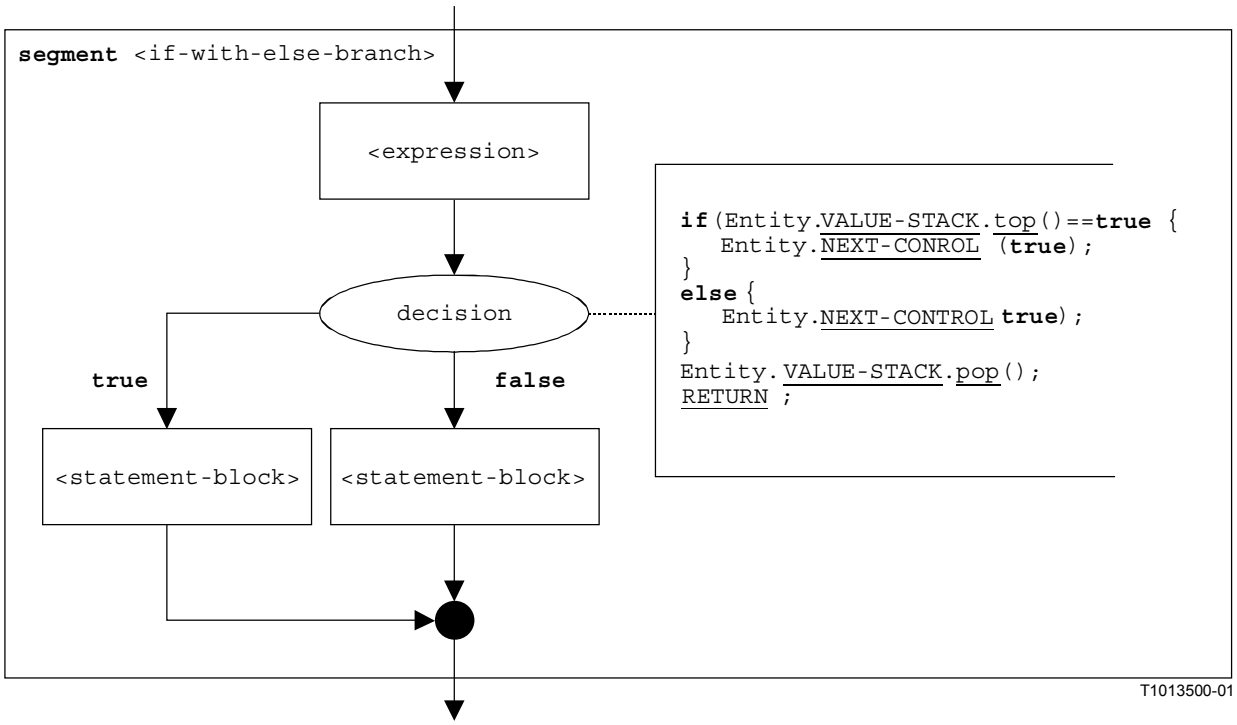


Figura B.82/Z.140 – Segmento de flujograma <if-with-else-branch>

B.3.7.30.2 Segmento de flujograma <if-without-else-branch>

La figura B.83 describe la ejecución de un enunciado **if-else** que no incluye la rama **else**. El <statement-block> en la rama **true** del nodo de decisión de la figura B.82 corresponde con el <statement-block$_1$> en la estructura sintáctica anterior.

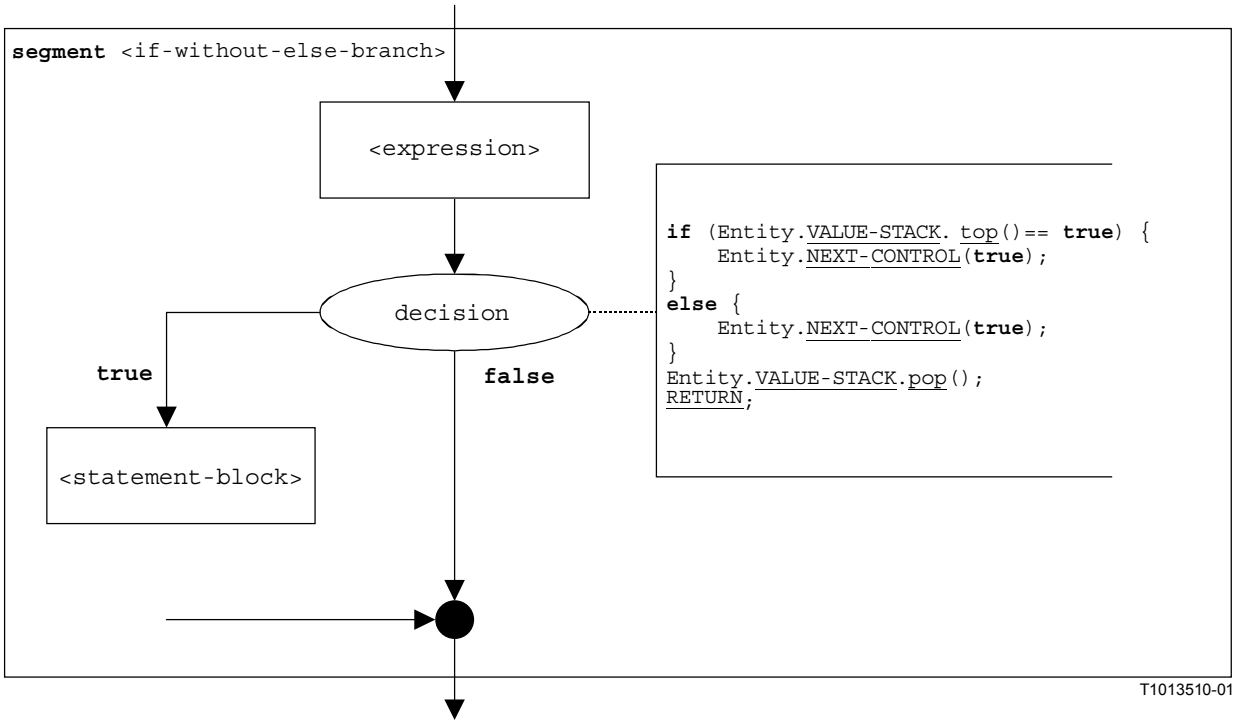


Figura B.83/Z.140 – Segmento de flujograma <if-without-else-branch>

B.3.7.31 Enunciado Label

La estructura sintáctica del enunciado **label** es:

label <labelId>

El segmento de flujograma <label-stmt> de la figura B.84 define la ejecución del enunciado **label**.

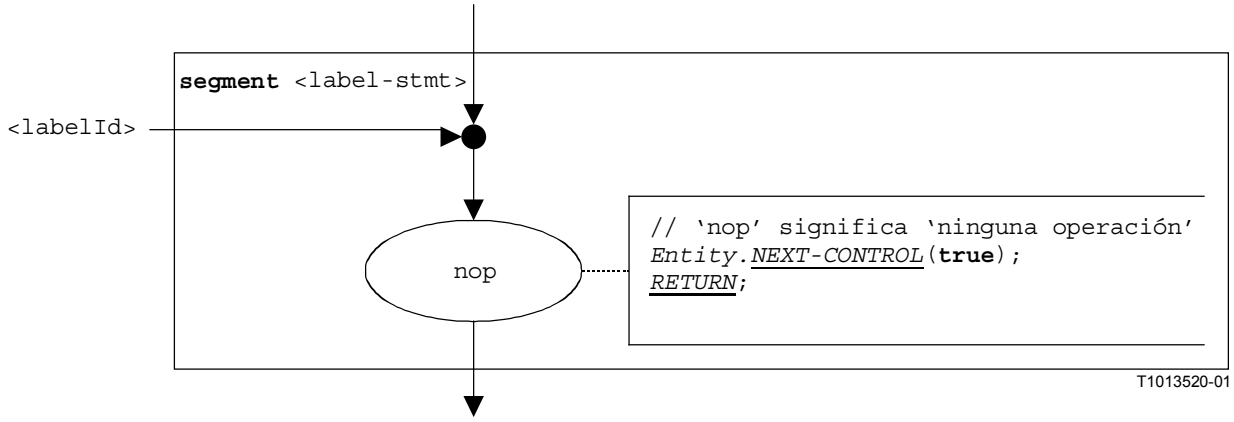


Figura B.84/Z.140 – Segmento de flujograma <label-stmt>

NOTA – El parámetro <labelId> del enunciado **label** indica la posibilidad de que una etiqueta puede ser el objetivo de un salto por medio de un enunciado **goto** (véase también B.3.7.29).

B.3.7.32 Enunciado Log

La estructura sintáctica del enunciado **log** es:

log (<informal-description>)

El segmento de flujograma <log-stmt> de la figura B.85 define la ejecución del enunciado **log**.

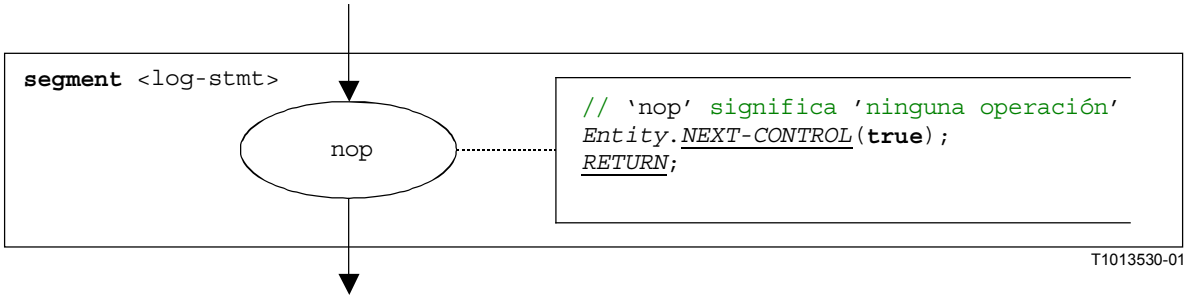


Figura B.85/Z.140 – Segmento de flujograma <log-stmt>

NOTA – El parámetro <informal description> del enunciado **log** no tiene significado para la semántica operacional, por lo que no se representa en el segmento de flujograma.

B.3.7.33 Operación Map

La estructura sintáctica de la operación **map** es:

map(<component_expression>.<portId1>,**system**.<portId2>)

Se considera que los identificadores <portId1> y <portId2> son identificadores de puertos del componente de prueba y de la interfaz de sistema de prueba correspondientes. Los componentes a los cuales pertenece <portId1> son referenciados por medio de la referencia de componente <component_expression>. La referencia puede ser almacenada en variables o es devuelta por una

función. Para simplificar, se considera que es una expresión que da una referencia de componente. Por tanto, la pila de valores se utiliza para almacenar la referencia de componentes.

NOTA – La operación **map** no tiene en cuenta si el enunciado **system**.<portId> aparece como primero o como segundo parámetro. Para simplificar, se supone que es siempre el segundo parámetro.

La ejecución de la operación **map** se define mediante el segmento de flujograma <map-op> mostrado en la figura B.86.

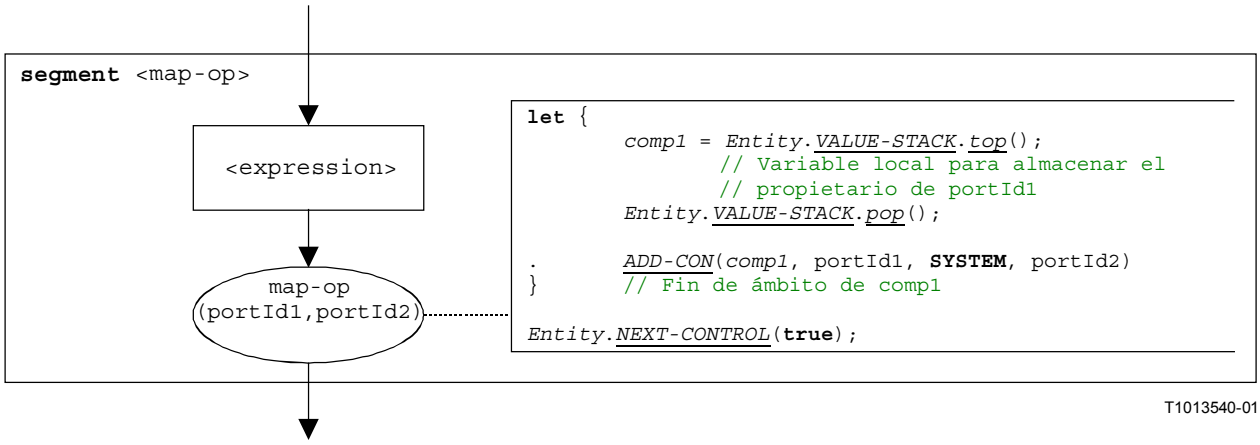


Figura B.86/Z.140 – Segmento de flujograma <map-op>

B.3.7.34 Operación MTC

La estructura sintáctica de la operación **mtc** es:

mtc

El segmento de flujograma <mtc-op> de la figura B.87 define la ejecución de la operación **mtc**.

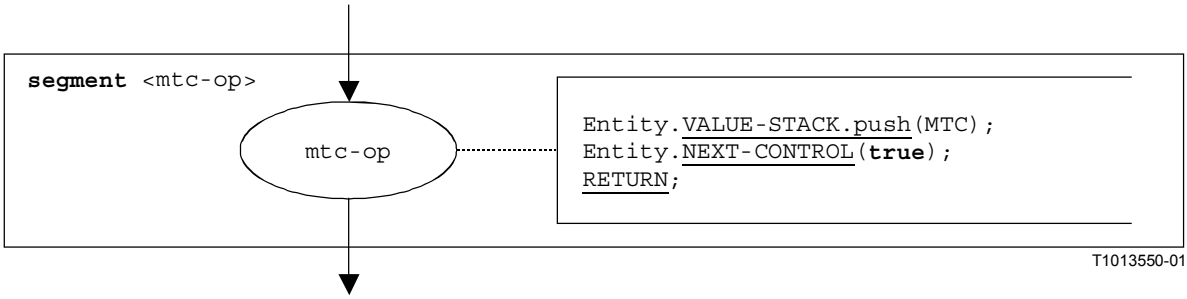


Figura B.87/Z.140 – Segmento de flujograma <mtc-op>

B.3.7.35 Operación Raise

La estructura sintáctica de la operación **raise** es:

<portId>.**raise** (<exceptSpec>) [**to** <component_expression>]

La <component_expression> facultativa en la cláusula **to** hace referencia a la entidad receptora. Puede ser proporcionada en forma de un valor de variable o el valor de retorno de una función.

El segmento de flujograma <raise-op> de la figura B.88 define la ejecución de una operación **raise**.

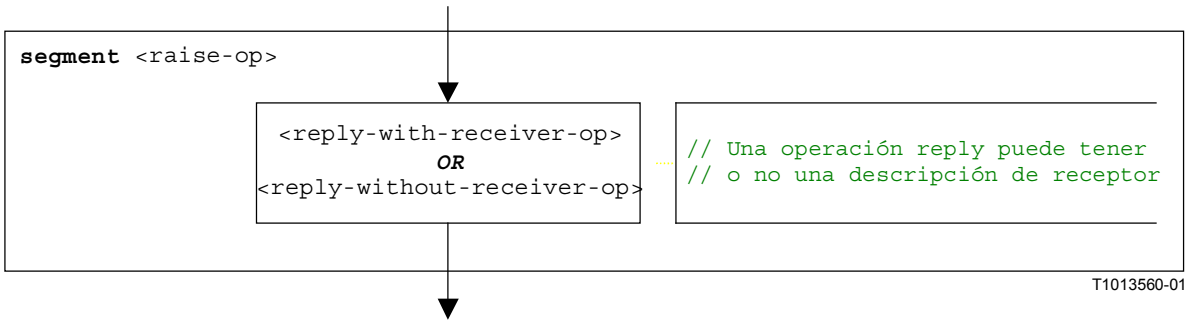


Figura B.88/Z.140 – Segmento de flujograma <raise-op>

B.3.7.35.1 Segmento de flujograma <raise-with-receiver-op>

El segmento de flujograma `<raise-with-receiver-op>` de la figura B.89 define la ejecución de una operación **`raise`** cuando el receptor se especifica en forma de una expresión.

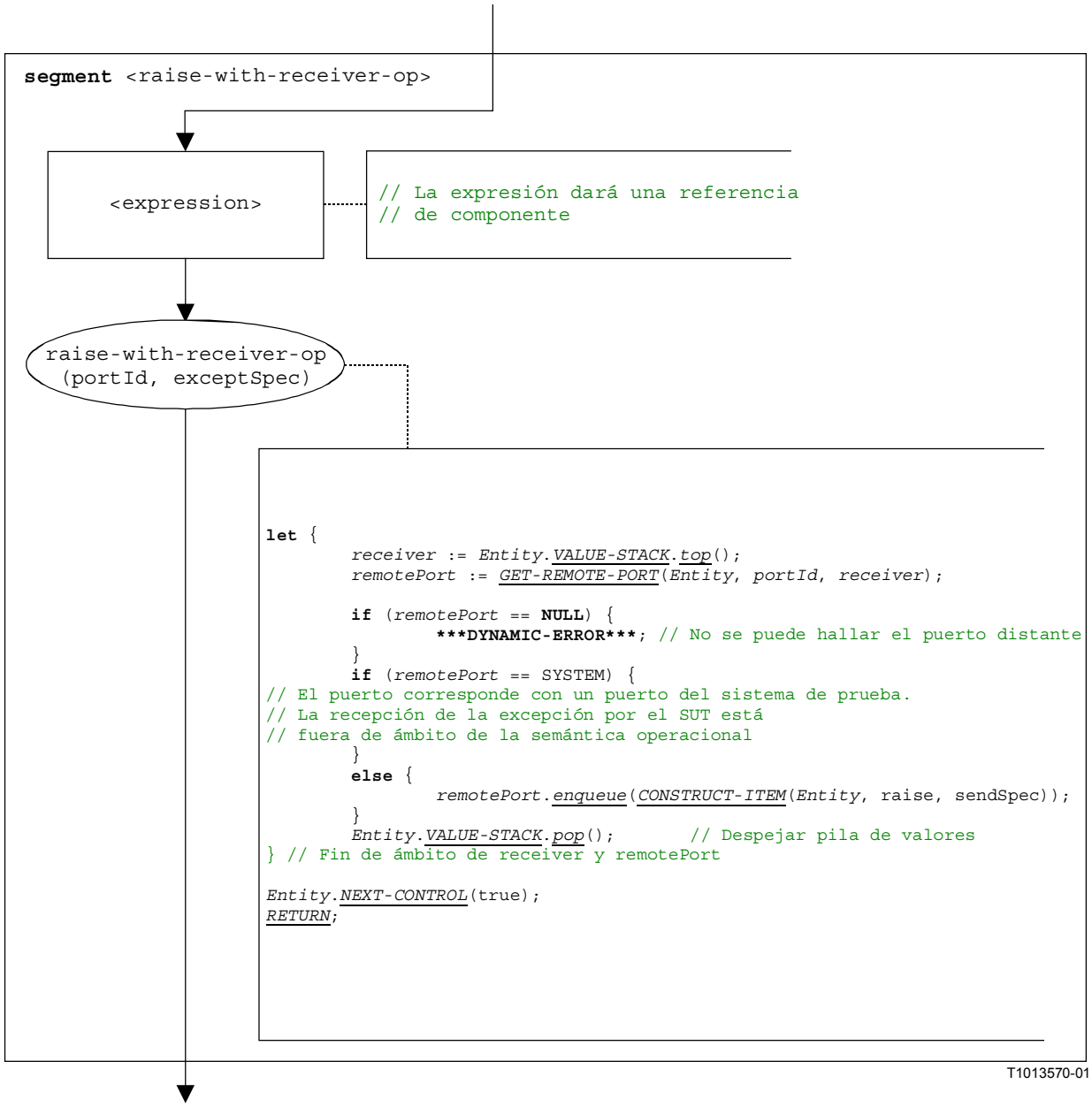


Figura B.89/Z.140 – Segmento de flujograma <raise-with-receiver-op>

B.3.7.35.2 Segmento de flujograma <raise-without-receiver-op>

El segmento de flujograma <raise-without-receiver-op> de la figura B.90 define la ejecución de una operación **raise** sin la cláusula **to**.

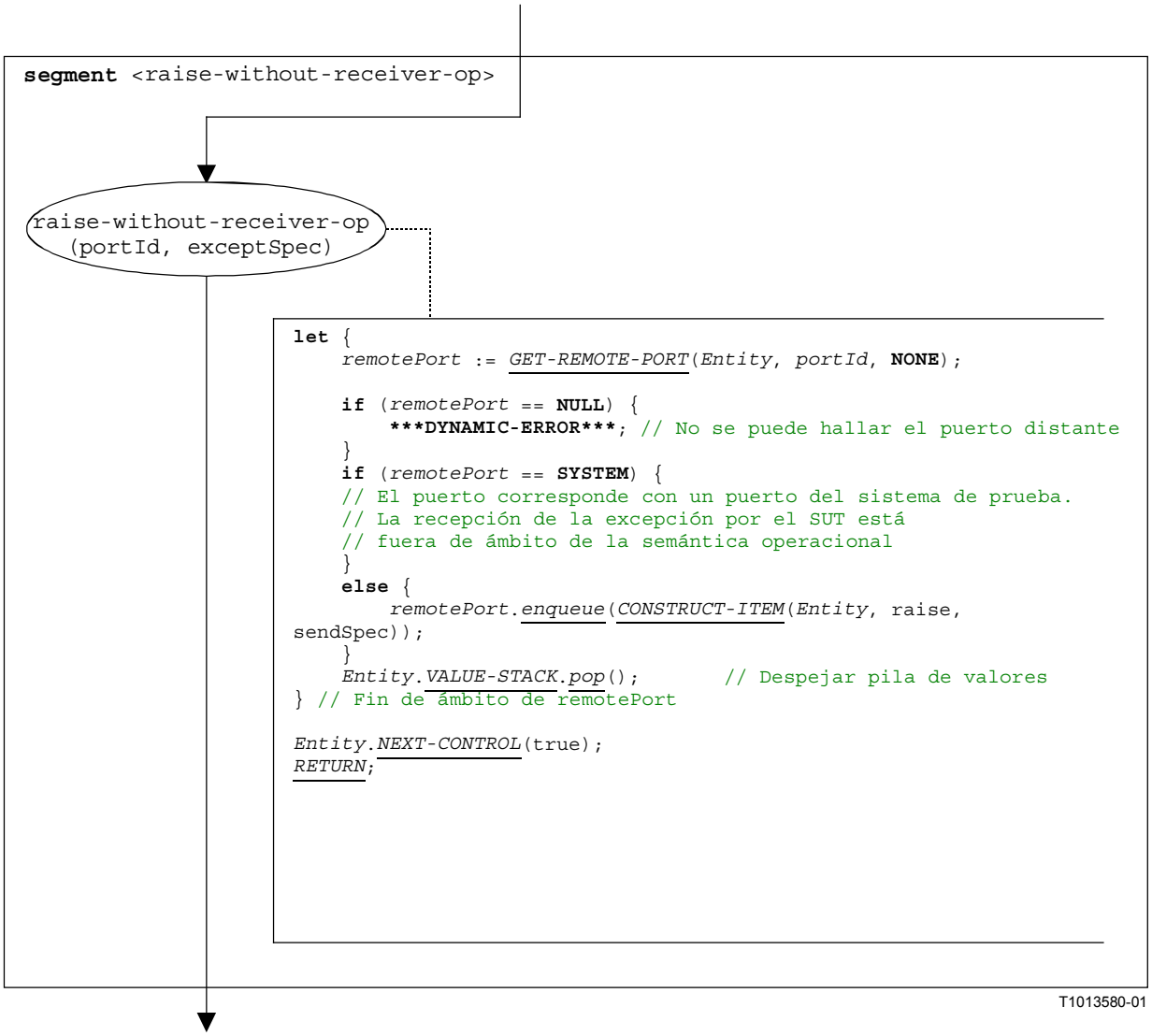


Figura B.90/Z.140 – Segmento de flujograma <raise-without-receiver-op>

B.3.7.36 Operación Read timer

La estructura sintáctica de la operación **read** timer es:

<timerId>**.read**

El segmento de flujograma <read-timer-op> de la figura B.91 define la ejecución de la operación **read** timer.

