Unión Internacional de Telecomunicaciones

UIT-T Z.140

(03/2006)

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 de prueba y de control de prueba

Notación de pruebas y de control de pruebas versión 3: Lenguaje núcleo

Recomendación UIT-T Z.140

UIT-T

ITU Unión Internacional de Telecomunicaciones

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
Lenguaje ampliado de definición de objetos	Z.130–Z.139
Notación de prueba y de control de prueba	**Z.140–Z.149**
Notación de requisitos de usuarios	Z.150–Z.159
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.349
Interfaces hombre-máquina orientadas a datos	Z.350–Z.359
Interfaces hombre-máquina para la gestión de las redes de telecomunicaciones	Z.360–Z.379
CALIDAD	
Calidad de soportes lógicos de telecomunicaciones	Z.400–Z.409
Aspectos de la calidad de las Recomendaciones relativas a los protocolos	Z.450–Z.459
MÉTODOS	
Métodos para validación y pruebas	Z.500–Z.519
SOPORTE INTERMEDIO	
Entorno del procesamiento distribuido	Z.600–Z.609

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

Recomendación UIT-T Z.140

Notación de pruebas y de control de pruebas versión 3: Lenguaje núcleo

Resumen

Esta Recomendación define el lenguaje núcleo de la versión 3 de la notación de pruebas y de control de pruebas TTCN-3 creada para especificar sucesiones de pruebas que no dependen de la plataforma, los métodos de prueba, los protocolos y las capas de protocolo. La notación TTCN-3 se puede utilizar para la especificación de todos los tipos de pruebas de sistemas reactivos en diversos puertos de comunicación. Las principales aplicaciones son las pruebas de protocolos (incluidos los protocolos móviles e Internet), 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) y las pruebas de interfaces de programas de aplicación (API). La especificación de series de pruebas para protocolos de capa física está fuera del ámbito de esta Recomendación.

Hay distintos formatos de presentación para el lenguaje núcleo de la TTCN-3. En esta Recomendación se describe el lenguaje núcleo, en la Rec. UIT-T Z.141 el formato tabular de la TTCN (TFT) y en la Rec. UIT-T Z.142 el formato gráfico de la TTCN (GFT). La especificación de estos formatos está fuera del ámbito de la presente Recomendación. El lenguaje núcleo tiene tres funciones:

1) lenguaje de prueba textual generalizado;
2) formato de intercambio normalizado de las series de pruebas TTCN entre herramientas TTCN;
3) base semántica (y base de sintaxis donde procede) para los distintos formatos de presentación.

El lenguaje núcleo se puede utilizar independientemente de los formatos de presentación, pero los formatos tabular y gráfico no se pueden utilizar sin el lenguaje núcleo. El lenguaje núcleo determina la utilización y la implementación de estos formatos de presentación.

Orígenes

La Recomendación UIT-T Z.140 fue aprobada el 16 de marzo de 2006 por la Comisión de Estudio 17 (2005-2008) del UIT-T por el procedimiento de la Recomendación UIT-T A.8.

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.

La observancia de esta Recomendación es voluntaria. Ahora bien, la Recomendación puede contener ciertas disposiciones obligatorias (para asegurar, por ejemplo, la aplicabilidad o la interoperabilidad), por lo que la observancia se consigue con el cumplimiento exacto y puntual de todas las disposiciones obligatorias. La obligatoriedad de un elemento preceptivo o requisito se expresa mediante las frases "tener que, haber de, hay que + infinitivo" o el verbo principal en tiempo futuro simple de mandato, en modo afirmativo o negativo. El hecho de que se utilice esta formulación no entraña que la observancia se imponga a ninguna de las partes.

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 en la dirección http://www.itu.int/ITU-T/ipr/.

ÍNDICE

Página

Recomendación UIT-T Z.140

Notación de pruebas y de control de pruebas versión 3: Lenguaje núcleo

1 Alcance

Esta Recomendación define el lenguaje núcleo de la notación de pruebas y de control de pruebas TTCN-3 (*testing and test control notation 3*). La notación TTCN-3 se puede utilizar para la especificación de todos los tipos de pruebas de sistemas reactivas en distintos puertos de comunicación. Las aplicaciones típicas 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 o las interfaces de programación de aplicación (API, *application programming interface*). La 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 sucesiones de pruebas para protocolos de capa física está fuera del ámbito de la presente Recomendación.

La notación TTCN-3 se utilizará para especificar sucesiones de pruebas que son independientes de los métodos de prueba, las capas y los protocolos. Se definen varios formatos de presentación para TTCN-3: un formato de presentación tabular (Rec. UIT-T Z.141 [1]) y un formato de presentación gráfica (Rec. UIT-T Z.142 [2]). La especificación de estos formatos está fuera del ámbito de la presente Recomendación.