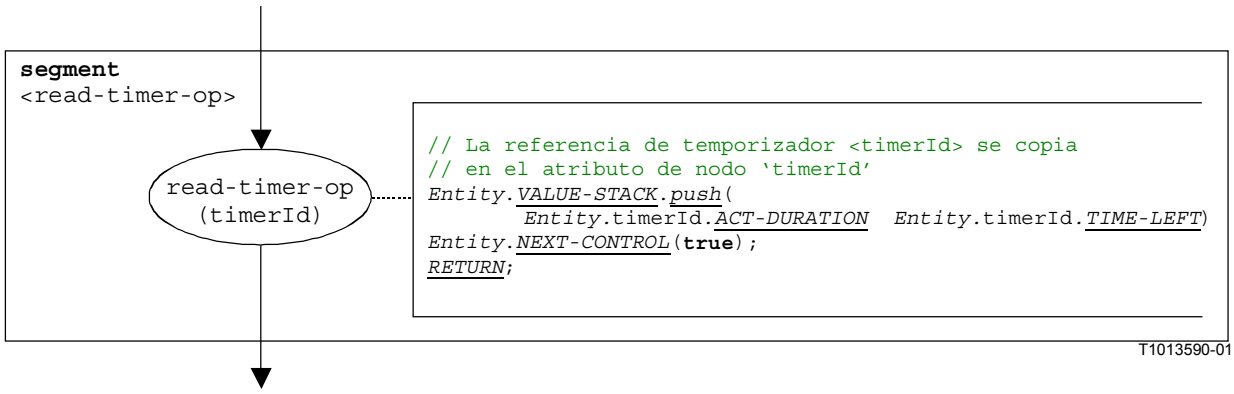


Figura B.91/Z.140 – Segmento de flujograma <read-timer-op>

B.3.7.37 Operación Receive

La estructura sintáctica de la operación **receive** es:

```
<portId>.receive (<matchingSpec>) [from <component_expression>] ->
                                                    [<assignmentPart>]
```

La <component_expression> facultativa en la cláusula **from** hace referencia a la entidad emisora. Se puede proporcionar en forma de un valor de variable o el valor de retorno de una función, es decir, se supone que es una expresión. La <assignmentPart> facultativa indica la asignación de información recibida si el mensaje recibido concuerda con la especificación de concordancia <matchingSpec> y con la cláusula **from** (facultativa).

El segmento de flujograma <receive-op> de la figura B.92 define la ejecución de una operación **receive**.

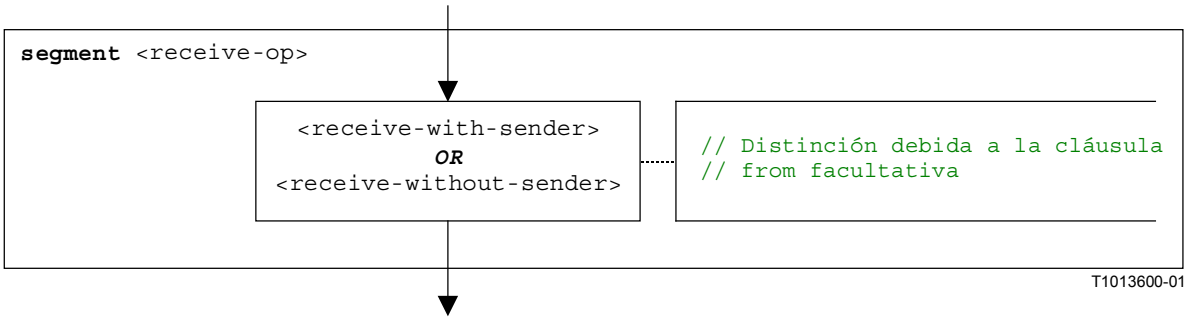


Figura B.92/Z.140 – Segmento de flujograma <receive-op>

B.3.7.37.1 Segmento de flujograma <receive-with-sender>

El segmento de flujograma `<receive-with-sender>` de la figura B.93 define la ejecución de una operación **`receive`** cuando el emisor se especifica en forma de una expresión.

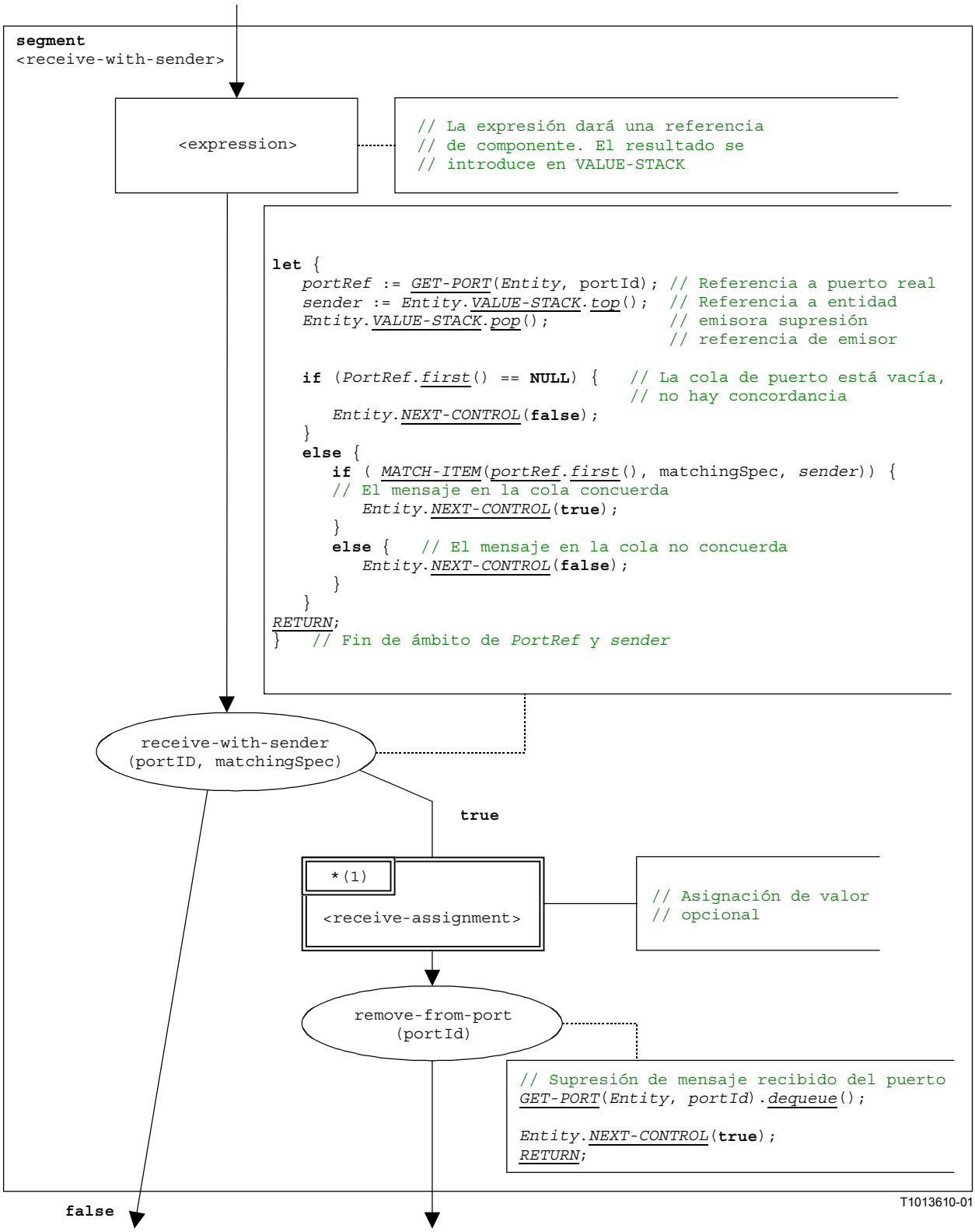


Figura B.93/Z.140 – Segmento de flujograma <receive-with-sender>

B.3.7.37.2 Segmento de flujograma <receive-without-sender>

El segmento de flujograma `<receive-without-sender>` de la figura B.94 define la ejecución de una operación **`receive`** sin una cláusula **`from`**.

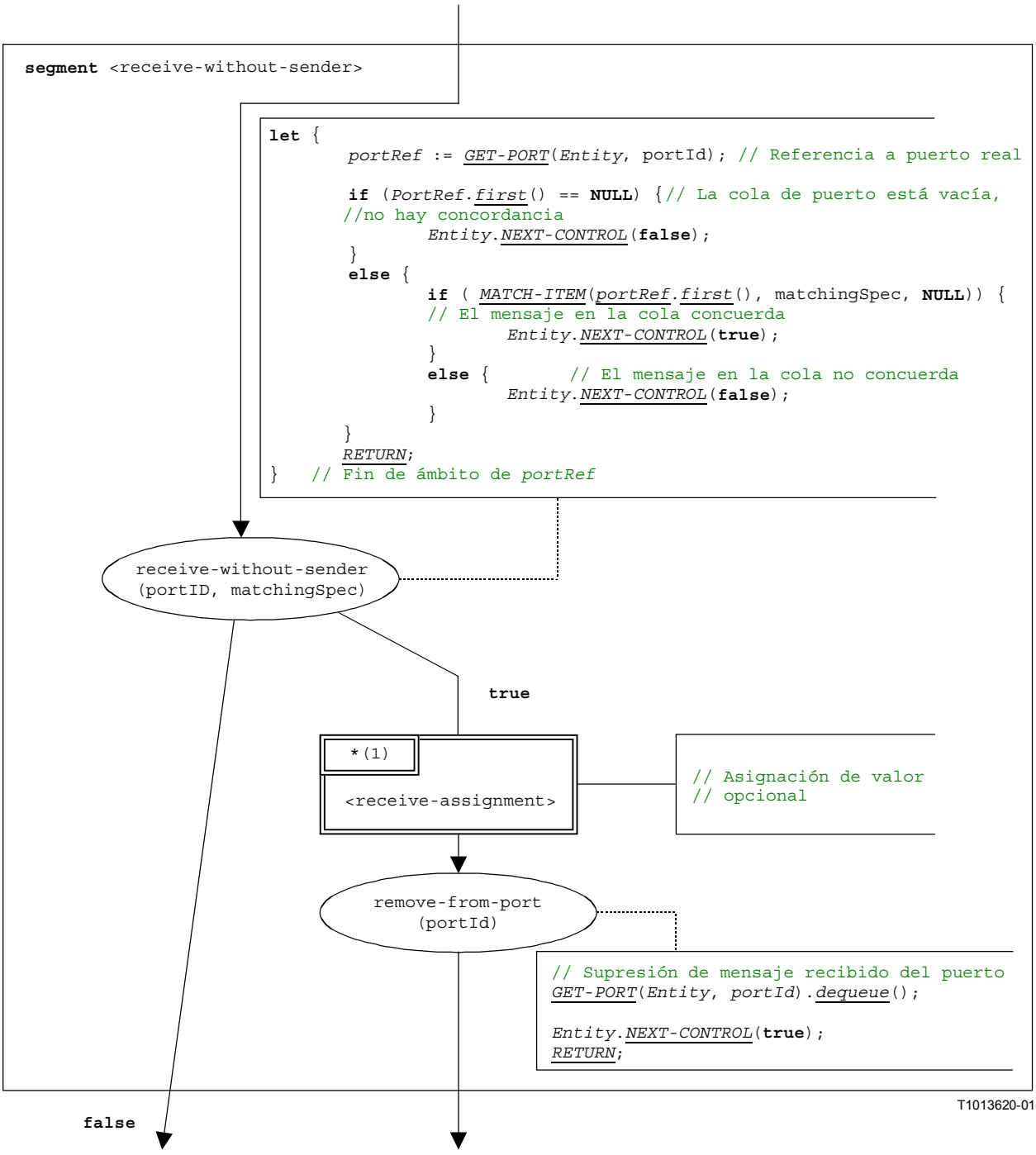


Figura B.94/Z.140 – Segmento de flujograma <receive-without-sender>

B.3.7.37.3 Segmento de flujograma <receive-assignment>

El segmento de flujograma `<receive-assignment>` de la figura B.95 define la extracción de información de mensajes recibidos y su asignación a variables.

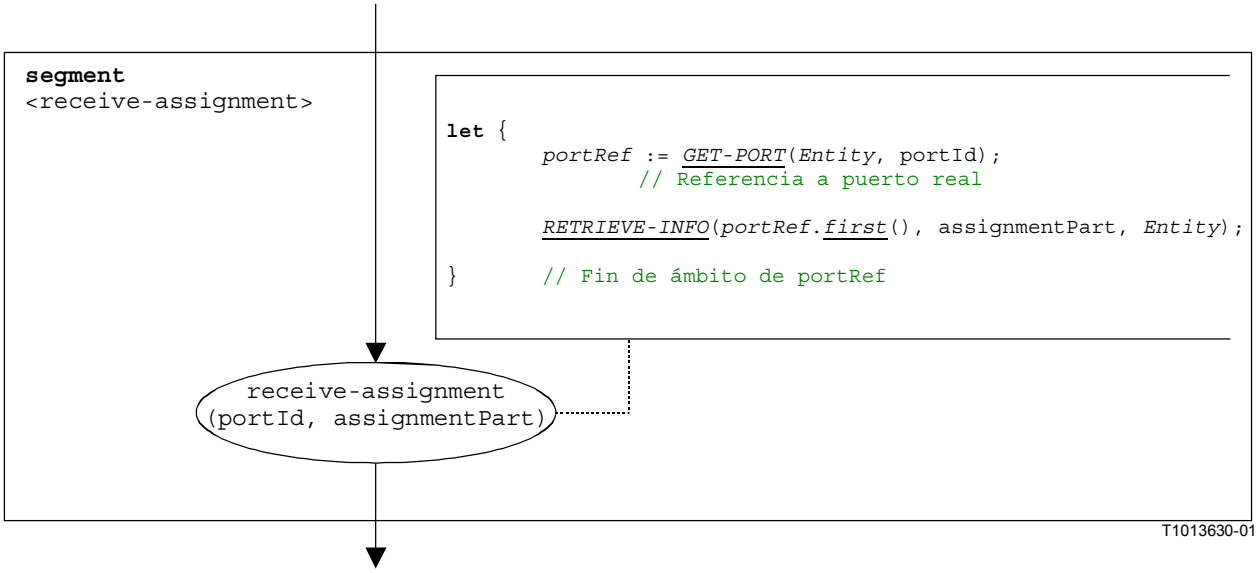


Figura B.95/Z.140 – Segmento de flujograma <receive-assignment>

B.3.7.38 Operación Reply

La estructura sintáctica de la operación **`reply`** es:

`<portId>.`**`reply`** `(<replySpec>) [`**`to`** `<component_expression>]`

La `<component_expression>` facultativa en la cláusula **`to`** hace referencia a la entidad receptora. Se puede proporcionar en forma de un valor de variable o el valor de retorno de una función.

El segmento de flujograma `<reply-op>` de la figura B.96 define la ejecución de una operación **`reply`**.

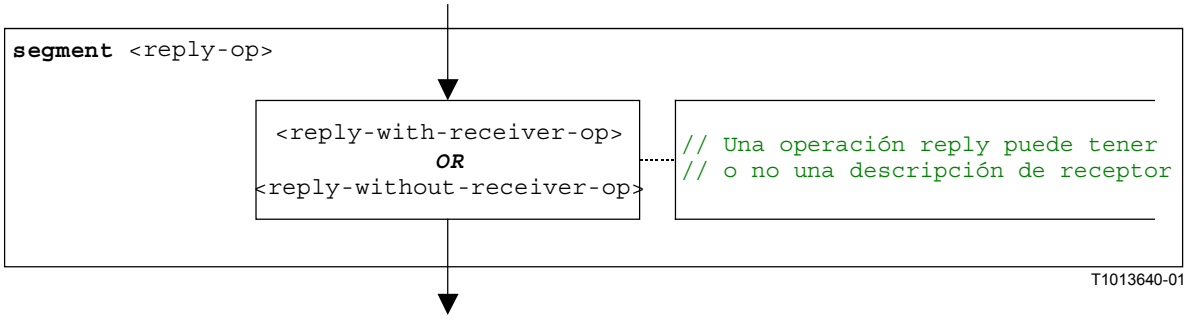


Figura B.96/Z.140 – Segmento de flujograma <reply-op>

B.3.7.38.1 Segmento de flujograma <reply-with-receiver-op>

El segmento de flujograma `<reply-with-receiver-op>` de la figura B.97 define la ejecución de una operación **`reply`** cuando el receptor se especifica en forma de una expresión.

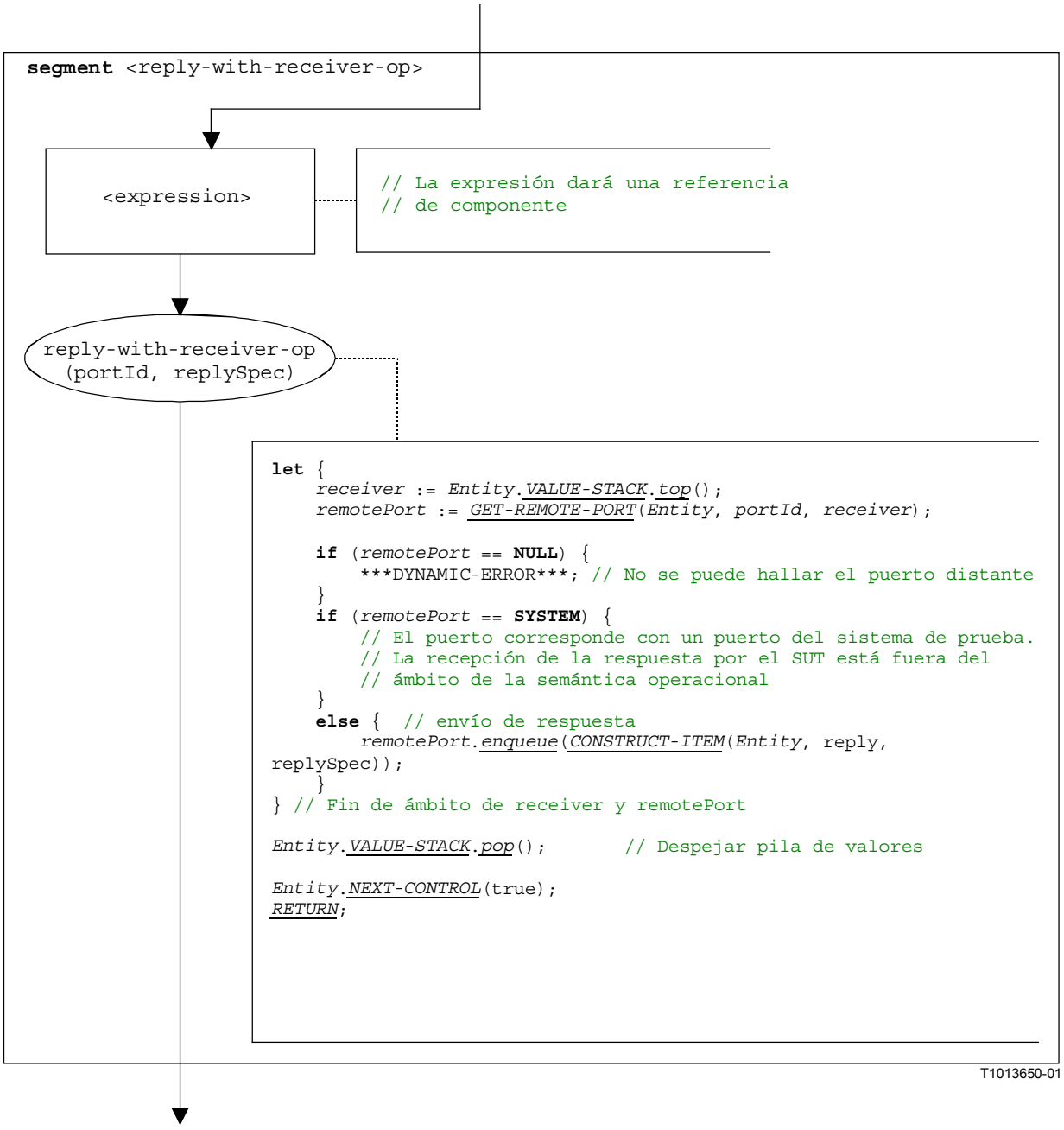


Figura B.97/Z.140 – Segmento de flujograma <reply-with-receiver-op>

B.3.7.38.2 Segmento de flujograma <reply-without-receiver-op>

El segmento de flujograma `<reply-without-receiver-op>` de la figura B.98 define la ejecución de una operación **`reply`** sin la cláusula **`to`**.

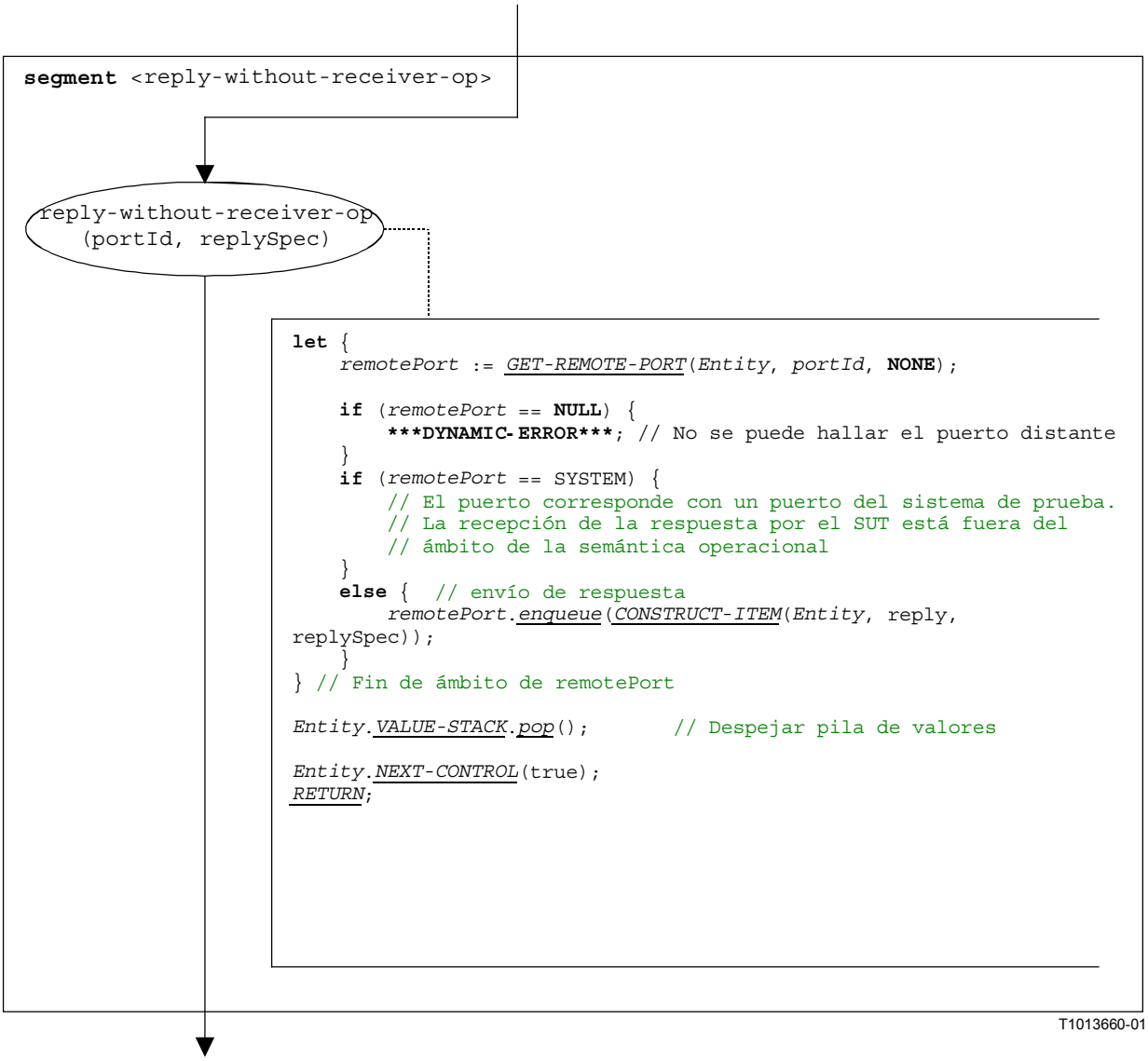


Figura B.98/Z.140 – Segmento de flujograma <reply-without-receiver-op>

B.3.7.39 Enunciado Return

La estructura sintáctica del enunciado **`return`** es:

```
return [<expression>]
```

La `<expression>` facultativa describe un posible valor devuelto de una función. La ejecución de un enunciado **`return`** significa que el control deja la unidad de ámbito real, es decir, las variables y los temporizadores sólo conocidos en este ámbito tienen que ser suprimidos y la pila de valores tiene que ser actualizada. El enunciado **`return`** tiene el efecto de una operación **`stop`**, si es el último enunciado en una descripción de comportamiento.

NOTA – Debido a la sustitución de notaciones abreviadas, los casos de prueba y el control de módulo terminarán siempre con una operación **`stop`**. Sólo otros componentes de prueba pueden terminar con un enunciado **`return`**.

El segmento de flujograma `<return-stmt>` de la figura B.99 define la ejecución de un enunciado **`return`**.

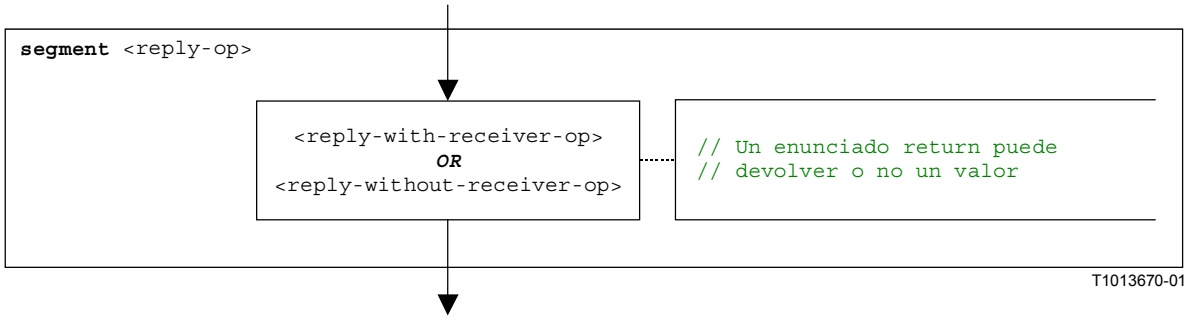


Figura B.99/Z.140 – Segmento de flujograma <return-stmt>

B.3.7.39.1 Segmento de flujograma <return-with-value>

El segmento de flujograma `<return-with-value>` de la figura B.100 define la ejecución de un enunciado **`return`** que devuelve un valor especificado en forma de una expresión.

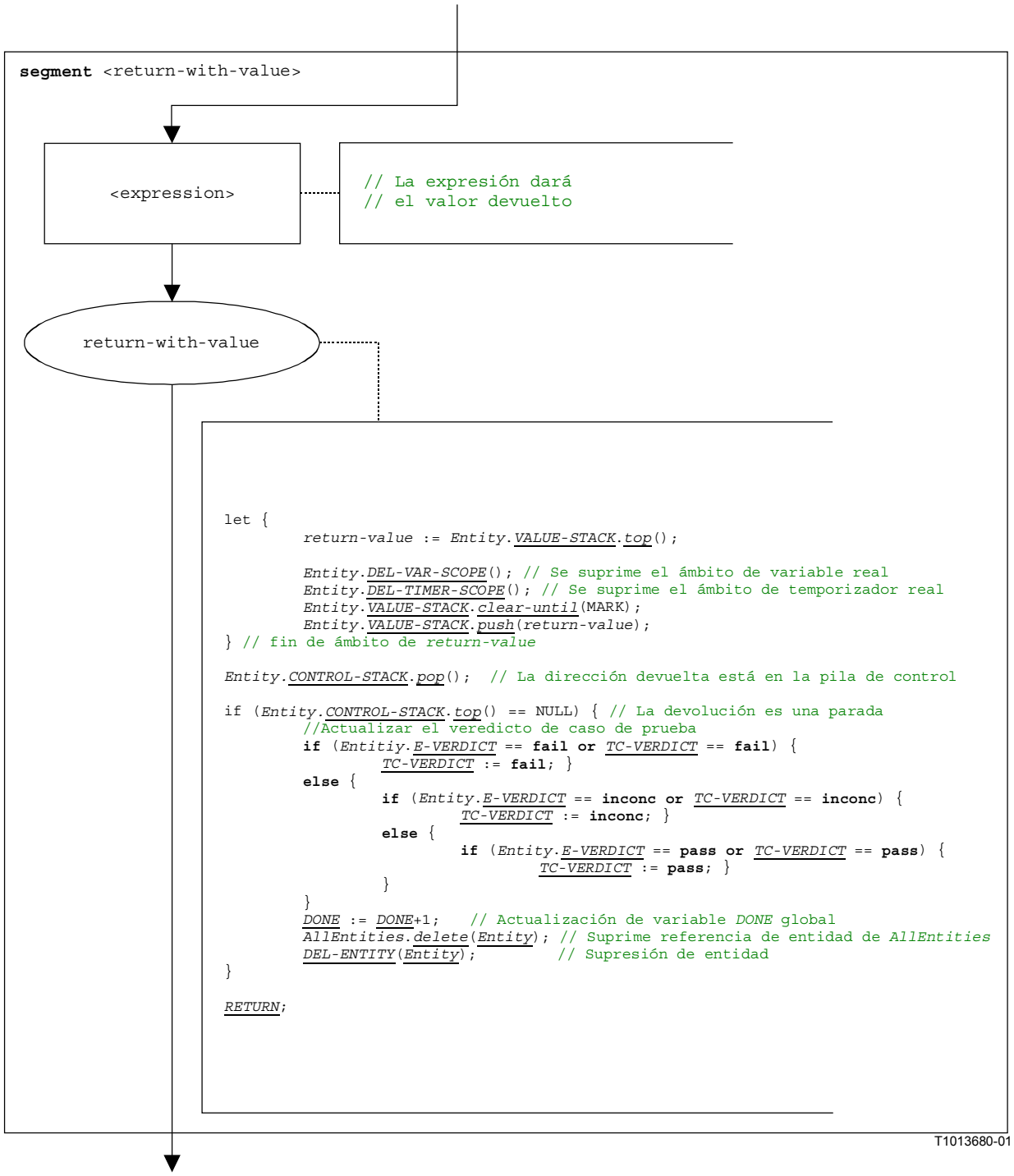


Figura B.100/Z.140 – Segmento de flujograma <return-with-value>

B.3.7.39.2 Segmento de flujograma <return-without-value>

El segmento de flujograma <return-without-value> de la figura B.101 define la ejecución de un enunciado **return** que no devuelve ningún valor.

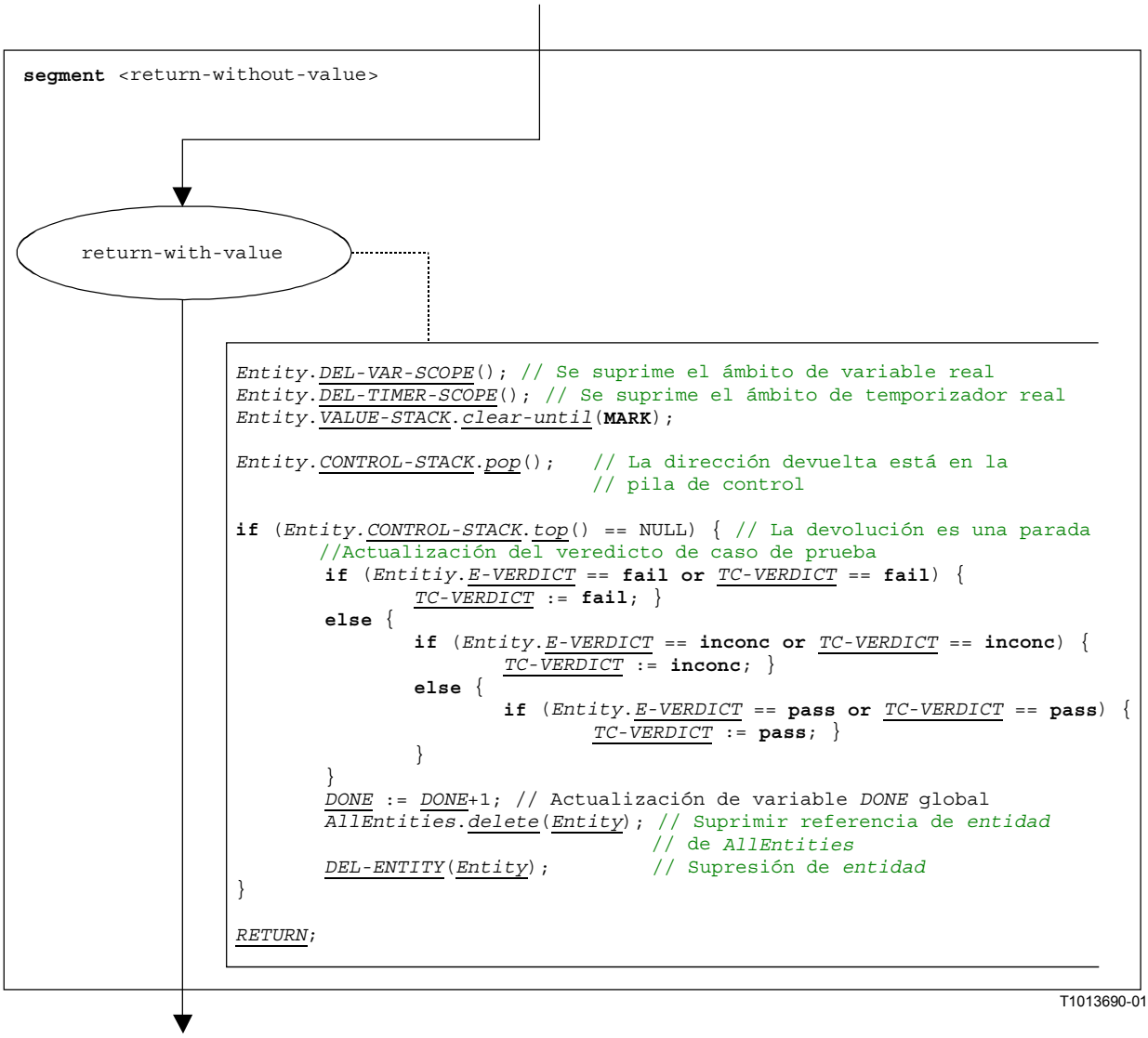


Figura B.101/Z.140 – Segmento de flujograma <return-without-value>

B.3.7.40 Operación Running-all-components

La operación **running-all-components** hace referencia a la utilización de las palabras clave **all component** en la operación **running** (cláusula 42). La operación **running-all-components** sólo puede ser invocada por el **mtc**. Permite comprobar si están funcionando todos los componentes de prueba paralelos de un caso de prueba. La estructura sintáctica de la operación **running-all-components** es:

```
all component.running
```

La ejecución de la operación **running-all-components** se define mediante el segmento de flujograma <running-all-comp-op> de la figura B.102.

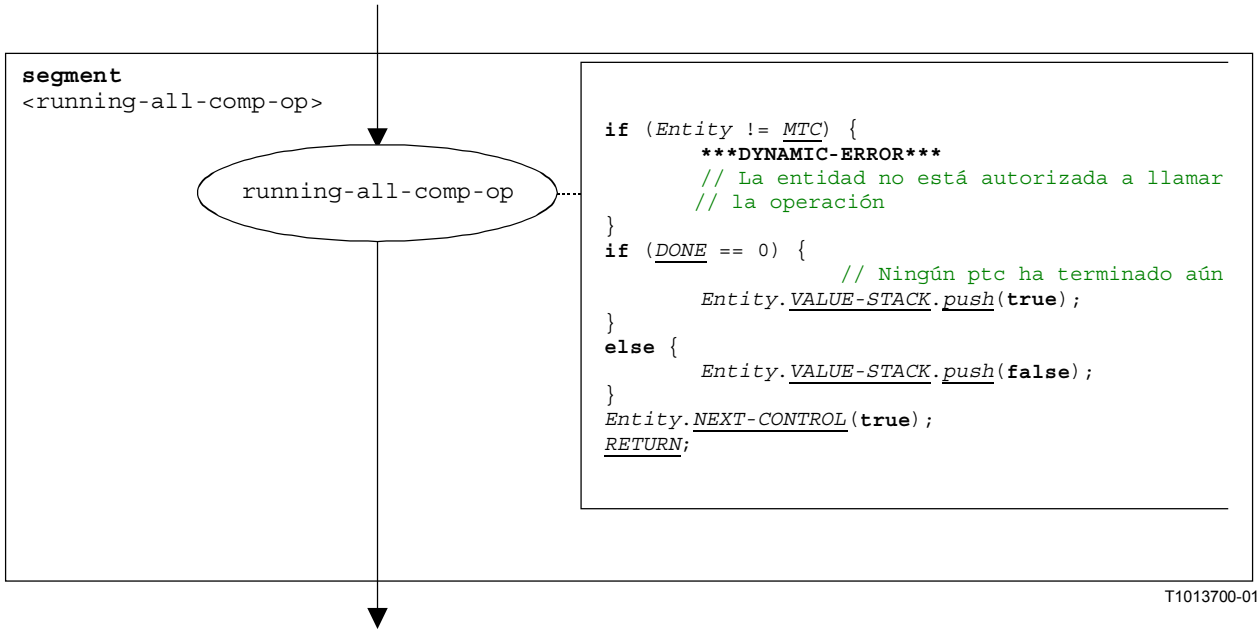


Figura B.102/Z.140 – Segmento de flujograma <running-all-comp-op>

B.3.7.41 Operación Running-any-component

La operación **running-any-component** hace referencia al uso de las palabras clave **any component** en la operación **running** (cláusula 42). La operación **running-any-components** sólo puede ser invocada por el **mtc**. Permite comprobar si está aún funcionando por lo menos un componente de prueba paralelo de un caso de prueba. La estructura sintáctica de la operación **running-any-components** es:

any component.running

La ejecución de la operación **running-any-components** se define mediante el segmento de flujograma <running-any-comp-op> de la figura B.103.

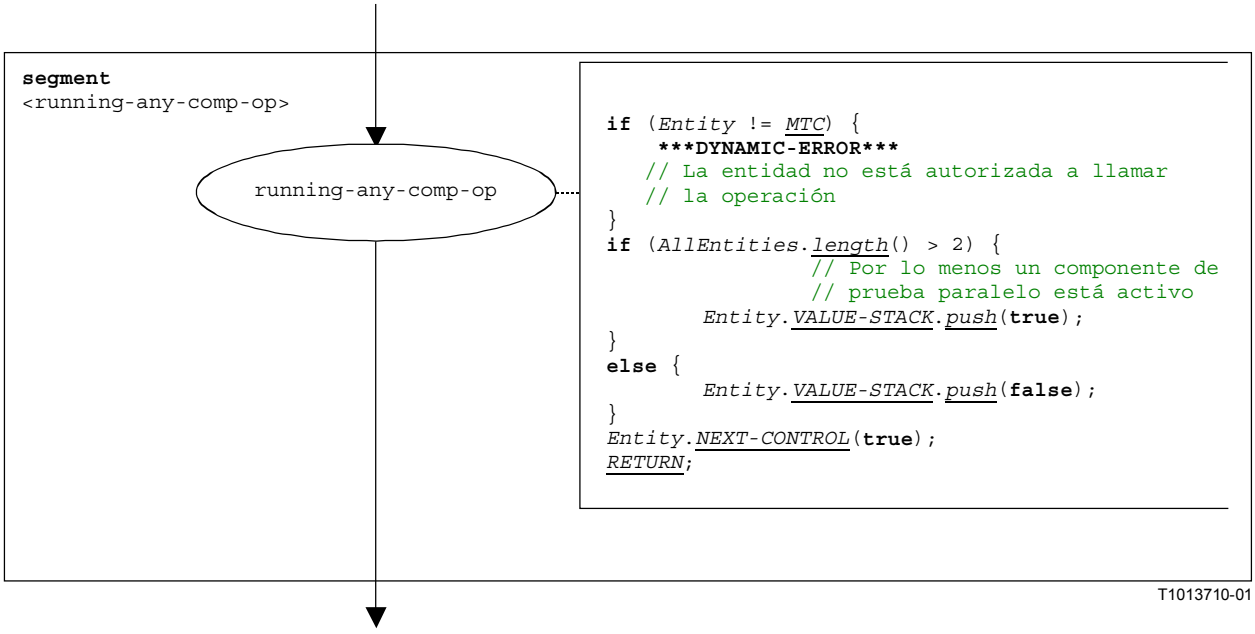


Figura B.103/Z.140 – Segmento de flujograma <running-any-comp-op>

B.3.7.42 Operación Running component

La estructura sintáctica de la operación **running** es:

```
<component_expression>.running
```

La operación **running component** comprueba si un componente está en funcionamiento o detenido. La utilización de una referencia de componente identifica el componente que ha de ser comprobado. La referencia puede ser almacenada en una variable o ser devuelta por una función. Para simplificar, se considera que ésta es una expresión que da una referencia de componente.

El segmento de flujograma <running-component-op> de la figura B.104 define la ejecución de la operación **running**.

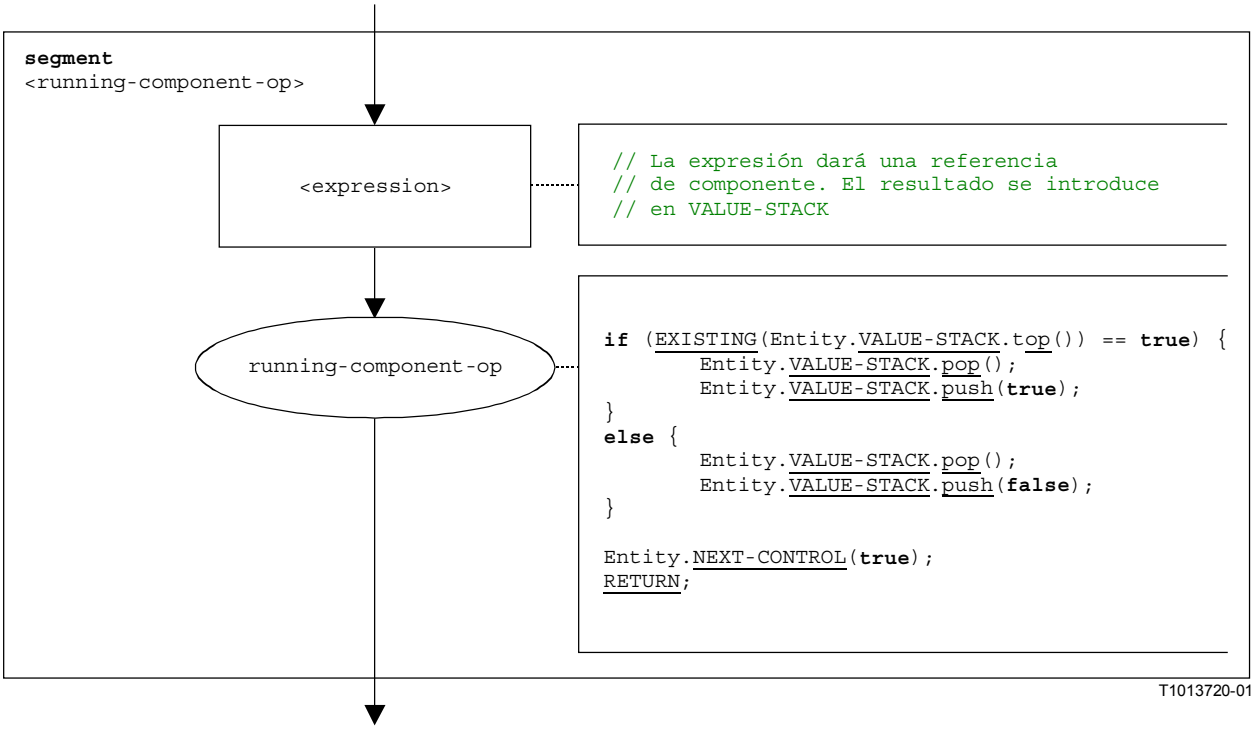


Figura B.104/Z.140 – Segmento de flujograma <running-component-op>

B.3.7.43 Operación Running timer

La estructura sintáctica de la operación **running** timer es:

```
<timerId>.running
```

El segmento de flujograma <running-timer-op> de la figura B.105 define la ejecución de la operación **running**.