Si bien se ha tomado en consideración la implementación de traductores y compiladores TTCN-3 en el diseño de la notación, no entra en el ámbito de la presente Recomendación definir la forma de crear sucesiones de pruebas ejecutables (ETS, *executable test suites*) a partir de sucesiones de pruebas abstractas (ATS, *abstract test suites*).

2 Referencias

Las siguientes Recomendaciones del UIT-T y otras referencias contienen disposiciones que, mediante su referencia en este texto, constituyen disposiciones de la presente Recomendación. Al efectuar esta publicación, estaban en vigor las ediciones indicadas. Todas las Recomendaciones y otras referencias son objeto de revisiones por lo que se preconiza que los usuarios de esta Recomendación investiguen la posibilidad de aplicar las ediciones más recientes de las Recomendaciones y otras referencias citadas a continuación. Se publica periódicamente una lista de las Recomendaciones UIT-T actualmente vigentes. En esta Recomendación, la referencia a un documento, en tanto que autónomo, no le otorga el rango de una Recomendación.

[1] Recomendación UIT-T Z.141 (2006), *Notación de pruebas y de control de pruebas versión 3: Formato de presentación tabular*.

[2] Recomendación UIT-T Z.142 (2006), *Notación de pruebas y de control de pruebas versión 3: Formato de presentación gráfico*.

[3] Recomendación UIT-T Z.143 (2006), *Notación de pruebas y de control de pruebas versión 3: Semántica operacional*.

[4] Recomendación UIT-T Z.144 (2006), *Notación de pruebas y de control de pruebas versión 3: Interfaz de ejecución*.

[5] Recomendación UIT-T Z.145 (2006), *Notación de pruebas y de control de pruebas versión 3: Interfaz de control*.

[6] Recomendación UIT-T Z.146 (2006), *Notación de pruebas y de control de pruebas versión 3: Utilización de ASN.1 con TTCN-3*.

[7] 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*.

[8] Recomendación UIT-T X.292 (2002), *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)*.

[9] Recomendación UIT-T T.50 (1992), *Alfabeto internacional de referencia (anteriormente alfabeto internacional N.º 5 o IA5) – Tecnología de la información – Juego de caracteres codificado de siete bits para intercambio de información*.

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

[10] ISO/CEI 10646:2003, *Information technology – Universal Multiple-Octet Coded Character Set (UCS)*.

[11] ISO/CEI 6429:1992, *Information technology – Control functions for coded character sets*.

3 Definiciones y abreviaturas

3.1 Definiciones

A los efectos de esta Recomendación, se aplican los términos y las definiciones de las Recs. UIT-T X.290 [7], UIT-T X.292 [8], así como los siguientes términos y definiciones:

3.1.1 parámetro efectivo: Valor, expresión, plantilla o referencia de nombre (identificador) que se transferirá como parámetro a la entidad invocada (función, caso de prueba, altstep, etc.) y que ha sido definido en el lugar de invocación.

NOTA – El número, el orden y el tipo de parámetros considerados que se van a transferir en una sola invocación deberán coincidir con la lista de parámetros formales definidos en la entidad invocada.

3.1.2 tipos básicos: Conjunto de tipos TTCN-3 previamente definidos, descritos en 6.1.0 y 6.1.1.

NOTA – Para hacer referencia a un tipo básico se utiliza el nombre correspondiente.

3.1.3 tipo compatible: La notación TTCN-3 no tiene una especificación de tipos rigurosa, pero el lenguaje requiere compatibilidad de tipos.

NOTA – Los tipos de variables, constantes, plantillas, etc., son compatibles si se cumplen las condiciones de 6.7.

3.1.4 puerto de comunicación: Mecanismo abstracto que facilita la comunicación entre componentes de prueba.

NOTA – El modelo de un puerto de comunicación es una cola FIFO (primero en entrar primero en salir) en el sentido de recepción. Pueden ser puertos por mensajes, puertos por procedimientos o mixtos.

3.1.5 tipos de datos: Nombre común de tipos básicos simples, tipos de cadenas básicos, tipos estructurados, el tipo especial de datos **`anytype`** y todos los tipos que el usuario define a partir de estos (véase el cuadro 3).

3.1.6 tipos definidos (tipos TTCN-3 definidos): Conjunto de todos los tipos TTCN-3 previamente definidos (tipos básicos, todos los tipos estructurados, el tipo **`anytype`**, la dirección, los tipos puerto y componente y el tipo por defecto) así como todos los tipos definidos por el usuario declarados en el módulo o importados de otros módulos TTCN-3.

3.1.7 parametrización dinámica: Clase de parametrización en la que los parámetros efectivos dependen de eventos durante la ejecución, por ejemplo el valor del parámetro efectivo es un valor recibido durante la ejecución o depende de un valor recibido conforme a una relación lógica.

3.1.8 excepción: En los casos de comunicación por procedimientos, es la entidad respondedora la que plantea una excepción (si ha sido definida) cuando no puede responder a una solicitud de procedimiento distante con la respuesta prevista normal.