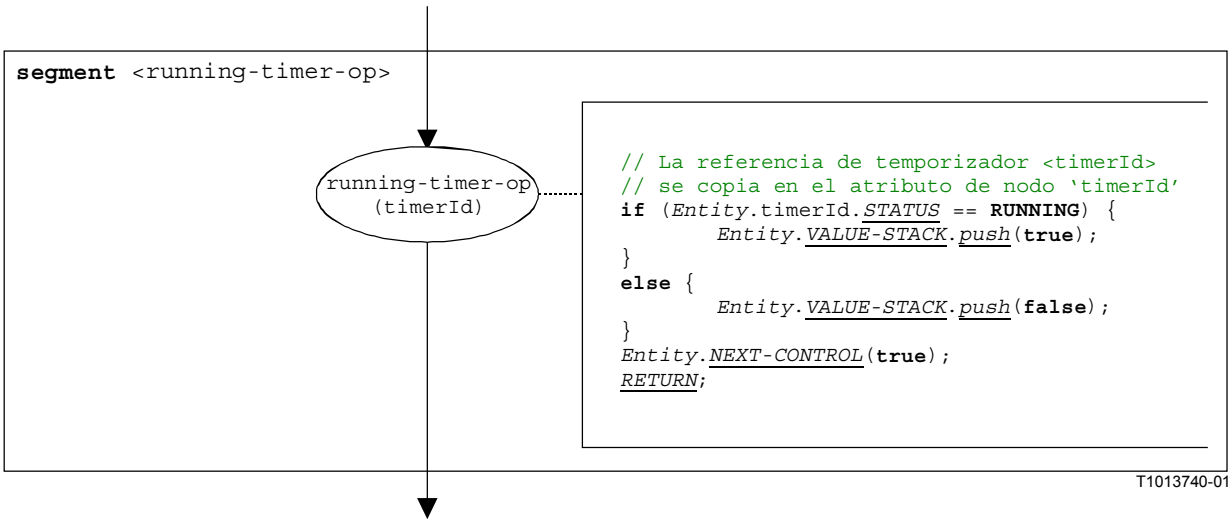


Figura B.105/Z.140 – Segmento de flujograma <running-timer-op>

B.3.7.44 Operación Send

La estructura sintáctica de la operación **send** es:

<portId>.**send** (<send-spec>) [**to** <component_expression>]

La <component_expression> facultativa en la cláusula **to** hace referencia a la entidad receptora. Puede ser proporcionada en forma de una valor de variable o el valor devuelto de una función.

El segmento de flujograma <send-op> de la figura B-106 define la ejecución de una operación **send**.

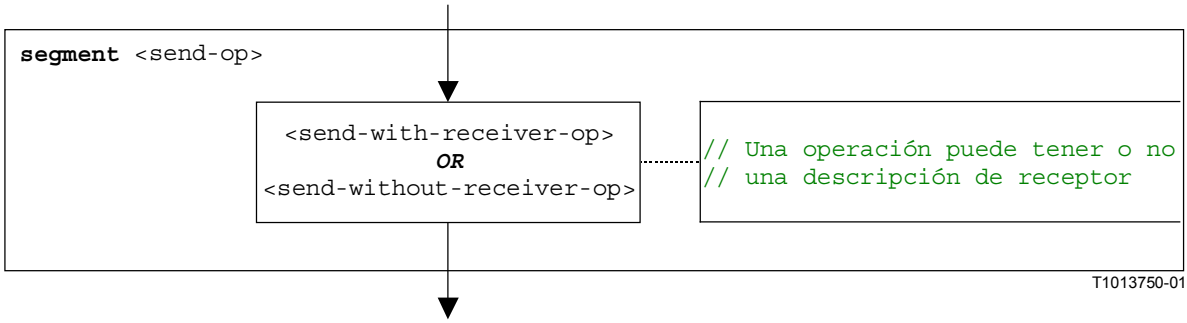


Figura B.106/Z.140 – Segmento de flujograma <send-op>

B.3.7.44.1 Segmento de flujograma <send-with-receiver-op>

El segmento de flujograma `<send-with-receiver-op>` de la figura B.107 define la ejecución de una operación **`send`** cuando el receptor se especifica en forma de una expresión.

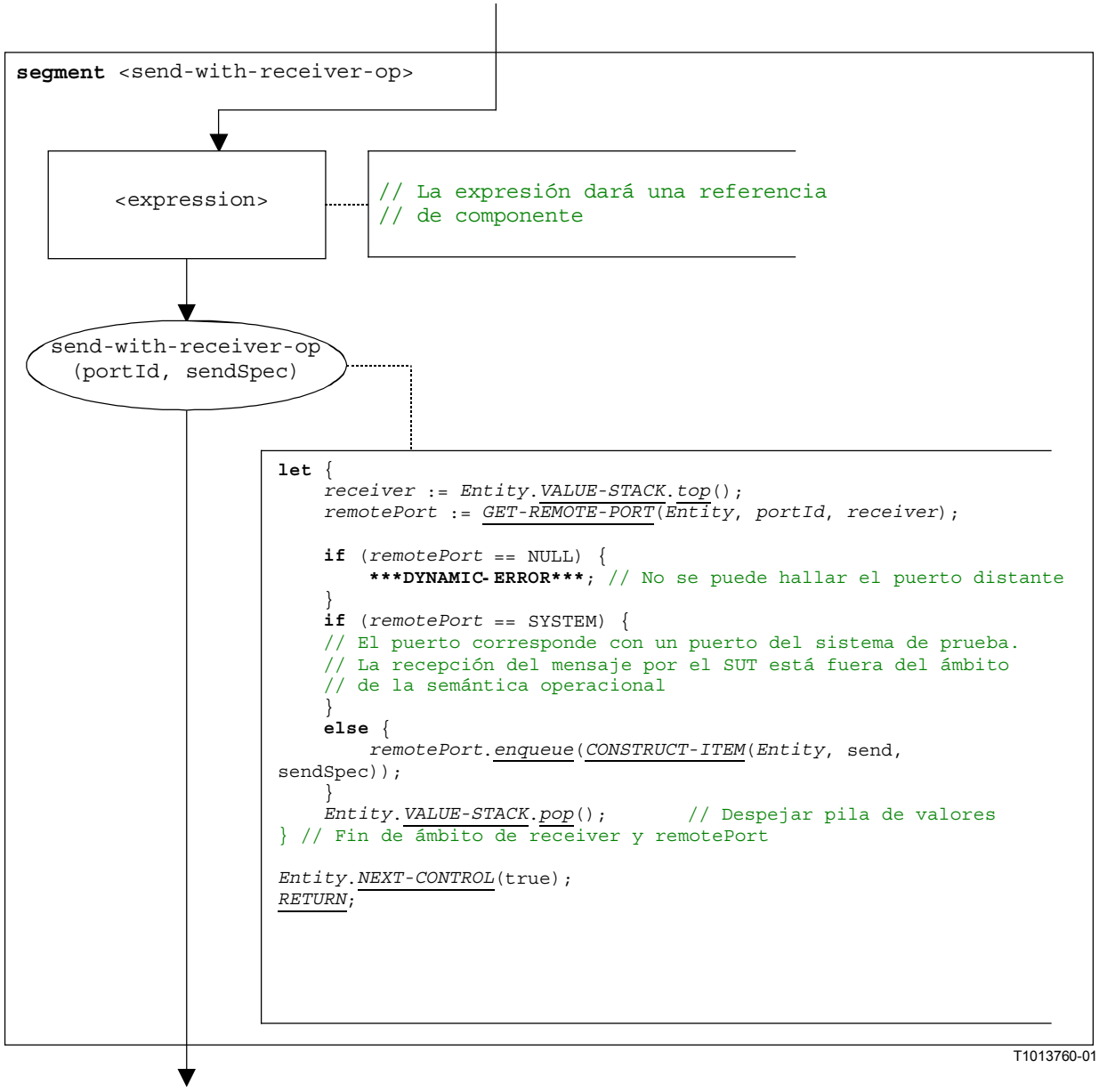


Figura B.107/Z.140 – Segmento de flujograma <send-with-receiver-op>

B.3.7.44.2 Segmento de flujograma <send-without-receiver-op>

El segmento de flujograma `<send-without-receiver-op>` de la figura B.108 define la ejecución de una operación **`send`** sin la cláusula **`to`**.

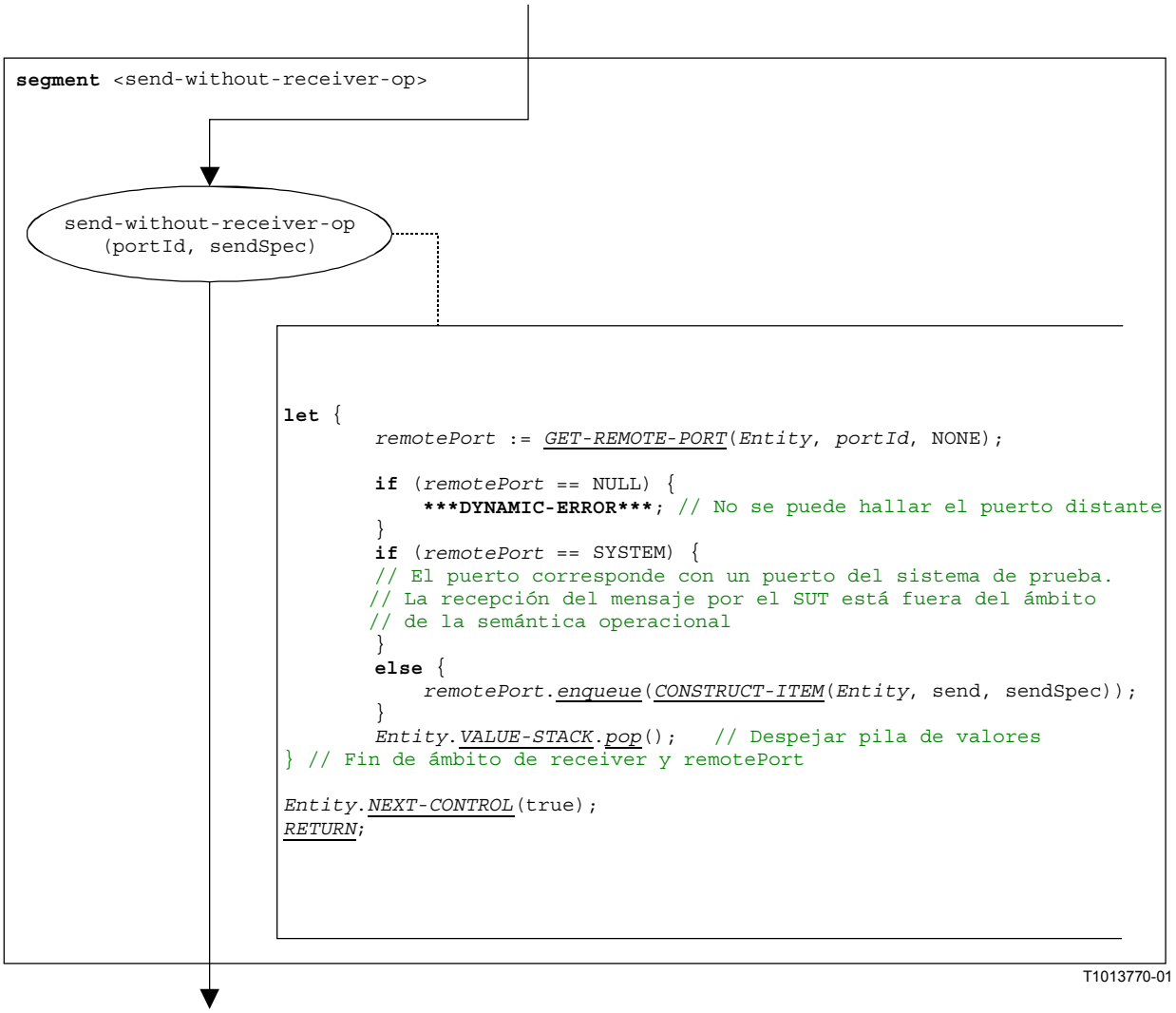


Figura B.108/Z.140 – Segmento de flujograma <send-without-receiver-op>

B.3.7.45 Operación Self

La estructura sintáctica de la operación **`self`** es:

`self`

El segmento de flujograma `<self-op>` de la figura B.109 define la ejecución de la operación **`self`**.

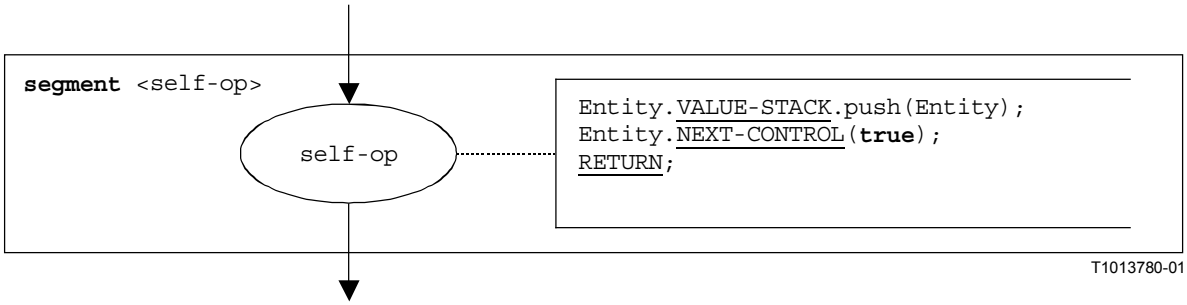


Figura B.109/Z.140 – Segmento de flujograma <self-op>

B.3.7.46 Operación Start component

La estructura sintáctica de la operación **start** component es:

```
<component_expression>.start(<function-name>(<act-par-desc1>, ... ,
                                                    <act-par-descn>))
```

La operación **start** component comienza un componente nuevamente creado. La utilización de una referencia de componente identifica el componente que se ha de comenzar. La referencia puede ser almacenada en una variable o devuelta por una función. Para simplificar, se considera que ésta es una expresión que da una referencia de componente.

El <function-name> indica el nombre de la función que define el comportamiento del nuevo componente y <act-par-descr$_1$>, ..., <act-par-descr$_n$> proporciona la descripción de los valores de parámetros reales de <function-name>. En el caso de un parámetro de valor, la descripción de un parámetro real se puede proporcionar en forma de una expresión que tiene que ser evaluada antes de poder ejecutar la llamada. El tratamiento de parámetros formales y reales es similar a su tratamiento en llamadas de función (véase B.3.7.22).

El segmento de flujograma <start-component-op> de la figura B.110 define la ejecución de la operación **start** component. Esta operación se ejecuta en cuatro pasos. En el primer paso se crea un registro de llamada. En el segundo paso se calculan los valores de parámetros reales. En el tercero paso se extrae la referencia del componente que ha de ser comenzado y, en el cuarto paso, se da al nuevo componente el control y registro de llamada.

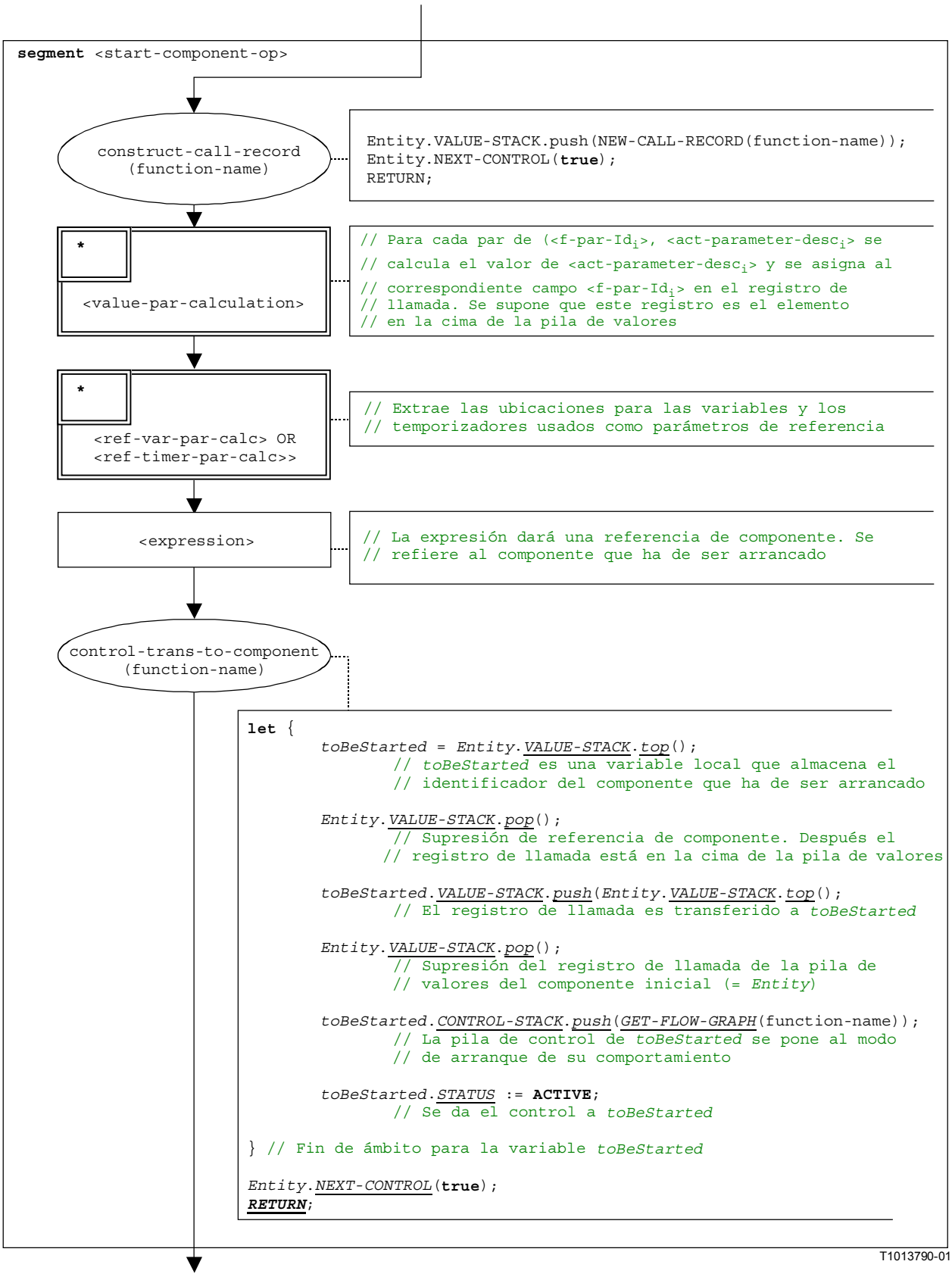


Figura B.110/Z.140 – Segmento de flujograma <start-component-op>

B.3.7.47 Operación Start port

La estructura sintáctica de la operación **start** port es:

```
<portId>.start
```

El segmento de flujograma <start-port-op> de la figura B.111 define la ejecución de la operación **start** port.

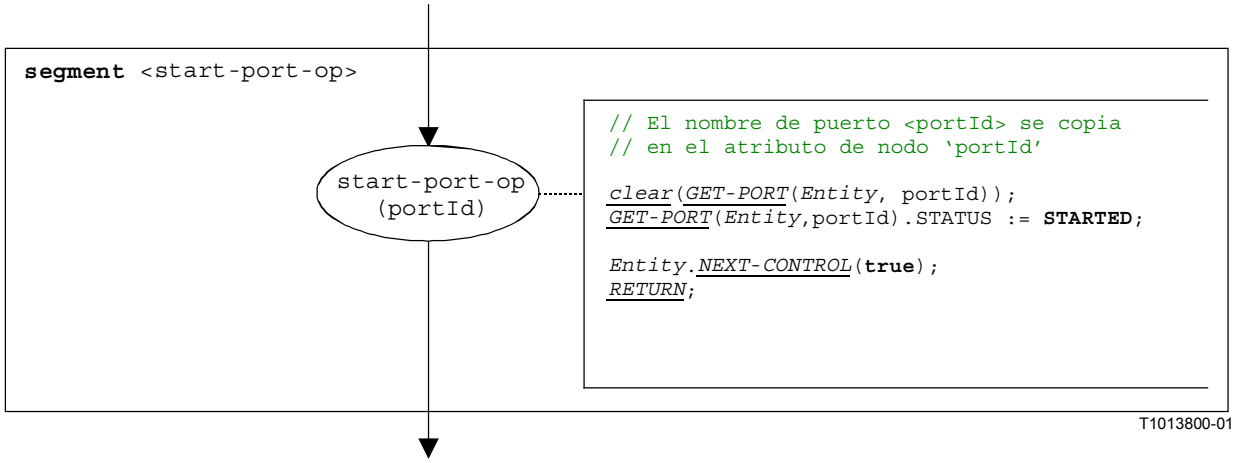


Figura B.111/Z.140 – Segmento de flujograma <start-port-op>

B.3.7.48 Operación Start timer

La estructura sintáctica de la operación **start** timer es:

```
<timerId>.start [(<float_expression>)]
```

El parámetro facultativo <float_expression> de la operación **start** timer indica la duración facultativa con la cual el temporizador será arrancado. Es una expresión que dará un valor de tipo **float**. Si se proporciona, la expresión será evaluada antes de aplicar la operación **start**. El resultado de la evaluación se introduce en *VALUE-STACK* de *Entity*.

El segmento de flujograma <start-timer-op> de la figura B.112 define la ejecución de la operación **start timer**.

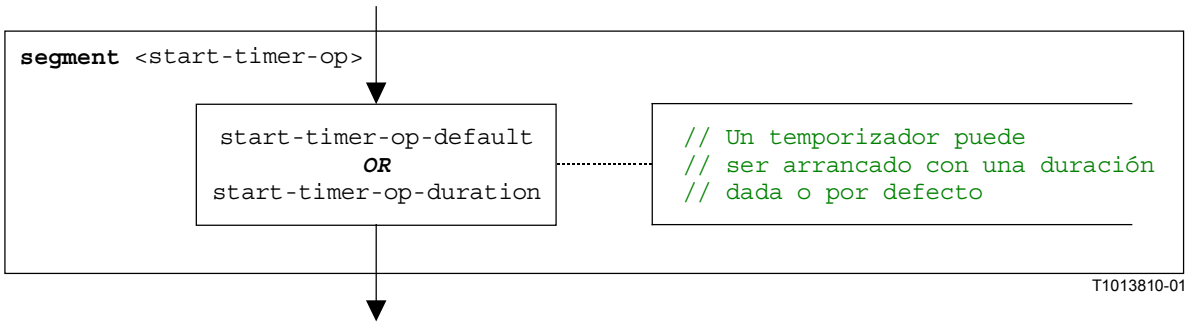


Figura B.112/Z.140 – Segmento de flujograma <start-timer-op>

B.3.7.48.1 Segmento de flujograma <start-timer-op-default>

El segmento de flujograma `<start-timer-op-default>` de la figura B.113 define la ejecución de la operación **`start timer`** con el valor por defecto.

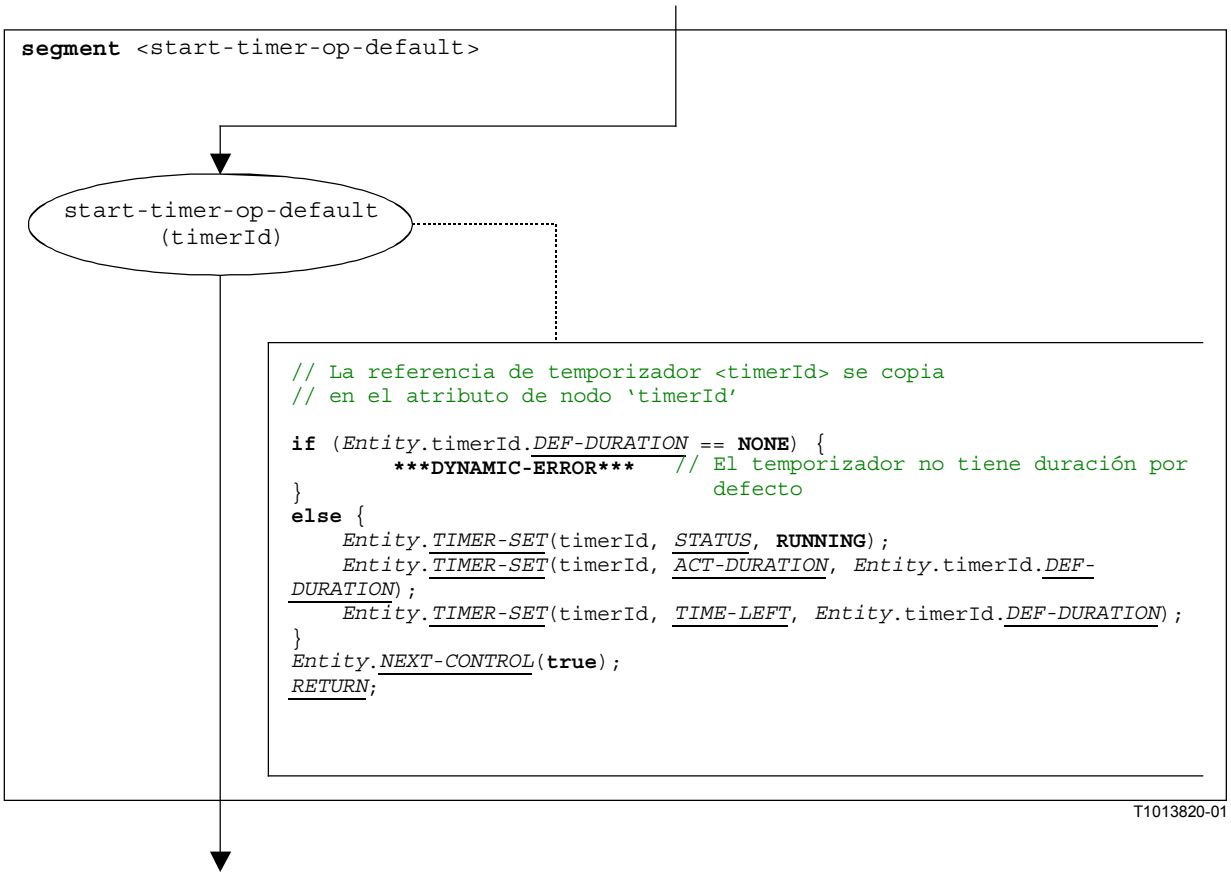


Figura B.113/Z.140 – Segmento de flujograma <start-timer-op-default>

B.3.7.48.2 Segmento de flujograma <start-timer-op-duration>

El segmento de flujograma `<start-timer-op-duration>` de la figura B.114 define la ejecución de la operación **`start`** timer con una duración dada.

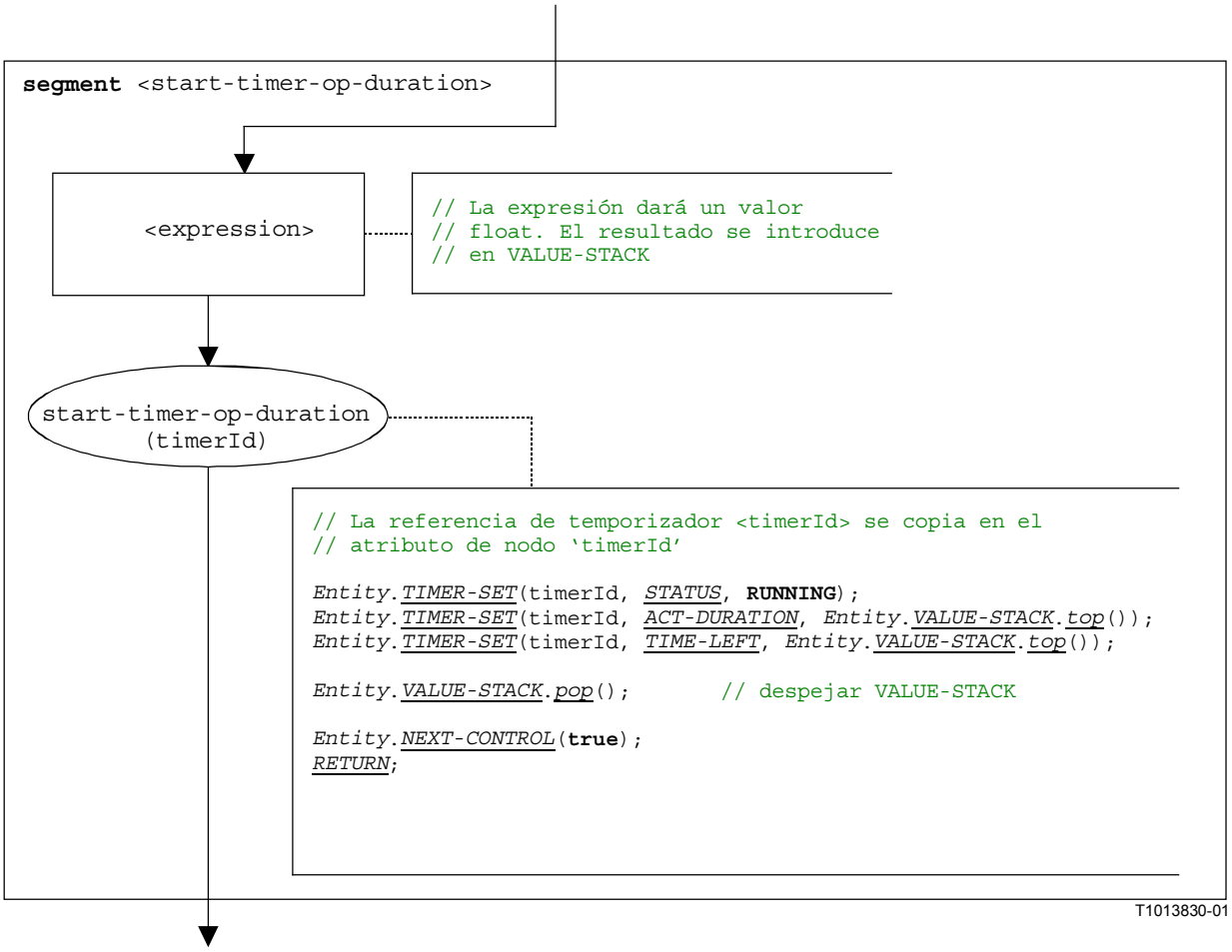


Figura B.114/Z.140 – Segmento de flujograma <start-timer-op-duration>

B.3.7.49 Bloque de enunciados

La estructura sintáctica de un bloque de enunciados es:

{ `<statement`$_1$`>; ... ; <statement`$_n$`>` **}**

Un bloque de enunciados es una unidad de ámbito. Al entrar en una unidad de ámbito, hay que inicializar nuevos ámbitos para variables, temporizadores y la pila de valores. Al salir de una unidad de ámbito, todas las variables, temporizadores y pila de valores de este ámbito tienen que ser destruidos.

El segmento de flujograma `<statement-block>` de la figura B.115 define la ejecución de un bloque de enunciados.

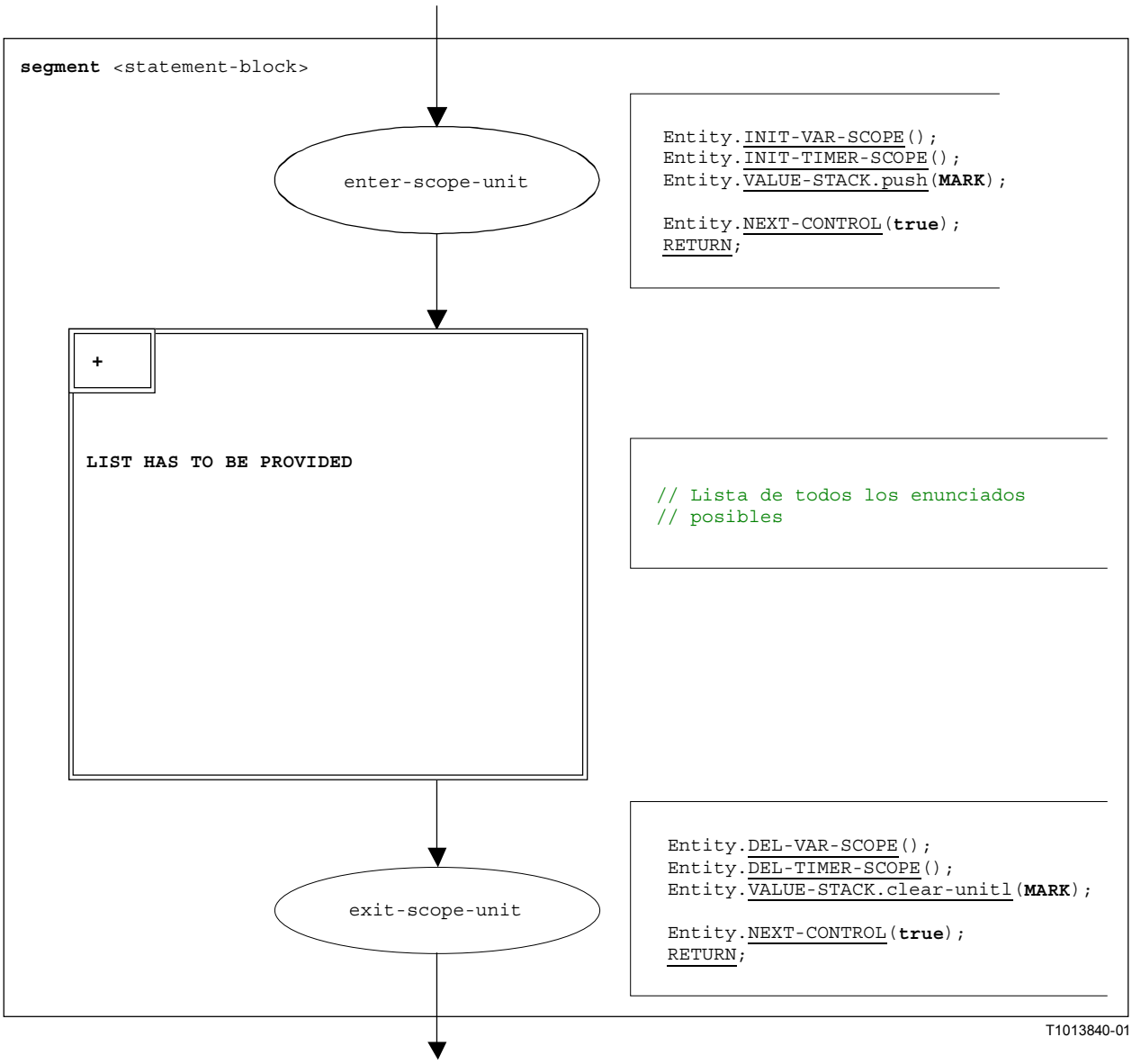


Figura B.115/Z.140 – Segmento de flujograma <statement-block>

B.3.7.50 Operación Stop entity

La estructura sintáctica de la operación **stop** entity es:

stop

El efecto de la operación **stop** depende de la entidad que ejecuta la operación **stop**.

a) Si **stop** es ejecutada por el control de módulo, la campaña de pruebas termina, es decir, todos los componentes de prueba y el control de módulo desaparecen del estado de módulo.

b) Si la operación **stop** es ejecutada por el **mtc**, todos los componentes de prueba paralelos y el **mtc** detienen la ejecución. El veredicto de caso de prueba global es actualizado e introducido en la pila de valores del control de módulo. Finalmente, el control es devuelto al control de módulo y el **mtc** termina.

c) Si la operación **stop** es ejecutada por un componente de prueba, el veredicto de caso de prueba global *TC-VERDICT* y la variable *DONE* son actualizados. Después, el componente desaparece completamente del módulo.

El segmento de flujograma <stop-entity-op> de la figura B.116 define la ejecución de la operación **stop** entity.

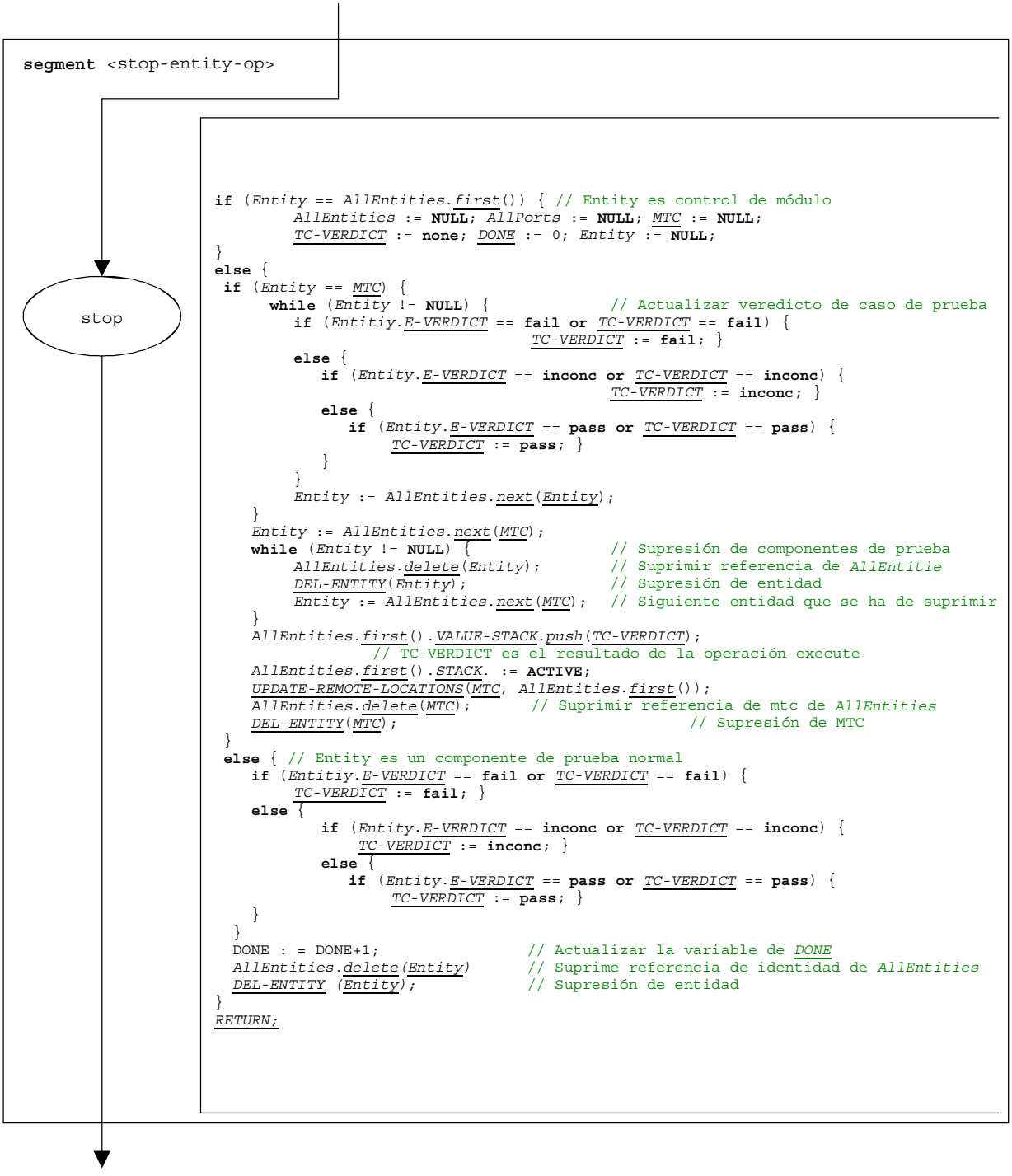


Figura B.116/Z.140 – Segmento de flujograma <stop-entity-op>

B.3.7.51 Operación Stop port

La estructura sintáctica de la operación **stop** port es:

<portId>**.stop**

El segmento de flujograma <stop-port-op> de la figura B.117 define la ejecución de una operación **stop** port.

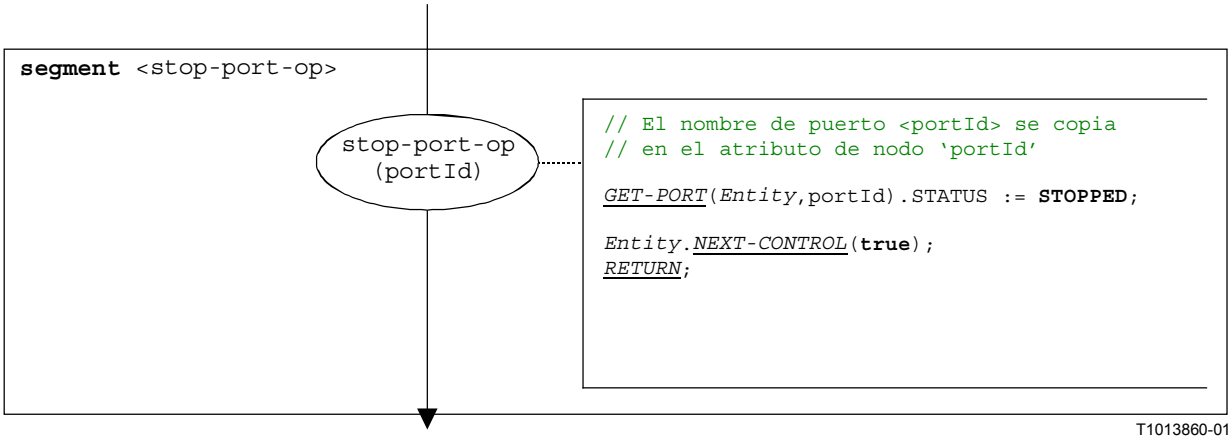


Figura B.117/Z.140 – Segmento de flujograma <stop-port-op>

B.3.7.52 Operación Stop timer

La estructura sintáctica de la operación **stop** timer es:

<timerId>**.stop**

El segmento de flujograma <stop-timer-op> de la figura B.118 define la ejecución de una operación **stop timer**.

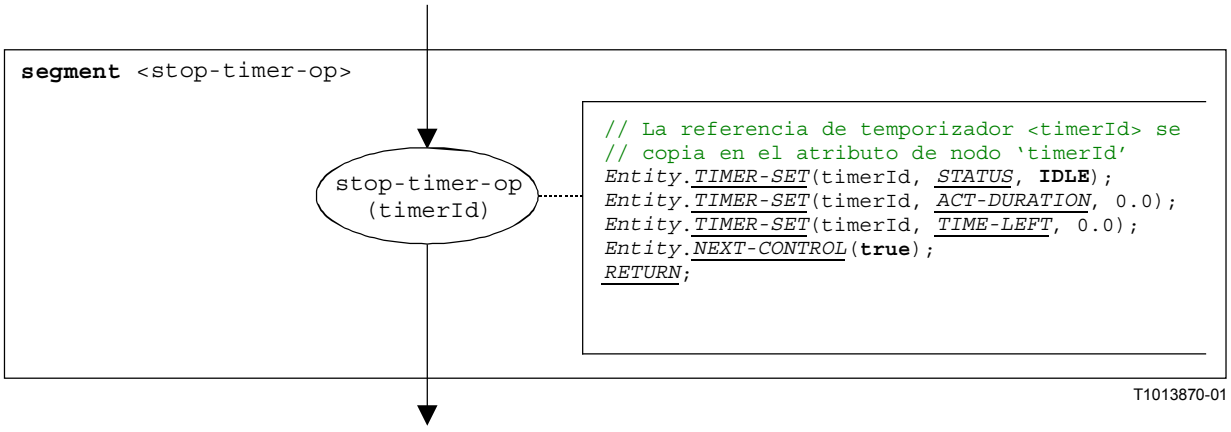


Figura B.118/Z.140 – Segmento de flujograma <stop-timer-op>

B.3.7.53 Operación Sut.action

La estructura sintáctica de la operación **sut.action** es:

sut.action (<informal description>)

El segmento de flujograma <sut.action-op> de la figura B.119 define la ejecución de la operación **sut.action**.