3.1.9 parámetro formal: Referencia (identificador) de nombre de tipo o de plantilla de tipo que no se traduce al definir una entidad (función, caso de prueba, altstep, etc.) sino al invocarla.

NOTA – Los valores o plantillas efectivos (o sus nombres) que se han de utilizar en lugar de los parámetros formales se transfieren desde el lugar de invocación de la entidad (véase la definición de parámetros efectivo).

3.1.10 visibilidad global: Atributo de una entidad (parámetro, constante, plantilla de módulo, etc.) cuyo identificador se puede introducir como referencia en cualquier lugar del módulo donde se define, incluidas todas sus funciones, casos de prueba y altsteps definidos dentro del mismo módulo y la parte de control de ese módulo.

3.1.11 declaración de conformidad de implementación (ICS, *implementation conformance statement*): Véase la Rec. UIT-T X.290 [7].

3.1.12 información suplementaria de implementación para las pruebas (IXIT, *implementation extra information for testing*): Véase la Rec. UIT-T X.290 [7].

3.1.13 implementación sometida a prueba (IUT, *implementation under test*): Véase la Rec. UIT-T X.290 [7].

3.1.14 tipos conocidos: Conjunto de todos los tipos predefinidos TTCN-3, tipos definidos en un módulo TTCN-3 y tipos importados a ese módulo de otros módulos TTCN-3 o de módulos que no son TTCN-3.

3.1.15 lado izquierdo (de una asignación): Identificador variable de valor o plantilla o nombre de campo de un valor de tipo o variable de plantilla estructurado (incluyendo un índice de matrices, si lo hubiere), que se encuentra a la izquierda de un símbolo de asignación (:=).

NOTA – Una constante, un parámetro de módulo, un temporizador, un nombre de campo de tipo estructurado o un encabezamiento de plantilla (incluyendo la lista de parámetros de tipo, nombre y formales de plantilla), que se encuentran a la izquierda de un símbolo de asignación (:=) en declaraciones y/o definiciones de plantilla modificadas, no entran en el ámbito de la presente definición por no formar parte de una asignación.

3.1.16 visibilidad local: Atributo de una entidad (constante, variable, etc.) cuyo identificador sólo puede mencionarse dentro de la función, el caso de prueba o **`altstep`** en los que ha sido definido.

3.1.17 componente de prueba principal (MTC, *main test component*): Véase la Rec. UIT-T X.292 [8].

3.1.18 transferencia de parámetro por valor: Forma de transferir parámetros en la que se hace una evaluación de los argumentos antes de entrar en una entidad parametrizable.

NOTA – Sólo se comunican los valores de los argumentos, y las modificaciones de argumentos en la entidad llamada no tienen efecto en los argumentos efectivos para la entidad llamante.

3.1.19 transferencia de parámetro por referencia: Forma de comunicar parámetros en la que no se hace una evaluación de los argumentos antes de entrar en la función, altstep, etc.; la referencia del parámetro (función, altstep, etc.) se transfiere desde el procedimiento solicitante hacia el procedimiento solicitado.

NOTA – Todas las modificaciones de argumentos en el procedimiento solicitado tienen efecto en los argumentos efectivos para la entidad llamante.

3.1.20 componente de prueba paralelo (PTC, *parallel test component*): Véase la Rec. UIT-T X.292 [8].

3.1.21 lado derecho (de una asignación): Expresión, identificador de parámetro de referencia de plantilla o de firma que se encuentra del lado derecho de un símbolo de asignación (:=).

NOTA – Las expresiones y las referencias de plantilla que se encuentran del lado derecho de un símbolo de asignación (:=) en declaraciones de constante, parámetro de módulo, temporizador, plantilla o plantilla modificada, no entran en el ámbito de la presente definición por no formar parte de una asignación.

3.1.22 tipo raíz: Tipo básico, tipo estructurado, tipo de dato especial, tipo de configuración especial o tipo por defecto especial que son la base de un tipo TTCN-3 definido por el usuario.

3.1.23 parametrización estática: Tipo de parametrización en la que los parámetros efectivos no dependen de eventos durante la ejecución: se conocen al compilar o en el caso parámetros de módulo al empezar a ejecutar la serie de pruebas (por ejemplo, se indican en las especificaciones de la sucesión de pruebas, incluidas las definiciones importadas, o el sistema de prueba conoce su valor antes de la ejecución).

NOTA – Todos los tipos se conocen al compilar, es decir, están unidos estáticamente.

3.1.24 definición de tipos rigurosa: Aplicación rigurosa de la compatibilidad de tipos por equivalencia de nombres de tipos, sin excepciones.

3.1.25 sistema sometido a prueba (SUT, *system under test*): Véase la Rec. UIT-T X.290 [7].

3.1.26 plantilla: Las plantillas de la notación TTCN-3 son estructuras de datos específicas para pruebas, que se utilizan para transmitir un conjunto de valores diferentes o comprobar si un conjunto de valores recibidos concuerda con la especificación de la plantilla.