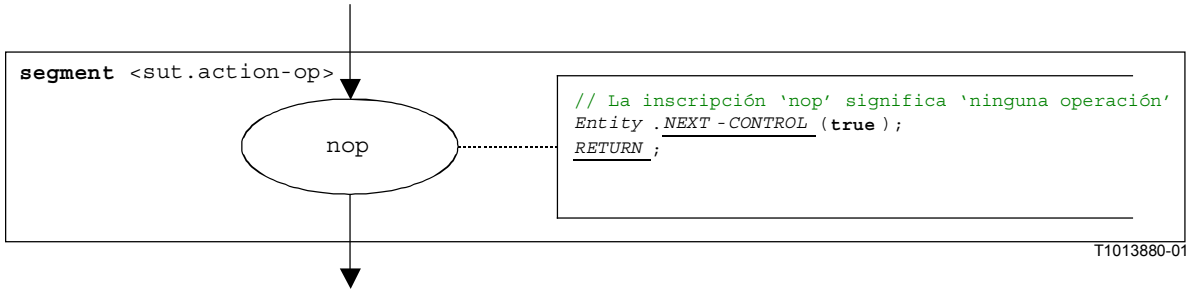


Figura B.119/Z.140 – Segmento de flujograma <sut.action-op>

NOTA – El parámetro <informal description> de la operación **sut.action** no tiene significado para la semántica operacional, por lo que no se representa en el segmento de flujograma.

B.3.7.54 Operación System

La estructura sintáctica de la operación **system** es:

system

El segmento de flujograma <system-op> de la figura B.120 define la ejecución de la operación **system**.

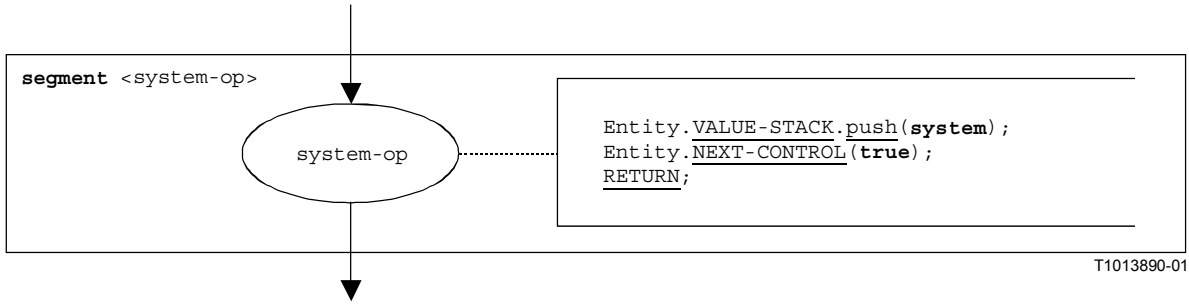


Figura B.120/Z.140 – Segmento de flujograma <system-op>

B.3.7.55 Operación Timeout timer

La estructura sintáctica de la operación **timeout** timer es:

<timerId>.**timeout**

El segmento de flujograma <timeout-timer-op> de la figura B.121 define la ejecución de la operación **timeout** timer.

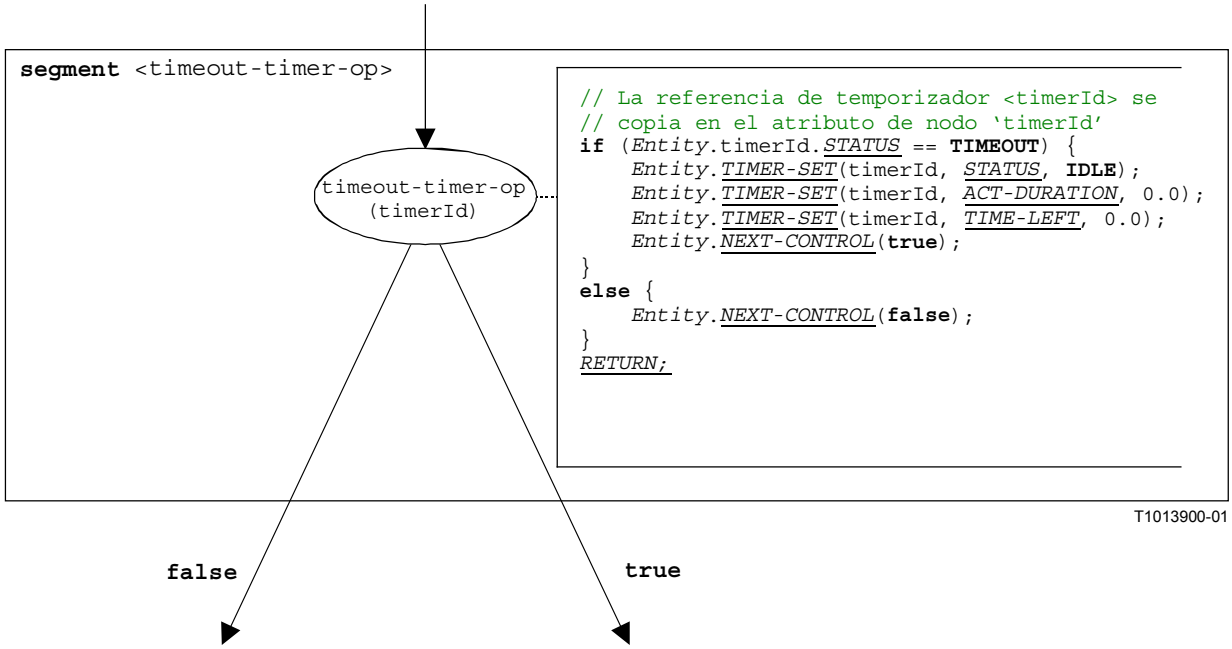


Figura B.121/Z.140 – Segmento de flujograma <running-timer-op>

NOTA – Una operación **timeout** está insertada en un enunciado **alt**. Si **timeout** da **true** o **false**, la ejecución continúa con el enunciado que sigue a la operación **timeout** (rama **true**), o hay que comprobar la siguiente alternativa en el enunciado **alt** (rama **false**).

B.3.7.56 Operación Unmap

La estructura sintáctica de la operación **unmap** es:

unmap(<component_expression>.<portId1>,**system**.<portId2>)

Se considera que los identificadores <portId1> y <portId2> son identificadores de puertos del componente de prueba y de la interfaz de sistema de prueba correspondientes. El componente al cual pertenece <portId1> es referenciado por medio de la referencia de componente <component_expression>. La referencia puede ser almacenada en variables o es devuelta por una función. Para simplificar, se considera que es una expresión que da una referencia de componente. De este modo, la pila de valores se utiliza para almacenar la referencia de componente.

NOTA – La operación **unmap** no tiene en cuenta si el enunciado **system**.<portId> aparece como primero o como segundo parámetro. Para simplificar, se supone que es siempre el segundo parámetro.

La ejecución de la operación **unmap** es definida por el segmento de flujograma <unmap-op> mostrado en la figura B.122.

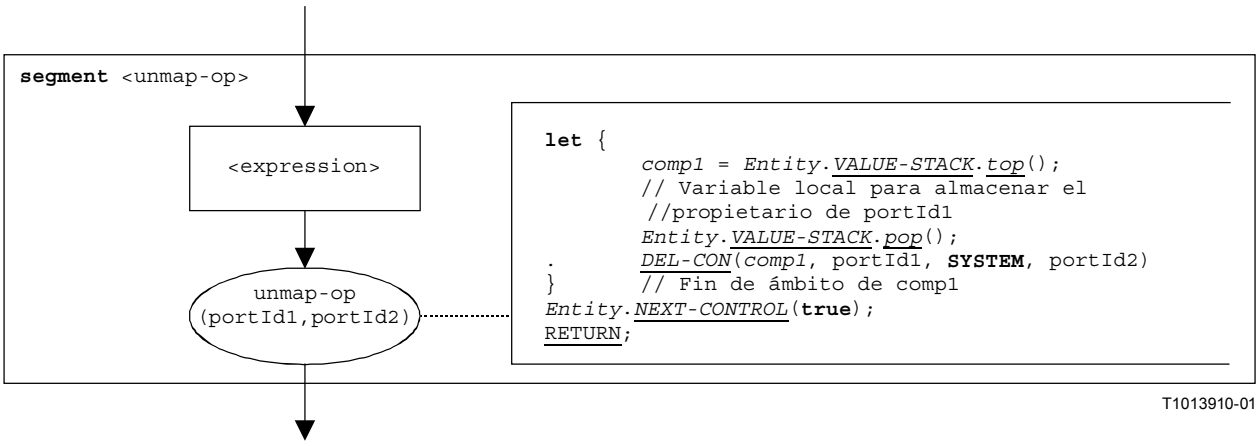


Figura B.122/Z.140 – Segmento de flujograma <unmap-op>

B.3.7.57 Operación Verdict.get

La estructura sintáctica de la operación **verdict.get** es:

verdict.get

El segmento de flujograma <verdict.get-op> de la figura B.123 define la ejecución de la operación **verdict.get**.

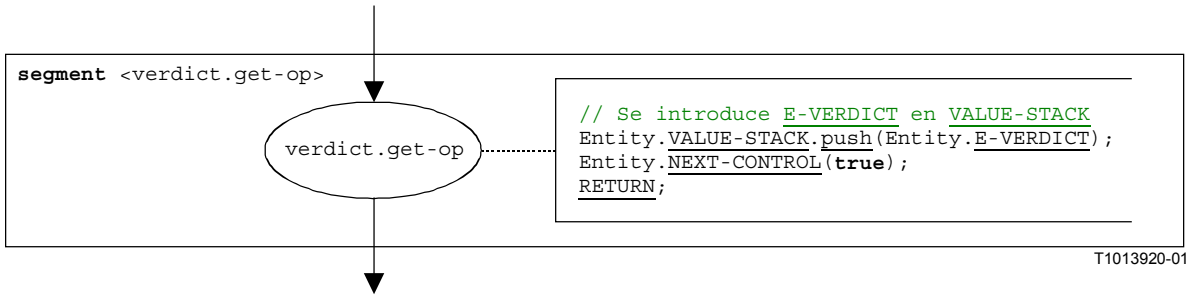


Figura B.123/Z.140 – Segmento de flujograma <verdict.get-op>

B.3.7.58 Operación Verdict.set

La estructura sintáctica de la operación **verdict.set** es:

verdict.set(<verdicttype_expression>)

NOTA – El parámetro <verdicttype_expression> de la operación **verdict.set** es una expresión que dará un valor de tipo **verdicttype**, es decir, **none**, **pass**, **inconc** o **fail**. La expresión es evaluada antes de aplicar la operación **verdict.set**.

El segmento de flujograma <verdict.set-op> de la figura B.124 define la ejecución de la operación **verdict.set**.

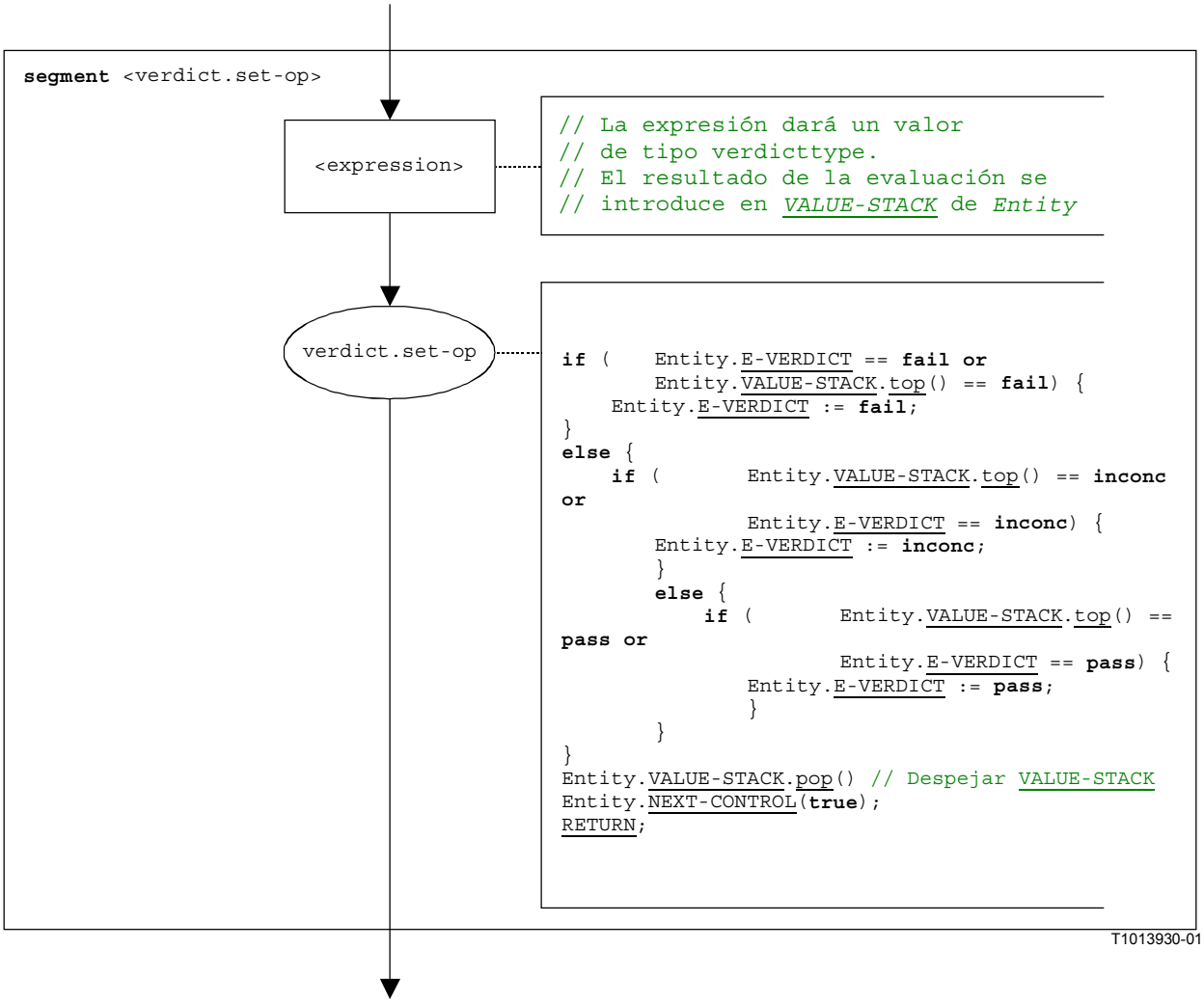


Figura B.124/Z.140 – Segmento de flujograma <verdict.set-op>

B.3.7.59 Enunciado While

La estructura sintáctica del enunciado **while** es:

while (<boolean-expression>) <statement-block>

La ejecución de un enunciado **while** se define mediante el segmento de flujograma <while-stmt> mostrado en la figura B.125.

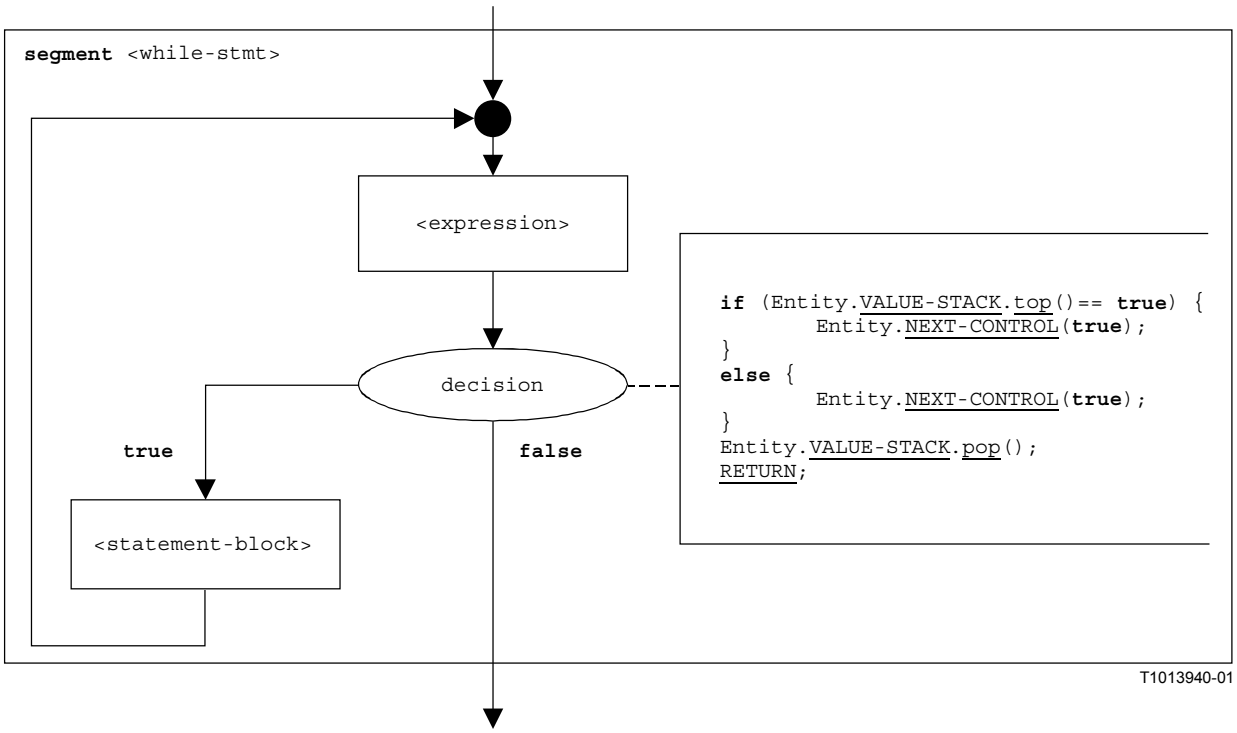


Figura B.125/Z.140 – Segmento de flujograma <while-stmt>

B.3.8 Listas de componentes semánticos operacionales

B.3.8.1 Funciones y estados

Nombre	Descripción	Referencia
NEXT	Extrae el nodo sucesor de un nodo dado en un flujograma	B.3.1.6
GET-FLOW-GRAPH	Extrae el nodo de comienzo de un flujograma	B.3.2.6
MTC	Referencia a **mtc** en el estado de módulo	B.3.3.1.1
TC-VERDICT	Veredicto de caso de prueba real en el estado de módulo	B.3.3.1.1
DONE	Número de componentes de prueba terminados (parte de estado de módulo)	B.3.3.1.1
append	Operación de lista 'append': anexa un ítem como último elemento de una lista	B.3.3.1.1
delete	Operación de lista 'delete': suprime un ítem de una lista	B.3.3.1.1
first	Operación de lista 'first': devuelve el primer elemento de una lista	B.3.3.1.1
	Operación de cola 'first': devuelve el primer elemento de una cola	B.3.3.3.2
length	Operación de lista 'length': devuelve la longitud de una lista	B.3.3.1.1
STATUS	Estado (**ACTIVE** o **BLOCKED**) de control de módulo o un componente de prueba	B.3.3.2.1
	Estado (**IDLE**, **RUNNING** o **TIMEOUT**) de un temporizador	B.3.3.2.4
	Estado (**STARTED** o **STOPPED**) de un puerto	B.3.3.3.2
E-VERDICT	Veredicto de prueba local de un componente de prueba	B.3.3.2.1

Nombre	Descripción	Referencia
CONTROL-STACK	Pila de nodos de flujograma que indican el estado de control real de una entidad	B.3.3.2.1
VALUE-STACK	Pila de valores para el almacenamiento de resultados de expresiones, operandos, operaciones y funciones	B.3.3.2.1
push	Operación de pila 'push': introduce un ítem en una pila	B.3.3.2.1
pop	Operación de pila 'pop': saca un ítem de una pila	B.3.3.2.1
top	Operación de pila 'top': devuelve el ítem en la cima de una pila	B.3.3.2.1
clear	Operación de pila 'clear': despeja una pila	B.3.3.2.1
	Operación de cola 'clear': suprime todos los elementos de una cola	B.3.3.3.2
clear-until	Operación de pila 'clear-until': saca los ítems hasta que un ítem específico es el elemento en la cima de la pila	B.3.3.2.1
NEW-ENTITY	Crea una nueva entidad de estado	B.3.3.2.1
VAR-SET	Fija el valor de una variable	B.3.3.2.4
TIMER-SET	Fija valores de un temporizador	B.3.3.2.4
DEF-DURATION	Duración por defecto de un temporizador	B.3.3.2.4
ACT-DURATION	Duración con la cual ha sido arrancado un temporizador activo	B.3.3.2.4
TIME-LEFT	Tiempo que un temporizador ha funcionado antes de expirar	B.3.3.2.4
INIT-VAR	Crea una nueva vinculación de variable	B.3.3.2.4
INIT-TIMER	Crea una nueva vinculación de temporizador	B.3.3.2.4
GET-VAR-LOC	Extrae la ubicación de una variable	B.3.3.2.4
GET-TIMER-LOC	Extrae la ubicación de un temporizador	B.3.3.2.4
INIT-VAR-LOC	Crea una nueva vinculación de variable con una ubicación existente	B.3.3.2.4
INIT-TIMER-LOC	Crea una nueva vinculación de temporizador con una ubicación existente	B.3.3.2.4
INIT-VAR-SCOPE	Inicializa un nuevo ámbito de variable	B.3.3.2.4
INIT-TIMER-SCOPE	Inicializa un nuevo ámbito de temporizador	B.3.3.2.4
DEL-VAR-SCOPE	Suprime un ámbito de variable	B.3.3.2.4
DEL-TIMER-SCOPE	Suprime un ámbito de temporizador	B.3.3.2.4
NEW-PORT	Crea un nuevo puerto	B.3.3.3.2
GET-PORT	Extrae una referencia de puerto	B.3.3.3.2
GET-REMOTE-PORT	Extrae la referencia de un puerto distante	B.3.3.3.2
ADD-CON	Añade una conexión a un estado de puerto	B.3.3.3.2
DEL-CON	Suprime una conexión de un estado de puerto	B.3.3.3.2
enqueue	Operación de cola 'enqueue': pone un ítem como último elemento en una cola	B.3.3.3.2
dequeue	Operación de cola de 'dequeue': Suprime el primer elemento de una cola	B.3.3.3.2
DEL-ENTITY	Suprime una entidad de un estado de módulo	B.3.3.4
EXISTING	Comprueba si un componente de prueba existe o no	B.3.3.4

Nombre	Descripción	Referencia
UPDATE-REMOTE-REFERENCES	Actualiza temporizadores y variables con la misma ubicación en diferentes entidades al mismo valor	B.3.3.4
CONSTRUCT-ITEM	Construye un ítem que se ha de enviar	B.3.4.3
MATCH-ITEM	Comprueba si un mensaje, llamada, respuesta o excepción que se ha recibido concuerda con una operación receptora	B.3.4.4
RETRIEVE-INFO	Extrae información de un mensaje, llamada, respuesta o excepción que se ha recibido	B.3.4.4
NEW-CALL-RECORD	Crea un registro para una llamada de función	B.3.5.1
INIT-FLOW-GRAPHS	Inicializa el tratamiento de flujogramas	B.3.6.1
GET-UNIQUE-ID	Devuelve un nuevo identificador único cuando es llamado	B.3.6.1
CONTINUE-COMPONENT	El componente real continúa su ejecución	B.3.6.1
RETURN	Devuelve el control al procedimiento de evaluación de módulo definido en la subcláusula B.3.6	B.3.6.1
DYNAMIC-ERROR	Describe la ocurrencia de un error dinámico	B.3.6.1

B.3.8.2 Palabras clave especiales

Palabra clave	Descripción	Referencia
MARK	Se usa como marca para *VALUE-STACK*	B.3.3.2
ACTIVE	*STATUS* de un estado de entidad	B.3.3.2
BLOCKED	*STATUS* de un estado de entidad	B.3.3.2
NULL	Valor simbólico para puntero y tipos semejantes a puntero para indicar que nada es direccionado	
IDLE	*STATUS* de un estado de temporizador	B.3.3.2.4
RUNNING	*STATUS* de un estado de temporizador	B.3.3.2.4
TIMEOUT	*STATUS* de un estado de temporizador	B.3.3.2.4
STARTED	*STATUS* de un puerto	B.3.3.2.4
STOPPED	*STATUS* de un puerto	B.3.3.2.4
NONE	Se usa para describir un valor indefinido	

B.3.8.3 Segmentos de flujograma

Identificador	Construcción TTCN-3 conexa	Referencia	
		Figura	Cláusula
`<alt-stmt>`	Enunciado `alt`	Figura B.25	B.3.7.1
`<alt-with-else>`	Enunciado `alt`	Figura B 26	B.3.7.1
`<alt-without-else>`	Enunciado `alt`	Figura B.27	B.3.7.1
`<assignment-stmt >`	Enunciado assignment	Figura B.29	B.3.7.2
`<b-call-with-receiver>`	`call`	Figura B.35	B.3.7.3.3
`<b-call-without-receiver>`	`call`	Figura B.36	B.3.7.3.4
`<b-call-with-rec-dur>`	`call`	Figura B.37	B.3.7.3.5

Identificador	Construcción TTCN-3 conexa	Referencia	
		Figura	Cláusula
<b-call-without-rec-dur>	call	Figura B.38	B.3.7.3.6
<blocking-call-op>	call	Figura B.31	B.3.7.3
<call-op>	call	Figura B.30	B.3.7.3
<catch-op>	catch	Figura B.39	B.3.7.4
<catch-with-sender>	Se usa en la operación catch	Figura B.40	B.3.7.4.1
<catch-without-sender>	Se usa en la operación catch	Figura B.41	B.3.7.4.2
<clear-port-op>	clear port	Figura B.42	B.3.7.5
<constant-declaration>	Declaración de una constante	Figura B.44	B.3.7.7
<connect-op>	connect	Figura B.43	B.3.7.6
<create-op>	create	Figura B.45	B.3.7.8
<disconnect-op>	disconnect	Figura B.53	B.3.7.12
<do-while-stmt>	Enunciado do-while	Figura B.54	B.3.7.13
<done-all-comp-op>	all component.done	Figura B.55	B.3.7.14
<done-any-comp-op>	any component.done	Figura B.56	B.3.7.15
<done-component-op>	done component	Figura B.57	B.3.7.16
<execute-stmt>	execute	Figura B.58	B.3.7.17
<execute-timeout>>	execute	Figura B.59	B.3.7.17
<execute-without-timeout>>	execute	Figura B.60	B.3.7.17
<expression>	Expression	Figura B.61	B.3.7.18
<finalize-component-init>	Se usa en el comportamiento de definiciones de tipo de componente	Figura B.66	B.3.7.19
<for-stmt>>	Enunciado for	Figura B.68	B.3.7.21
<function-call>	Llamada de funciones definidas por el usuario	Figura B.69	B.3.7.22
<func-op-call>	Se usa en <expression>	Figura B.64	B.3.7.18.3
<getcall-op>	**getcall**	Figura B.74	B.3.7.27
<getcall-with-sender>	Se usa en la operación **getcall**	Figura B.75	B.3.7.27.1
<getcall-without-sender>	Se usa en la operación **getcall**	Figura B.76	B.3.7.27.2
<getreply-op>	**getreply**	Figura B.76	B.3.7.28
<getreply-with-sender>	Se usa en la operación **getreply**	Figura B.78	B.3.7.28.1
<getreply-without-sender>	Se usa en la operación **getreply**	Figura B.79	B.3.7.28.2
<goto-stmt>	**goto**	Figura B.80	B.3.7.29
<if-else-stmt>	**if-else**	Figura B.80	B.3.7.30
<if-with-else-branch>	**if-else**	Figura B.82	B.3.7.30.1
<if-without-else-branch>	**if-else**	Figura B.83	B.3.7.30.2
<init-component-scope>	Se usa en el comportamiento de definiciones de tipo de componente	Figura B.67	B.3.7.20
<label-stmt>	**label**	Figura B.84	B.3.7.31
<lit-value>	Se usa en <expression>	Figura B.62	B.3.7.18.1
<log-stmt>	**log**	Figura B.85	B.3.7.32

Identificador	Construcción TTCN-3 conexa	Referencia	
		Figura	Cláusula
<map-op>	Operación **map**	Figura B.86	B.3.7.33
<mtc-op>	**mtc**	Figura B.87	B.3.7.34
<nb-call-with-receiver>	**call**	Figura B.33	B.3.7.3.1
<nb-call-without-receiver>	**call**	Figura B.34	B.3.7.3.2
<non-blocking-call-op>	**call**	Figura B.32	B.3.7.3
<operator-appl>	Se usa en <expression>	Figura B.65	B.3.7.18.4
<parameter-handling>	Creación de entidades, llamadas de funciones	Figura B.73	B.3.7.26
<port-declaration>	Declaración de un puerto	Figura B.46	B.3.7.9
<raise-op>	**raise**	Figura B.88	B.3.7.35
<raise-with-receiver-op>	**raise**	Figura B.89	B.3.7.35.1
<raise-without-receiver-op>	**raise**	Figura B.90	B.3.7.35.2
<read-timer-op>	**read** timer	Figura B.91	B.3.7.36
<receive-assignment>	Se usa en la operación **receive**	Figura B.95	B.3.7.37.3
<receive-op>	**receive**	Figura B.92	B.3.7.37
<receive-with-sender>	Se usa en la operación **receive**	Figura B.93	B.3.7.37.1
<receive-without-sender>	Se usa en la operación **receive**	Figura B.94	B.3.7.37.2
<receiving-branch>	Enunciado **alt**	Figura B.28	B.3.7.1.1
<reply-op>	**reply**	Figura B.96	B.3.7.38
<reply-with-receiver-op>	**reply**	Figura B.97	B.3.7.38.1
<reply-without-receiver-op>	**reply**	Figura B.98	B.3.7.38.2
<ref-par-var-calc>	Creación de entidades, llamadas de funciones	Figura B.71	B.3.7.24
<ref-par-timer-calc>	Creación de entidades, llamadas de funciones	Figura B.72	B.3.7.25
<return-stmt>	**return**	Figura B.99	B.3.7.39
<return-with-value>	**return**	Figura B.100	B.3.7.39.1
<return-without-value>	**return**	Figura B.101	B.3.7.39.2
<running-all comp-op>	**all component.running**	Figura B.102	B.3.7.40
<running-any comp-op>	**any component.running**	Figura B.103	B.3.7.41
<running-component-op>	**running** component	Figura B.104	B.3.7.42
<running-timer-op>	**running** timer	Figura B.105	B.3.7.43
<self-op>	**self**	Figura B.109	B.3.7.45
<send-op>	**send**	Figura B.106	B.3.7.44
<send-with-receiver-op>	**send**	Figura B.107	B.3.7.44.1
<send-without-receiver-op>	**send**	Figura B.108	B.3.7.44.2
<start-component-op>	**start** component	Figura B.110	B.3.7.46
<start-port-op>	**start** port	Figura B.111	B.3.7.47
<start-timer-op>	**start** timer	Figura B.112	B.3.7.48
<start-timer-op-default>	**start** timer	Figura B.113	B.3.7.48.1

Identificador	Construcción TTCN-3 conexa	Referencia	
		Figura	Cláusula
<start-timer-op-duration>	**start** timer	Figura B.114	B.3.7.48.2
<stop-entity-op>	**stop** execution de control de módulo, mtc o un componente de prueba	Figura B.116	B.3.7.50
<stop-port-op>	**stop** port	Figura B.117	B.3.7.51
<statement-block>	Bloque de enunciados	Figura B.115	B.3.7.49
<stop-timer-op>	**stop** timer	Figura B.118	B.3.7.52
<sut.action-op>	**sut.action-op**	Figura B.119	B.3.7.53
<system-op>	**system**	Figura B.120	B.3.7.54
<timeout-timer-op>	**timeout** timer	Figura B.121	B.3.7.55
<timer-declaration>	Declaración de un temporizador	Figura B.47	B.3.7.10
<timer-decl-default>	Declaración de un temporizador con una duración por defecto	Figura B.48	B.3.7.10.1
<timer-decl-no-def>	Declaración de un temporizador sin una duración por defecto	Figura B.49	B.3.7.10.2
<unmap-op>	Operación **unmap**	Figura B.122	B.3.7.56
<value-par-calculation>	Creación de entidades, llamadas de funciones	Figura B.70	B.3.7.23
<variable-declaration>	Declaración de una variable	Figura B.50	B.3.7.11
<variable-declaration-init>	Declaración de una variable con valores iniciales	Figura B.51	B.3.7.11.1
<variable-declaration-undef>	Declaración de una variable sin un valor inicial	Figura B.52	B.3.7.11.2
<var-value>	Se usa en <expression>	Figura B.63	B.3.7.18.2
<verdit.get-op>	**verdict.get**	Figura B.123	B.3.57
<verdit.set-op>	**verdict.set**	Figura B.124	B.3.7.58
<while-stmt>	Enunciado **while**	Figura B.125	B.3.7.59

Anexo C

Concordancia de valores entrantes

C.1 Mecanismos de concordancia de plantillas

El presente anexo especifica los mecanismos de concordancia que pueden ser utilizados en plantillas TTCN-3 (y sólo en plantillas).

C.1.1 Concordancia de valores específicos

Los valores específicos son los mecanismos de concordancia básicos de las plantillas TTCN-3. Los valores específicos en plantillas son expresiones que no contienen ningún mecanismo de concordancia ni comodines. A menos que se especifique otra cosa, un campo de plantilla concuerda con el valor de campo entrante correspondiente solamente si el valor de campo entrante tiene exactamente el mismo valor que el valor que da la expresión en la plantilla. Por ejemplo:

```
// Dada la definición de tipo de mensaje

type record MyMessageType
{
   integer  field1,
   charstring  field2,
   boolean  field3 optional,
   integer[4]  field4
}

// Una plantilla de mensaje que usa valores específicos podría ser
template MyMessageType MyTemplate:=
{
   field1 := 3+2,           // Valor específico de tipo integer
   field2 := "My string",   // Valor específico de tipo charstring
   field3 := true,          // Valor específico de tipo boolean
   field4 := {1,2,3}        // valor específico de matriz de integer
}
```

C.1.2 Mecanismos de concordancia de valores

C.1.2.1 Lista de valores

Las listas de valores especifican listas de valores entrantes aceptables. Pueden ser utilizadas en valores de todos tipos. Un campo de plantilla que utiliza una lista de valores concuerda con el campo entrante correspondiente solamente si el valor de campo entrante concuerda con cualquiera de los valores en la lista de valores. Cada valor de la lista será del tipo declarado para el campo de plantilla en el cual este mecanismo se utiliza. Por ejemplo:

```
template Mymessage MyTemplate:=
{
   field1 := (2,4,6),                // Lista de valores integer
   field2 := ("String1", "String2"), // Lista de valores charstring
   :
   :
}
```

C.1.2.2 Lista de valores complementados

La palabra clave **complement** indica una lista de valores que no serán aceptados como valores entrantes (es decir, es el complemento de una lista de valores). Se puede utilizar en todos los valores de todos los tipos.

Cada valor en la lista será del tipo declarado para el campo de plantilla en el cual se utiliza el complemento. Un campo de plantilla que utiliza complemento concuerda con el campo entrante correspondiente solamente si el campo entrante no concuerda con ninguno de los valores enumerados en la lista de valores. Naturalmente, la lista de valores puede ser un solo valor.

Ejemplo:

```
template Mymessage MyTemplate:=
{
   complement (1,3,5),    // Lista de valores integer inaceptables
   :
   field3 not(true)       // Concordará false
   :
}
```

C.1.2.3 Omisión de valores

La palabra clave **omit** indica que un campo de plantillas facultativo estará ausente. Se puede utilizar en valores de todos los tipos a condición de que el campo de plantilla sea facultativo. Por ejemplo:

```
template Mymessage:MyTemplate:=
{  :
   :
   field3 := omit,        // Omitir este campo
   :
}
```

C.1.2.4 Cualquier valor

El símbolo de concordancia "?" (*AnyValue*) se utiliza para indicar que es aceptable cualquier valor entrante válido. Se puede utilizar en valores de todos los tipos. Un campo de plantilla que utiliza el mecanismo cualquier valor concuerda con el campo entrante correspondiente solamente si el campo entrante da un solo elemento del tipo especificado. Por ejemplo:

```
template Mymessage:MyTemplate:=
{
   field1 := ?,  // Concordará con cualquier integer
   field2 := ?,  // Concordará con cualquier valor charstring no vacío
   field3 := ?,  // Concordará con true o false
   field4 := ?   // Concordará con cualquier secuencia de enteros
}
```

C.1.2.5 Cualquier valor o ningún valor

El símbolo de concordancia "*" (*AnyValueOrNone*) se utiliza para indicar que es aceptable cualquier valor entrante válido, incluida la omisión de ese valor. Se puede utilizar en valores de todos los tipos, a condición de que el campo de plantilla sea declarado como facultativo.

Un campo de plantilla que utiliza este símbolo concuerda con el campo entrante correspondiente solamente si el campo entrante evalúa cualquier elemento del tipo especificado, o si el campo entrante está ausente. Por ejemplo:

```
template Mymessage:MyTemplate:=
{  :
   field3 := *,  // Concordará true o false o campo omitido
   :
}
```

C.1.2.6 Gama de valores

Las gamas indican un gama limitada de valores aceptables. Se utilizarán solamente en valores de tipo **integer** (y subtipos de integer). Un valor límite será:

a) infinito o -infinito;

b) una expresión que da un valor entero específico.

La frontera inferior se pondrá en el lado izquierdo de la gama y la frontera superior en el lado derecho. La frontera inferior será menor que la frontera superior. Un campo de plantilla que utiliza una gama concuerda con el correspondiente campo entrante solamente si el valor de campo entrante es igual a uno de los valores de la gama. Por ejemplo:

```
template Mymessage:MyTemplate:=
{
   field1 := (1 .. 6), // Gama de tipo integer
   :
   :
   :
}
// Otras entradas para field1 pudieran ser (-infinity a 8) o (12 a infinity)
```

C.1.3 Mecanismos de concordancia dentro de valores

C.1.3.1 Cualquier elemento

El símbolo de concordancia "?" (*AnyElement*) se utiliza para indicar que sustituye a elementos individuales de una cadena (salvo cadenas de caracteres), **record of**, **set of** o una matriz. Se utilizará solamente dentro de valores de tipos de cadena, tipos **record of**, tipos **set of** y matrices. Por ejemplo:

```
template Mymessage MyTemplate:=
{  :
   field2 := "abcxyz",
   field3 := '10???'B, // Donde cada "?" puede ser 0 ó 1
   field4 := {1, ?, 3} // Donde? puede ser cualquier valor entero
}
```

NOTA – El símbolo "?" en field4 puede ser interpretado como *AnyValue,* como un valor entero, o *AnyElement* dentro de un **record of**, **set of** o matriz. Como ambas interpretaciones resultan en la misma concordancia no se plantean problemas.

C.1.3.1.1 Utilización de comodines de un carácter

Si se requiere expresar el comodín "?" en cadenas de caracteres, esto se hará utilizando patrones de caracteres (véase C.1.5). Por ejemplo "abcdxyz", "abccxyz" "abcxxyz" etc., todos concordarán con el **pattern** "abc?xyz". Sin embargo, "abcxyz", "abcdefxyz", etc., no concordarán.

C.1.3.2 Cualquier número de elementos o ningún elemento

El símbolo de concordancia "*" (*AnyElementsOrNone*) se utiliza para indicar que sustituye a ninguno o a cualquier número de elementos consecutivos de una cadena (salvo cadenas de caracteres), un **record of**, un **set of** o una matriz. Se utilizará solamente dentro de valores de tipo de cadenas o matrices. El símbolo "*" concuerda la secuencia más larga de elementos posible, de acuerdo con el esquema especificado por los símbolos que rodean el "*". Por ejemplo:

```
template Mymessage MyTemplate:=
{  :
   field2 := "abcxyz",
   field3 := '10*11'B, // Donde "*" puede ser cualquier secuencia de bits
                       // (posiblemente vacía)
   field4 := {*, 2, 3} // Donde el primer elemento puede ser cualquier valor
                          entero u omitido
}

var charstring MyStrings[4];
MyPCO.receive(MyStrings:{"abyz", *, "abc" });
```

Si aparece "*" en el nivel más alto dentro de una cadena, **record of**, **set of** o matriz, se interpretará como *AnyElementsOrNone*.

NOTA – Esta regla impide la interpretación posible de "*" como *AnyValueOrNone* que sustituye a un elemento dentro de una cadena, **record of**, **set of** o matriz.

C.1.3.2.1 Utilización de comodines de múltiples caracteres

Si se requiere expresar el comodín "`*`" en cadenas de caracteres, esto se hará utilizando esquemas de caracteres (véase C.1.5). Por ejemplo: "`abcxyz`", "`abcdefxyz`" "`abcabcxyz`" etc., todos concordarán con el **`pattern`** "`abc*xyz`".

C.1.4 Concordancia de atributos de valores

C.1.4.1 Restricciones de longitud

El atributo restricciones de longitud se utiliza para restringir la longitud de valores de cadena y el número de elementos en una estructura **`set of`** o **`record of`**. Se utilizará solamente como un atributo de los siguientes mecanismos: *AnyValue*, *AnyValueOrNone*, *AnyElement* y *AnyElementsOrNone*. Se puede utilizar también junto con el atributo **`ifpresent`** (si está presente). La sintaxis de **`length`** se indica en 6.2.3 y 6.3.3.

Las unidades de longitud han de ser interpretadas de acuerdo con el cuadro 4 en el caso de valores de cadena. Para tipos **`set of`** y **`record of`**, la unidad de longitud es el tipo replicado. Las fronteras serán indicadas por expresiones que se resuelven en valores **`integer`** no negativos específicos. Como otra posibilidad, se puede utilizar la palabra clave **`infinity`** como un valor para la frontera superior con el fin de indicar que no hay un límite de longitud superior.

Las especificaciones de longitud para la plantilla no estarán en conflicto con las restricciones de longitud (si las hubiere) del tipo correspondiente. Un campo de plantilla que utiliza **`length`** como un atributo de un símbolo concuerda con el correspondiente campo entrante solamente si el campo entrante concuerda con el símbolo y su atributo asociado. El atributo **`length`** concuerda si la longitud del campo entrante es mayor o igual al límite inferior especificado y menor o igual al límite superior. En el caso de un solo valor de longitud, el atributo **`length`** concuerda solamente si la longitud del campo recibido es exactamente el valor especificado.

En el caso de un campo omitido, se considera que el atributo **`length`** concuerda (es decir, con **`omit`** es redundante). Con *AnyValueOrNone* e **`ifpresent`** impone una restricción al valor entrante, si lo hubiere. Por ejemplo:

```
template Mymessage MyTemplate:=
{
   field1 := complement (4,5) length (1 .. 6),   // Es igual que (1,2,3,6)
   field2 := "ab*ab" length(13)    // la longitud máxima de la cadena
                                   // AnyElementsOrNone es 9 caracteres
      :
   }
```

C.1.4.2 El indicador IfPresent

`ifpresent` indica que puede haber una concordancia si un campo facultativo está presente (es decir, no está omitido). Este atributo se puede utilizar con todos los mecanismos de concordancia, a condición de que el tipo sea declarado como facultativo.

El campo de plantilla que utiliza **`ifpresent`** concuerda con el campo entrante correspondiente solamente si el campo entrante concuerda según el mecanismo de concordancia asociado, o si el campo entrante está ausente. Por ejemplo:

```
template Mymessage:MyTemplate:=
{  :
   field2 := "abcd" ifpresent,  // Concuerda con "abcd" si no se omite
   :
   :
}
```

NOTA – *AnyValueOrNone* tiene exactamente el mismo significado que **`? ifpresent`**.

C.1.5 Concordancia de patrones de caracteres

Se pueden utilizar patrones de caracteres en plantillas para definir el formato de una cadena de caracteres requerida que se ha de recibir. Se pueden utilizar para concordar valores **charstring** y **universal charstring**. Además de caracteres literales, los patrones de caracteres permiten el uso de los metacaracteres ? y * para significar cualquier carácter y cualquier número de cualquier carácter, respectivamente. Por ejemplo:

```
template charstring MyTemplate:= pattern "ab??xyz*";
```

Esta plantilla concordaría con cualquier cadena de caracteres formada por los caracteres 'ab', seguidos por dos caracteres cualesquiera, seguidos por los caracteres 'xyz', seguidos por cualquier número de cualesquiera caracteres.

Si se requiere interpretar cualquier metacarácter, éste debe estar precedido por el metacarácter '\'. Por ejemplo:

```
template charstring MyTemplate:= pattern "ab?\?xyz*";
```

Esta plantilla concordaría con cualquier cadena de caracteres formada por los caracteres 'ab', seguidos por cualesquiera caracteres, seguidos por los caracteres '?xyz', seguidos por cualquier número de cualesquiera caracteres.

Además de valores de cadena directos, es posible también dentro del enunciado **pattern** (patrón) utilizar referencias a plantillas, constantes o variables existentes. La referencia se resolverá en uno de los tipos de cadena de caracteres y más de uno. Por ejemplo:

```
const charstring MyString:= "ab?";
template charstring MyTemplate:= pattern MyString;
```

Esta plantilla concordaría con cualquier cadena de caracteres formada por los caracteres 'ab', seguidos por cualesquiera caracteres. En efecto, se interpretará que cualquier cadena de caracteres que sigue a la palabra clave **pattern** explícitamente o mediante referencia cumple las reglas definidas en esta cláusula.