3.1.27 comportamiento de prueba: (o comportamiento) Caso de prueba o función iniciada en un componente de prueba cuando se ejecuta un enunciado de componente **`execute`** o **`start`** y se solicitan todas las funciones y altsteps recurrentemente

NOTA – Durante la ejecución de un caso de prueba cada componente de prueba tiene su propio comportamiento y por consiguiente varios comportamientos de prueba pueden funcionar simultáneamente en el sistema de prueba (es decir, un caso de prueba puede considerarse como un grupo de comportamientos de prueba).

3.1.28 Caso de prueba: Véase la Rec. UIT-T X.290 [7].

3.1.29 error de caso de prueba: Véase la Rec. UIT-T X.290 [7].

3.1.30 sucesión de pruebas: Conjunto de módulos TTCN-3 que contiene un conjunto de casos de prueba completamente definido y suplementado facultativamente con una o varias partes de control TTCN-3.

3.1.31 sistema de prueba: Véase la Rec. UIT-T X.290 [7].

3.1.32 interfaz de sistema de prueba: Componente de prueba que relaciona los puertos disponibles en el sistema de prueba TTCN-3 (abstracto) y los puertos del SUT.

3.1.33 compatibilidad de tipos: Característica del lenguaje que permite utilizar valores, expresiones o plantillas de un determinado tipo como valores efectivos de otro tipo (por ejemplo al definir asignaciones, como parámetros efectivos al invocar una función, al mencionar una plantilla, etc. o el valor que devuelve una función).

NOTA – Es condición que tanto el tipo como la definición actual del valor, la expresión o la plantilla sean compatibles con el otro tipo.

3.1.34 parametrización de valores: Capacidad de transferir un valor o plantilla como parámetro efectivo en un objeto parametrizado.

NOTA – Este parámetro de valor efectivo completa la especificación de ese objeto.

3.1.35 tipo definido por el usuario: Tipo definido como variante de un tipo básico declarando un tipo estructurado.

NOTA – Para hacer referencia a los tipos definido por el usuario se utilizan sus identificadores (nombres).

3.1.36 notación de valor: Notación que asocia un identificador a un determinado valor o una gama de un determinado tipo.

NOTA – Los valores pueden ser constantes o variables.

3.2 Abreviaturas, siglas o acrónimos

En esta Recomendación se utilizan las siguientes abreviaturas, siglas o acrónimos.

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

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

BMP Plano multilingüe básico (*basic multilingual plane*)

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*)

ICS Declaración de conformidad de implementación (*implementation conformance statement*)

IRV Versión internacional de referencia (*international reference version*)

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

IXIT Información suplementaria de implementación para pruebas (*implementation eXtra information for testing*)

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

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

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

TSI Interfaz de sistema de prueba (*test system interface*)

4 Introducción

4.0 Consideraciones generales

La notación TTCN-3 es un lenguaje flexible y muy completo para especificar toda clase de pruebas de sistemas reactivos en distintas interfaces de comunicación. Las aplicaciones típicas son la prueba de protocolos (incluidos los protocolos móvil e Internet), la prueba de servicios (incluidos los servicios suplementarios), la prueba de módulos, la prueba de plataformas basadas en CORBA, la prueba de API, etc. Esta notación no está restringida a la prueba de conformidad y se puede utilizar para muchas otras clases de pruebas: interoperabilidad, robustez, regresión, sistemas e integración.

Las características esenciales de la notación TTCN-3 son:

- la capacidad de especificar configuraciones dinámicas de pruebas concurrentes;
- operaciones para la comunicación por procedimientos o por mensajes;
- la capacidad de especificar información de codificación y otros atributos (incluyendo una ampliación por parte del usuario);

- la capacidad de especificar plantillas de datos y de firmas con sutiles mecanismos de concordancia;
- la parametrización de valores;
- la asignación y el tratamiento de veredictos de prueba;
- mecanismos para parametrizar sucesiones de pruebas y seleccionar casos de prueba;
- la utilización combinada de TTCN-3 y otros lenguajes;
- sintaxis, formato de intercambio y semántica estática claramente definidos;
- distintos formatos de presentación (por ejemplo, formatos de presentación tabular y gráfica);
- un algoritmo de ejecución preciso (semántica operacional).

4.1 El lenguaje núcleo y los formatos de presentación

La especificación de TTCN-3 se divide en varias partes. La primera parte definida en esta Recomendación es el lenguaje núcleo. La segunda parte, definida en la Rec. UIT-T Z.141 [1], es el formato de presentación tabular. La tercera parte, definida en la Rec. UIT-T Z.142 [2], es el formato de presentación gráfica. La cuarta parte, definida en la Rec. UIT-T Z.143 [3], incluye la semántica operacional del lenguaje. La quinta parte, definida en la Rec. UIT-T Z.144 [4], describe la interfaz de tiempo de ejecución de TTCN-3 (TRI, *TTCN-3 runtime interface*). La sexta parte, definida en la Rec. UIT-T Z.145 [5], describe las interfaces de control de TTCN-3 (TCI, *TTCN-3 control interfaces*). La séptima parte, Rec. UIT-T Z.146 [6] especifica la utilización de las definiciones de ASN.1 con TTCN-3.

El lenguaje núcleo tiene tres funciones:

a) es un lenguaje de prueba textual generalizado;

b) es un formato de intercambio normalizado de sucesiones de pruebas TTCN-3 entre herramientas de la TTCN-3;

c) es una base semántica (y base de sintaxis cuando procede) para distintos 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 se pueden utilizar sin el lenguaje núcleo. La utilización 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: formatos de presentación normalizados o formatos de presentación de dominio privado definidos por los propios usuarios de TTCN-3. Estos formatos adicionales no se definen en la presente Recomendación.

La notación TTCN-3 se puede utilizar facultativamente con otras notaciones de tipo-valor; en ese caso pueden emplearse definiciones en otros lenguajes como sintaxis alternativa de tipos de datos y de valores. En otras partes de esta norma se especifica la utilización de algunos otros lenguajes con TTCN-3. El soporte de otros lenguajes no está limitado a los que se especifican en la serie de Recomendaciones Z.140, pero para soportar lenguajes para los que está definido el uso combinado con TTCN-3 se aplicarán las normas establecidas en esta Recomendación.

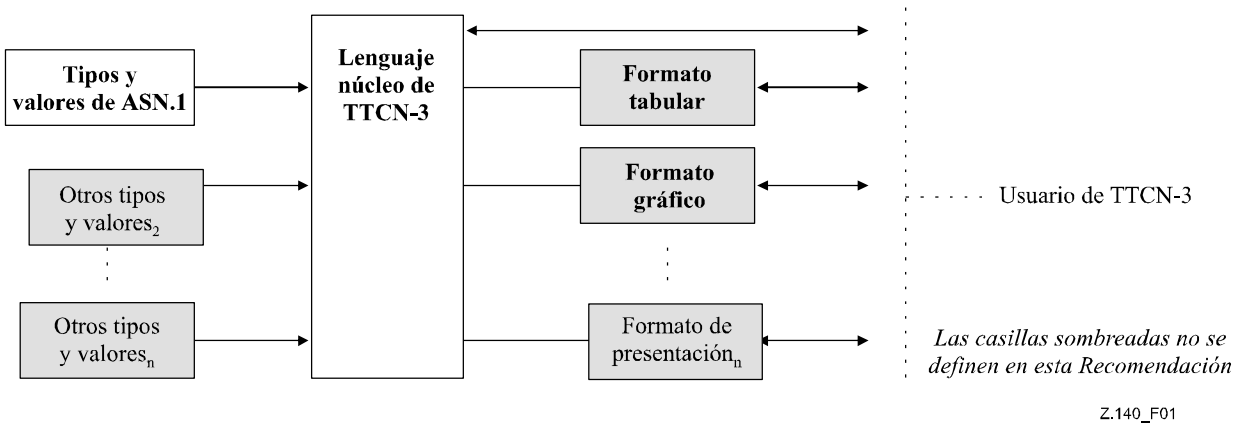


Figura 1/Z.140 – Visión del usuario del lenguaje núcleo y de los diversos formatos de presentación

El lenguaje núcleo se define mediante una sintaxis completa (véase el anexo A) y la semántica operacional (Rec. UIT-T Z.143 [3]). Contiene la semántica estática mínima (definida en el texto de esta Recomendación y el anexo A) que no restringe el uso del lenguaje a ningún dominio de aplicación ni metodología subyacente.

4.2 Unanimidad de la especificación

La semántica y la sintaxis del lenguaje se han especificado mediante un texto descriptivo en esta Recomendación (cláusulas 5 a 28) y también formalmente en el anexo A. La descripción formal completa el texto descriptivo cada vez que es necesario. La especificación formal prevalecerá si hay discrepancias con el texto descriptivo.

4.3 Conformidad

Sólo podrá considerarse que una implementación es conforme a esta versión del lenguaje si se observan coherentemente todos los requisitos de esta Recomendación y de la Rec. UIT-T Z.143 [3] en las características de implementación.

5 Elementos de lenguaje básicos

5.0 Consideraciones generales

La unidad de nivel superior TTCN-3 es un módulo. Un módulo no se puede estructurar en submódulos. Un módulo puede importar definiciones de otros módulos. Los módulos pueden tener parámetros de módulo para facilitar la parametrización de sucesiones de pruebas.

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 estructuradas por procedimientos en los puertos, casos de prueba, etc.

La parte de control de un módulo solicita los casos de prueba y controla su ejecución. La parte de control también puede 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 cada caso de prueba. La TTCN-3 no soporta el concepto de variables globales.