El enunciado **pattern** permite utilizar también el operador concatenado y, en el caso de cadenas de caracteres universales, el uso de la producción Quadruple para especificar un carácter individual. Por ejemplo:

```
const charstring MyString:= "ab?";
template universal charstring MyTemplate:= pattern MyString
   & "de" & (1, 1, 13, 7);
```

Esta plantilla concordaría con cualquier cadena de caracteres formada por los caracteres 'ab', seguidos por cualesquiera caracteres, seguidos por los caracteres 'de', seguidos por el carácter de ISO/CEI 10646-1 con grupo=1, plano=1, fila=65 y célula=7.

Anexo D

Funciones TTCN-3 predefinidas

D.1 Funciones TTCN-3 predefinidas

Este anexo define las funciones TTCN-3 predefinidas.

D.1.1 Entero a carácter

```
int2char(integer value) return char
```

Esta función convierte un valor **integer** en la gama de 0 .. 127 (codificación de 8 bits) en un valor de carácter de UIT-T T.50 e ISO/CEI 646 [5]. El valor entero describe la codificación de 8 bits del carácter.

La función devuelve –1 si el valor del argumento es negativo o mayor que 127.

D.1.2 Carácter a entero

```
char2int(char value) return integer
```

Esta función convierte un valor **char** de UIT-T T.50 e ISO/CEI 646 [5] en un valor entero de la gama 0 .. 127. El valor entero describe la codificación de 8 bits del carácter.

D.1.3 Entero a carácter universal

```
int2unichar(integer value) return universal char
```

Esta función convierte un valor **integer** en la gama de 0 .. 268435455 (codificación de 32 bits) en un valor de carácter de ISO/CEI 10646 [6]. El valor entero describe la codificación de 32 bits del carácter.

La función devuelve a –1 si el valor del argumento es negativo o mayor que 268435455.

D.1.4 Carácter universal a entero

```
unichar2int(universal char value) return integer
```

Esta función convierte un valor **univeral char** de ISO/CEI 10646-1 [6] en un valor entero de la gama 0 .. 268435455. El valor entero describe la codificación de 32 bits del carácter.

D.1.5 Cadena de bits a entero

```
bit2int(bitstring value) return integer
```

Esta función convierte un solo valor **bitstring** en un solo valor **integer**.

A los efectos de esta conversión, se interpretará **bitstring** como un valor **integer** con base positiva 2. El bit más a la derecha es el menos significativo, el bit más a la izquierda es el más significativo. Los bits 0 y 1 representan los valores decimales 0 y 1, respectivamente.

D.1.6 Cadena hexadecimal a entero

```
hex2int(hexstring value) return integer
```

Esta función convierte un solo valor **hexstring** en un solo valor **integer**.

A los efectos de esta conversión, se interpretará que **hexstring** es un valor **integer** base 16 positivo. El dígito hexadecimal más a la derecha es el menos significativo. El dígito hexadecimal más a la izquierda es el más significativo. Los dígitos hexadecimales 0 .. F representan los valores decimales 0 .. 15, respectivamente.

D.1.7 Cadena de octetos a entero

```
oct2int(octetstring value) return integer
```

Esta función convierte un valor **octetstring** en un valor **integer**.

A los efectos de esta conversión, se interpretará que **hexstring** es un valor **integer** base 16 positivo. El dígito hexadecimal más a la derecha es el menos significativo, el dígito hexadecimal más a la izquierda es el más significativo. El número de dígitos hexadecimales proporcionados serán múltiplos de 2 porque un octeto se compone de dos dígitos hexadecimales. Los dígitos hexadecimales 0 .. F representan los valores decimales 0 .. 15, respectivamente.

D.1.8 Cadena de caracteres a entero

```
str2int(charstring value) return integer
```

Esta función convierte una **charstring** que representa un valor **integer** al **integer** equivalente. Si la cadena no representa un valor entero válido, la función devuelve el valor cero (0).

Ejemplos:

```
str2int("66") devolverá el valor integer 66
str2int("-66") devolverá el valor integer -66
str2int("abc") devolverá el valor integer 0
str2int("0") devolverá el valor integer 0
```

D.1.9 Entero a cadena de bits

```
int2bit(integer value, length) return bitstring
```

Esta función convierte un solo valor **integer** en un solo valor **bitstring**. La cadena resultante tiene una longitud en bits.

A los efectos de conversión, se interpretará que una **bitstring** es un valor **integer** base 2 positivo. El bit más a la derecha es el menos significativo, el bit más a la izquierda es el más significativo. Los bits 0 a 1 representan los valores decimales 0 y 1, respectivamente. Si la conversión da un valor con menos bits que los especificados en el parámetro `length`, la **bitstring** será rellenada a la izquierda con ceros. Se producirá un error si `value` es negativo o si la **bitstring** resultante contiene más bits que los especificados en el parámetro `length`.

D.1.10 Entero a cadena hexadecimal

```
int2hex(integer value, length) return hexstring
```

Esta función convierte un solo valor **integer** a un solo valor **hexstring**. La cadena resultante tiene una longitud de dígitos hexadecimales `length`.

A los efectos de esta conversión, se interpretará que una **hexstring** es un valor **integer** base 16 positivo. El dígito hexadecimal más a la derecha es el menos significativo, el dígito hexadecimal más a la izquierda es el más significativo. Los dígitos hexadecimales 0 .. F representan los valores decimales 0 .. 15 respectivamente. Si la conversión da un valor con menos dígitos hexadecimales que los especificados en el parámetro `length`, **hexstring** será rellenada a la izquierda con ceros. Se producirá un error de caso de prueba si `value` es negativo o si la **hexstring** resultante contiene más dígitos hexadecimales que los especificados en el parámetro `length`.

D.1.11 Entero a cadena de octetos

```
int2oct(integer value, length) return octetstring
```

Esta función convierte un solo valor **integer** en un solo valor **octetstring**. La cadena resultante tiene una longitud de octetos `length`.

A los efectos de esta conversión, se interpretará que una **octetstring** es un valor **integer** base 16 positivo. El dígito hexadecimal más a la derecha es el menos significativo, el dígito hexadecimal más a la izquierda es el más significativo. El número de dígitos hexadecimales proporcionados serán múltiplos de 2 porque un octeto se compone de 2 dígitos hexadecimales. Los dígitos hexadecimales 0 .. F representan los valores decimales 0 .. 15, respectivamente. Si la conversión da un valor con menos dígitos hexadecimales que los especificados en el parámetro `length`, se rellenará **hexstring** a la izquierda con ceros. Se producirá un error de caso de prueba si `value` es negativo o si la **hexstring** resultante contiene más dígitos hexadecimales que los especificados en el parámetro `length`.

D.1.12 Entero a cadena de caracteres

```
int2str(integer value) return charstring
```

Esta función convierte el valor entero en su cadena equivalente (la base de la cadena devuelta es siempre decimal).

Ejemplos:

```
int2str(66) devolverá el valor charstring "66"
int2str(-66) devolverá el valor charstring "-66"
int2str(0) devolverá el valor integer "0"
```

D.1.13 Longitud de tipo de cadena

```
lengthof(any_string_type value) return integer
```

Esta función devuelve la longitud de un valor que es del tipo **bitstring**, **hexstring**, **octetstring**, o cualquier cadena de caracteres. Las unidades de longitud para cada tipo de cadena se definen en el cuadro 4.

Ejemplo:

```
lengthof('010'B) // devuelve 3
lengthof('F3'H)  // devuelve 2
lengthof('F2'O)  // devuelve 1
lengthof ("Length_of _Example") // devuelve 17
```

D.1.14 Número de elementos en un tipo estructurado

```
sizeof(structured_type value) return integer
```

Esta función devuelve el número real de elementos de **record**, **record of**, **set**, **set of**, **template** o matriz.

```
// dado
type record MyPDU
   {  boolean field1,
      integer field2
   }

// entonces
sizeof(MyPDU)
// devuelve 2
```

D.1.15 La función IsPresent

```
ispresent(any_type value) return boolean
```

Esta función devuelve el valor **true** solamente si el valor del campo referenciado está presente en el objeto de datos referenciado real. El argumento de **ispresent** será una referencia a un campo dentro de un objeto de datos que se define como **optional** (facultativo).

```
// Dado
type record MyRecord
   {  boolean  field1 optional,
      integer field2
   }
// y dado que MyPDU es una plantilla de tipo MyRecord
// y la PDU recibida es también de tipo MyRecord
// entonces
MyPort.receive(MyPDU) -> value received_PDU
ispresent(received_PDU.field1)
// devuelve true si field1 está presente en el caso real de MyPDU
```

D.1.16 La función IsChosen

```
ischosen(any_type value) return boolean
```

Esta función devuelve el valor **true** solamente si la referencia de objeto de datos especifica la variante del tipo **union** que se selecciona realmente para un objeto de datos dado.

Ejemplo:

```
// Dado

type union MyUnion
   {  PDU_type1   p1,
      PDU_type2   p2,
      PDU_type p3
   }

// y dado que MyPDU es una plantilla de tipo MyUnion
// y la PDU recibida es también de tipo MyUnion
// entonces
MyPort.receive(MyPDU) -> value received_PDU
ischosen(received_PDU.p2)
// devuelve true si el caso real de MyPDU transporta una PDU del tipo
// PDU_type2
```

Anexo E

Utilización de otros tipos de datos con TTCN-3

E.1 Utilización de ASN.1 con TTCN-3

Este anexo define la utilización facultativa de ASN.1 con TTCN-3.

TTCN-3 proporciona una interfaz limpia para utilizar la versión de 1997 de la ANS.1 (definida en la serie de Recomendaciones UIT-T X.680 [7], [8], [9] y [10]) en módulos TTCN-3. Cuando se importa en un módulo TTCN-3, el identificador de lenguaje para la versión de 1997 de la ASN.1 será "ASN.1:1997".

Cuando se utiliza ASN.1 con TTCN-3, las palabras clave enumeradas en el cuadro E.1 no se utilizarán como identificadores en un módulo TTCN-3. Las palabras clave de ASN.1 cumplirán los requisitos indicados en UIT-T X.680 [7].

Cuadro E.1/Z.140 – Lista de palabras clave de ASN.1

ABSENT	**EMBEDDED**	**INTERSECTION**	**SEQUENCE**
ABSTRACT-SYNTAX	**END**	**Iso10646string**	**SET**
ALL	**ENUMERATED**	**MAX**	**SIZE**
APPLICATION	**EXCEPT**	**MIN**	**STRING**
AUTOMATIC	**EXPLICIT**	**MINUS-INFINITY**	**SYNTAX**
BEGIN	**EXPORTS**	**NULL**	**T61String**
BIT	**EXTERNAL**	**NumericString**	**TAGS**
BMPSTRING	**FALSE**	**OBJECT**	**TeletexString**
BOOLEAN	**FROM**	**ObjectDescriptor**	**TRUE**
BY	**GeneralizedTime**	**OCTET**	**TYPE-IDENTIFIER**
CHARACTER	**GeneralString**	**OF**	**UNION**
CHOICE	**IA5String**	**OPTIONAL**	**UNIQUE**
CLASS	**IDENTIFIER**	**PDV**	**UNIVERSAL**
COMPONENT	**IMPLICIT**	**PLUS-INFINITY**	**UniversalString**
COMPONENTS	**IMPORTS**	**PRESENT**	**UTCTime**
CONSTRAINED	**INCLUDES**	**PrintableString**	**VideotexString**
DEFAULT	**INSTANCE**	**PRIVATE**	**VisibleString**
DEFINITIONS	**INTEGER**	**REAL**	**WITH**

E.1.1 Equivalentes de tipos ASN.1 y TTCN-3

Los tipos ASN.1 enumerados en el cuadro E.2 se consideran equivalentes a sus contrapartidas en TTCN-3.

Cuadro E.2/Z.140 – Lista de equivalentes de tipos ASN.1 y TTCN-3

Tipo ASN.1	Corresponde con el equivalente TTCN-3
BOOLEAN	boolean
INTEGER	integer
REAL	float
OBJECT IDENTIFIER	objid
BIT STRING	bitstring
OCTET STRING	octetstring
SEQUENCE	record
SEQUENCE OF	record of
SET	set
SET OF	set of
ENUMERATED	enumerated
CHOICE	union

Todos los operadores, funciones, mecanismos de concordancia, notación de valores, etc., de TTCN-3 que pueden ser utilizados con un tipo TTCN-3 indicado en el cuadro E.2 pueden ser utilizados también con el correspondiente tipo ASN.1.

E.1.2 Tipos y valores de datos ASN.1

Los tipos y valores ASN.1 se pueden utilizar en módulos TTCN-3. Las definiciones ASN.1 se hacen utilizando un módulo ASN.1 separado.

Ejemplo:

```
MyASN1module DEFINITIONS ::=
BEGIN
       Z::=   INTEGER              -- Definición de tipo simple


       BMessage::= SET             -- Definición de tipo ASN.1
       {
          name Name,
          title  VisibleString,
          date Date
       }

       johnValues Bmessage ::=     -- Definición de valor ASN.1
       {
          name "John Doe",
          title  "Mr",
          date "April 12th"
       }
END
```

El módulo ASN.1 será escrito de acuerdo con la sintaxis de la serie de Recomendaciones UIT-T X.680 [7], [8], [9] y [10]). Una vez declarados, los tipos y valores ASN.1 pueden ser utilizados dentro de módulos TTCN-3 exactamente de la misma manera que los tipos y valores TTCN-3 ordinarios de otros módulos TTCN-3 (es decir, se importarán las definiciones requeridas).

Ejemplo:

```
module MyTTCNModule
{
      import all from MyASN1module language "ASN.1:1997";

      const Bmessage MyTTCNConst:= johnValues;
}
```

NOTA – Las definiciones ASN.1 que no son tipos ni valores (es decir, clases de objeto de información o conjuntos de objetos de información) no son accesibles directamente desde la notación TTCN-3. Estas definiciones se resolverán en un tipo o valor dentro del módulo ASN.1 antes de que puedan ser referenciadas desde el interior del módulo TTCN-3.

E.1.2.1 Alcance de identificadores ASN.1

Los identificadores ASN.1 importados siguen las mismas reglas de alcance que los tipos y valores TTCN-3 importados (véase 5.4).

E.1.3 Parametrización en ASN.1

Se permite hacer referencia a definiciones de tipos y valores ASN.1 parametrizados desde el módulo TTCN-3. Sin embargo, todas las definiciones parametrizadas ASN.1 utilizadas en un módulo TTCN-3 tendrán parámetros reales (no se permiten tipos o valores abiertos) y los parámetros reales proporcionados serán resueltos en el tiempo de compilación.

El lenguaje núcleo de TTCN-3 no soporta parametrización de objetos específicos ASN.1 únicamente. Por consiguiente, la parametrización específica de la ASN.1, que comprende objetos que no pueden ser definidos directamente en el lenguaje núcleo de TTCN-3, será resuelta en la parte ASN.1 antes de utilizarla dentro de TTCN-3. Los objetos específicos de la ASN.1 son:

a) Conjunto de valores.

b) Clases de objetos de información.

c) Objetos de información.

d) Conjuntos de objetos de información.

Por ejemplo, la siguiente definición no es legal porque define un tipo TTCN-3 que toma un conjunto de objetos ASN.1 como un parámetro real:

```
MyASN1module DEFINITIONS ::=
BEGIN
     -- Definición de módulo ASN.1
     -- Definición de clase de objeto de información
     MESSAGE ::= CLASS { &msgTypeValueINTEGER UNIQUE,
                         &MsgFields}
     -- Definición de objeto de información
     setupMessage MESSAGE ::=          {    &msgTypeValue  1,
                  &MsgFields         OCTET STRING}
       setupAckMessage MESSAGE ::=     {    &msgTypeValue  2,
                  &MsgFields         BOOLEAN}
     -- Definición de conjunto de objetos de información
     MyProtocol MESSAGE ::= { setupMessage | setupAckMessage}

     -- Tipo ASN.1 constreñido por conjunto de objetos
     MyMessage{ MESSAGE : MsgSet} ::= SEQUENCE
     {
       code   MESSAGE.&msgTypeValue({ MsgSet}),
       Type   MESSAGE.&MsgFields({ MsgSet})
}
END

module MyTTCNModule
{
   // Definición de módulo TTCN-3
   import all from MyASN1module language "ASN.1:1997";

   // Tipo TTCN-3 ilegal con conjunto de objetos como parámetro
   type record Q(MESSAGE MyMsgSet) ::= {   Z       field1,
                              MyMessage(MyMsgSet) field2}
}
```

Para que esta definición sea legal, hay que definir el tipo ASN.1 suplementario My Message1, como se muestra a continuación. Esto resuelve la parametrización del conjunto de objetos de información y por tanto puede ser usado directamente en el módulo TTCN-3.

```
MyASN1module DEFINITIONS ::=

BEGIN
     -- Definición de módulo ASN.1

     ...

     MyProtocol MESSAGE ::= { setupMessage | setupAckMessage}

     -- Tipo ASN.1 suplementario para suprimir parametrización de conjunto
     -- de objetos
     MyMessage1 ::= MyMessage{ MyProtocol}
END
```

```
module MyTTCNModule
{
   // Definición de módulo TTCN-3
   import all from MyASN1module language "ASN.1:1997";

   // Tipo TTCN-3 legal sin conjunto de objetos como parámetro
   type record Q := {  Z          field1,
                       MyMessage1 field2}
}
```

E.1.4 Definición de tipos de mensajes con ASN.1

Los mensajes ASN.1 se definen utilizando SEQUENCE (o posiblemente SET).

Ejemplo:

```
MyASN1module DEFINITIONS ::=
BEGIN
     -- Definición de módulo ASN.1

     MyMessageType ::= SEQUENCE
     {
       field1  Field1Type,
       field2  Field2Type    OPTIONAL, -- Se puede omitir este campo
       :
       fieldN  FieldNType
     }
END
```

Naturalmente, los mensajes que se definen utilizando la ASN.1 pueden también ser subestructurados empleando SEQUENCE, SET, etc.

E.1.5 Definición de plantillas de mensajes ASN.1

Si se definen mensajes en ASN.1 utilizando, por ejemplo: SEQUENCE (o posiblemente SET), los mensajes reales, para los eventos **send** y **receive**, pueden ser especificados con la sintaxis de valores de ASN.1.

Ejemplo:

```
MyASN1module DEFINITIONS ::=
BEGIN
     -- Definición de módulo ASN.1

     -- La definición de mensaje
MyMessageType::= SEQUENCE
     { field1 [1] IA5STRING,           // Como cadena de caracteres TTCN-3
       field2 [2] INTEGER OPTIONAL,    // Como entero TTCN-3
       field3 [4] Field3Type,          // Como registro TTCN-3
       field4 [5] Field4Type           // Como matriz TTCN-3
     }

     Field3Type::= SEQUENCE {field31 BIT STRING, field32 INTEGER, field33 OCTET
       STRING},
     Field4Type::= SEQUENCE OF BOOLEAN
```

```
    -- puede tener el siguiente valor
    myValue MyMessageType::=
    {
      field1  "A string",
      field2  123,
      field3  {field31 '11011'B, field32 456789, field33 'FF'O},
      field4  {true, false}
    }
END
```

E.1.5.1 Recepción de mensajes ASN.1 con la sintaxis de plantillas de TTCN-3

En la sintaxis ASN.1 normalizada no se soportan mecanismos de concordancia. Por consiguiente, si se debe utilizar mecanismos de concordancia en la recepción de un mensaje ASN.1, habrá que utilizar en cambio la sintaxis TTCN-3 para plantillas en recepción. Obsérvese que esta sintaxis incluye referencias de componentes para poder hacer referencia a los componentes individuales en SEQUENCE, SET, etc., de ASN.1.

Ejemplo:

```
import type myMessageType from MyASN1module language "ASN.1:1997";

// una plantilla de mensaje que utiliza mecanismos de concordancia dentro de
// TTCN-3 pudiera ser
template myMessageType MyValue:=
{
   field1 :=        "A"<?>"tr"<*>"g",
   field2 :=        *,
   field3.field31 :='110??'B,
   field3.field32 :=?,
   field3.field33 :='F?'O,
   field4.[0] :=    true,
   field4.[1] :=    false
}

// // la siguiente sintaxis es igualmente válida
template myMessageType MyValue:=
{
   field1 := "A"<?>"tr"<*>"g",    // cadena con comodines
   field2 := *,                   // cualquier entero o ninguno
   field3 := {'110??'B, ?, 'F?'O},
   field4 := {?, false}
}
```

E.1.5.2 Ordenación de campos de plantilla

Cuando se utilizan plantillas TTCN-3 para tipos ASN.1, el significado del orden de los campos en la plantilla dependerá del tipo de construcción ASN.1 utilizada para definir el tipo de mensaje. Por ejemplo: si se utiliza SEQUENCE o SEQUENCE OF, los campos de mensajes serán enviados o concordados en el orden especificado en la plantilla. Si se utiliza SET o SET OF, los campos de mensajes pueden ser enviados o concordados en cualquier orden.

E.1.6 Información de codificación

TTCN-3 permite hacer referencia a reglas de codificación y variaciones dentro de las reglas de codificación que han de ser asociadas con distintos elementos de lenguaje de TTCN-3. Es posible también definir codificaciones inválidas. Esta información de codificación se especifica utilizando el enunciado **with** de acuerdo con la siguiente sintaxis:

Ejemplo:

```
module MyModule
{  :
   import type myMessageType from MyASN1module language "ASN.1:1997" with
      {encode:= "PER:1997"}
               // Todos los ejemplares de MyMessageType deben ser codificados
               // utilizando PER:1997
               // PER:1997

} with {encode "BER:1997"}  // La codificación por defecto para todo el módulo
                            // (serie de pruebas) es BER:1997
```

E.1.6.1 Atributos de codificación ASN.1

Las siguientes cadenas son los atributos de codificación predefinidos (normalizados) para ASN.1:

a) "BER:1997" significa codificado de acuerdo con UIT-T X.690 (BER) [11].

b) "CER:1997" significa codificado de acuerdo con UIT-T X.690 (CER) [11].

c) "DER:1997" significa codificado de acuerdo con UIT-T X.690 (DER) [11].

d) "PER-BASIC-UNALIGNED:1997" significa codificado de acuerdo con UIT-T X.691 (PER no alineada) [1997].

e) "PER-BASIC-ALIGNED:1997" significa codificada de acuerdo con UIT-T X.691 (PER alineada) [1997].

f) "PER-CANONICAL-UNALIGNED:1997" significa codificada de acuerdo con UIT-T X.691 [1997] (PER no alineada canónica).

g) "PER-CANONICAL-ALIGNED:1997" significa codificada de acuerdo con UIT-T X.691 (PER alineada canónica) [1997].

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**

Impreso en Suiza
Ginebra, 2002