La notación TTCN-3 tiene varios tipos de datos básicos predefinidos y también tipos estructurados, tales como registros (*records*), conjuntos (*sets*), uniones (*unions*), tipos enumerados (*enumerated types*) y matrices (*arrays*).

La plantilla (*template*) es una estructura de datos especial que proporciona mecanismos de parametrización y concordancia para especificar los datos de prueba que se han de enviar o recibir por los puertos de prueba. Las operaciones en estos puertos proporcionan capacidades de comunicación por procedimientos o por mensajes. Se podrán utilizar llamadas por procedimientos para probar implementaciones que no se basan en mensajes.

Se habla de casos de prueba para referirse al comportamiento de prueba dinámico. Las instrucciones de programa de la notación TTCN-3 comprenden mecanismos muy completos para describir el comportamiento: recepción alternativa de eventos de comunicación y de temporización, entrelazado y comportamiento por defecto. También soporta mecanismos de asignación y registro de veredictos de prueba.

Por último, es posible asignar información de codificación, atributos de visualización y otros atributos a los elementos de lenguaje de TTCN-3. También pueden especificarse atributos definidos por el usuario (no normalizados).

Cuadro 1/Z.140 – Síntesis 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/ altsteps/casos de prueba	Especificado en el tipo de componente 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			

Cuadro 1/Z.140 – Síntesis 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/ altsteps/casos de prueba	Especificado en el tipo de componente de prueba
Definiciones de constantes	**`const`**	Sí	Sí	Sí	Sí
Definiciones de plantillas de datos/firmas	**`template`**	Sí	Sí	Sí	Sí
Definiciones de funciones	**`function`**	Sí			
Definiciones de alternativas (altsteps)	**`altstep`**	Sí			
Definiciones de casos de prueba	**`testcase`**	Sí			
Declaraciones de valores variables	**`var`**		Sí	Sí	Sí
Declaraciones de plantillas variables	**`var template`**		Sí	Sí	Sí
Declaraciones de temporizadores	**`timer`**		Sí	Sí	Sí
NOTA – Las nociones de "definición" y "declaración" de variables, constantes, tipos y otros elementos de lenguaje se emplean de manera indistinta en toda esta Recomendación. La distinción entre ambas nociones sólo es útil para fines de implementación, como es el caso en lenguajes de programación como C++. En el nivel de TTCN-3, dichas nociones tienen el mismo significado.					

5.1 Ordenación de elementos de lenguaje

En general, el orden de las declaraciones es arbitrario, pero dentro de un bloque de instrucciones y declaraciones, por ejemplo en el texto de una función o una sección de una instrucción **`if-else`**, todas las declaraciones (en su caso) deben aparecer al principio del bloque.

EJEMPLO:

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

 :
```

Las declaraciones pueden aparecer en cualquier orden en la parte de definiciones del módulo. No obstante, dentro de la parte de control del módulo las definiciones de casos de prueba, las funciones, las altsteps y todas las declaraciones necesarias, deben obtenerse con antelación. Esto significa que en particular, las variables locales, los temporizadores locales y las constantes locales no deben utilizarse en ningún caso antes de que sean declarados. Las etiquetas son la única excepción a esta regla. Es posible hacer referencias hacia adelante a una etiqueta en las declaraciones **`goto`** antes de que aparezca la etiqueta (véase 19.5).

5.2 Parametrización

5.2.0 Parametrización estática y dinámica

La notación TTCN-3 soporta la parametrización de *valores* con las siguientes limitaciones:

a) los elementos de lenguaje que no se pueden parametrizar 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 sucesiones de pruebas: puede ser una parametrización traducible o no en el momento de la compilación, pero se ha de traducir antes de empezar la ejecución (parametrización *estática* en el momento de ejecución). Por tanto, los valores de parámetros del módulo son visibles globalmente en el momento de ejecución, pero no pueden modificarse;

c) todas las definiciones de **`type`** (**tipo**) del usuario (incluidas las definiciones de tipos estructurados tales como **`record`** (**registro**), **`set`** (**conjunto**), etc.) y el tipo de configuración especial **`address`** (**dirección**) soportan la parametrización de valores *estática*: parametrización que se ha traducir en el momento de la compilación;

d) los elementos de lenguaje **`template`** (**plantilla**), **`signature`** (**firma**), **`testcase`** (**caso de prueba**), **`altstep`** (**alternativa**) y **`function`** (**función**) soportan la parametrización de valores *dinámica* (es decir, parametrización traducible en el momento de ejecución).

En el cuadro 2 se indican los elementos de lenguaje que se pueden parametrizar y lo que se puede transferir a ellos como parámetros.

Cuadro 2/Z.140 – Los elementos del lenguaje TTCN-3 parametrizables

Palabra clave	Parametrización de valores	Tipos de valores que pueden aparecer en listas de parámetros formales/efectivos
`module`	Estática al empezar la ejecución	*Valores de*: todos los tipos básicos, todos los tipos definidos por el usuario y el tipo **`address`**.
`type` (nota 1)	Estática al hacer la compilación	*Valores de*: todos los tipos básicos, todos los tipos definidos por el usuario y el tipo **`address`**.
`template`	Dinámica en el momento de ejecución	*Valores de*: todos los tipos básicos, todos los tipos definidos por el usuario, el tipo **`address`** y **`template`**.
`function`	Dinámica en el momento de ejecución	*Valores de*: todos los tipos básicos, todos los tipos definidos por el usuario, el tipo address, component, port, default, **`template`** y **`timer`**.
`altstep`	Dinámica en el momento de ejecución	*Valores de*: todos los tipos básicos, todos los tipos definidos por el usuario, el tipo **`address`**, **`component`**, **`port`**, default, **`template`** y **`timer`**.
`testcase`	Dinámica en el momento de ejecución	*Valores de*: todos los tipos básicos, todos los tipos definidos por el usuario, el tipo **`address`** y **`template`**.
`signature`	Dinámica en el momento de ejecución	*Valores de*: todos los tipos básicos, todos los tipos definidos por el usuario, el tipo address y **`component`**.
NOTA 1 – No es posible parametrizar las definiciones de **`record of`**, **`set of`**, **`enumerate`**, **`port`**, **`component`** y **`sub-type`**. NOTA 2 – En las cláusulas apropiadas de la Recomendación se incluyen ejemplos de la sintaxis y la utilización particular de la parametrización con distintos elementos del lenguaje.		

5.2.1 Transferencia de parámetros por referencia y por valor

5.2.1.0 Consideraciones generales

La opción por defecto es transferir por valor todos los parámetros efectivos de los tipos básicos, tipos de cadena básicos, tipos estructurados definidos por el usuario, tipo dirección y tipo componente. Puede indicarse (es facultativo) mediante la palabra clave **`in`**. Si estos parámetros se van a introducir por referencia habrá que utilizar las palabras clave **`out`** o **`inout`**.

Los temporizadores y los puertos siempre se transfieren por referencia. El parámetro de temporizador se identifica con la palabra clave **`timer`**. Los parámetros de puerto se identifican por su tipo de puerto. La introducción por referencia puede indicarse (es facultativo) mediante la palabra clave **`inout`**.

5.2.1.1 Parámetros transferidos por referencia

La introducción de parámetros por referencia tiene las siguientes limitaciones:

a) Sólo las listas de parámetros formales de altsteps, funciones, firmas y casos de prueba pueden contener parámetros introducidos por referencia.

NOTA – Hay otras restricciones para la utilización de parámetros introducidos por referencia en las firmas (véase la cláusula 23) y las alsteps (véanse 16.2.1 y 21.3.1).

b) Todos los parámetros efectivos han de ser valores o plantillas variables, puertos o temporizadores.

EJEMPLO:

```
function MyFunction(inout boolean MyReferenceParameter){ … };
// MyReferenceParameter se transfiere por referencia. El parámetro efectivo se puede leer
// y fijar desde el interior de la función
```

```
function MyFunction(out template boolean MyReferenceParameter){ … };
// MyReferenceParameter se transfiere por referencia. El parámetro efectivo sólo se puede
// fijar desde el interior de la función
```

5.2.1.2 Parámetros transferidos por valor

Los parámetros efectivos que son introducidos por valor pueden ser variables, constantes, plantillas, etc.

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

5.2.2 Listas de parámetros formales y efectivos

El número de elementos y el orden en el que aparecen en una lista de parámetros efectivos deben coincidir con el número de elementos y el orden en el que aparecen en la correspondiente lista de parámetros formales. Además, el tipo de cada parámetro efectivo ha de ser compatible con el tipo de cada parámetro formal correspondiente:

EJEMPLO:

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

// Solicitud de función con una lista de parámetros efectivos
MyFunction(123, true,'1100'B);
```

5.2.3 Lista de parámetros formales vacía

Si la lista de parámetros formales de los elementos de lenguaje TTCN-3 **`function`**, **`testcase`**, **`signature`**, **`altstep`** o **`external function`** 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:

```
// Definición de función con una lista de parámetros vacía
function MyFunction(){ … }

// Definición de registro con una lista de parámetros vacía
type record MyRecord { … }
```

5.2.4 Listas de parámetros jerarquizados

Todas las entidades parametrizadas especificadas como un parámetro efectivo tendrán sus propios parámetros resueltos en la lista de parámetros efectivos de nivel superior.

EJEMPLO:

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

// Ésta sería una posible plantilla de mensaje:
template MyMessageType MyTemplate(integer MyValue) :=
{
    field1 := MyValue,
    field2 := pattern "abc*xyz",
    field3 := true
}

// Éste sería un posible caso de prueba parametrizado con una plantilla:
testcase TC001(template MyMessageType RxMsg) runs on PTC1 system TS1
:
    MyPCO.receive(RxMsg);
}

// Si el caso de prueba se solicita en la parte de control y la plantilla parametrizada
// se utiliza como un parámetro efectivo, es necesario proporcionar los parámetros
// efectivos para la plantilla
control
{   :
    execute(TC001(MyTemplate(7)));
    :
}
```

5.2.5 Parámetros formales de tipo plantilla

5.2.5.1 Parametrización con plantillas y atributos concordantes

Para posibilitar que las plantillas o los símbolos concordantes se introduzcan como parámetros efectivos se debe añadir la palabra clave suplementaria **`template`** antes del campo de tipo del parámetro formal correspondiente. Esto hace que el parámetro se convierta en un tipo plantilla y en realidad amplía los parámetros permitidos para que el tipo asociado incluya el conjunto adecuado de atributos concordantes (véase el anexo B), 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 optional,
    field2 := pattern "abc*xyz",
    field3 := true
}

// puede utilizarse de la siguiente manera
pco1.receive(MyTemplate(?));
// O de la siguiente
pco1.receive(MyTemplate(omit));
```

5.2.5.2 Elementos de lenguaje que emplean parámetros de tipo plantilla

Únicamente las definiciones de **`function`**, **`testcase`**, **`altstep`** y **`template`** pueden tener parámetros formales de tipo plantilla.

EJEMPLO:

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

5.3 Reglas que determinan el ámbito de aplicación

5.3.0 Consideraciones generales

La notación TTCN-3 proporciona siete unidades de ámbito básicas:

a) parte de definiciones del módulo;

b) parte de control del módulo;

c) tipos component;

d) funciones;

e) altsteps (alternativas);

f) casos de prueba;

g) "bloques de instrucciones y declaraciones" en las declaraciones compuestas.

NOTA 1 – Hay otras reglas de alcance para los grupos (véase 7.3.1).

NOTA 2 – Hay otras reglas de alcance para los contadores y los bucles **`for`** (véase 19.7).

Cada unidad de ámbito consiste en declaraciones (facultativas). En varias unidades de ámbito: parte de control del módulo, funciones, casos de prueba, alternativas (altsteps) y "bloques de instrucciones y declaraciones" de declaraciones compuestas también se puede especificar una forma de comportamiento con declaraciones y operaciones de programa de la notación TTCN-3 (véase la cláusula 18).

Las definiciones que se hagan en la parte de definiciones del módulo, pero fuera de otras unidades de ámbito, son visibles globalmente: pueden utilizarse en cualquier lugar del módulo, incluidas todas las funciones, los casos de prueba y las alternativas (altsteps) definidos en el módulo y en la parte de control. Los identificadores importados de otros módulos también son visibles globalmente en todo el módulo importador.

Las definiciones que se hagan en la parte de control del módulo tienen una visibilidad local: sólo pueden utilizarse dentro de la parte de control.

Las definiciones que se hagan en un tipo de componente de prueba sólo pueden utilizarse en las funciones, los casos de prueba y las alternativas (altsteps) en los que se mencione este tipo de componente o un tipo de componente compatible (véase 16.3) mediante una cláusula **`runs on`**.

Los casos de prueba, las alternativas (altsteps) y las funciones son unidades de ámbito particulares que no tienen ninguna relación jerárquica entre ellas: las declaraciones que se hagan al principio del texto de programación son de visibilidad local y sólo han de utilizarse en el correspondiente caso de prueba, alternativa (altstep) o función (por ejemplo, una declaración hecha en un caso de prueba no es visible en una función solicitada por el caso de prueba ni en una alternativa (altstep) utilizada por el caso de prueba).

Las declaraciones compuestas, por ejemplo **`if-else`**-, **`while`**-, **`do-while`**- y las declaraciones alternativas (**`alt`**-statements) contienen "bloques de instrucciones y declaraciones" que pueden utilizarse dentro de la parte de control de un módulo, en casos de prueba, alternativas (altsteps) o funciones, y que pueden incluirse en otras declaraciones compuestas, por ejemplo una declaración **`if-else`** utilizada dentro de un bucle **`while`**.

Los "bloques de instrucciones y declaraciones" de instrucciones compuestas y las instrucciones compuestas incluidas tienen una relación jerárquica con la unidad de ámbito que incluye ese "bloque de instrucciones y declaraciones" y con todos los "bloques de instrucciones y declaraciones" incluidos. Las declaraciones que se hagan dentro de un "bloque de instrucciones y declaraciones" tienen visibilidad local.

En la figura 2 se ha representado la jerarquía de las unidades de ámbito. Las declaraciones de una unidad de ámbito son visibles para todas las unidades de niveles inferiores dentro del mismo ramal de jerarquía. Las declaraciones de una unidad de ámbito inferior no son visibles para las unidades de niveles de jerarquía superiores.

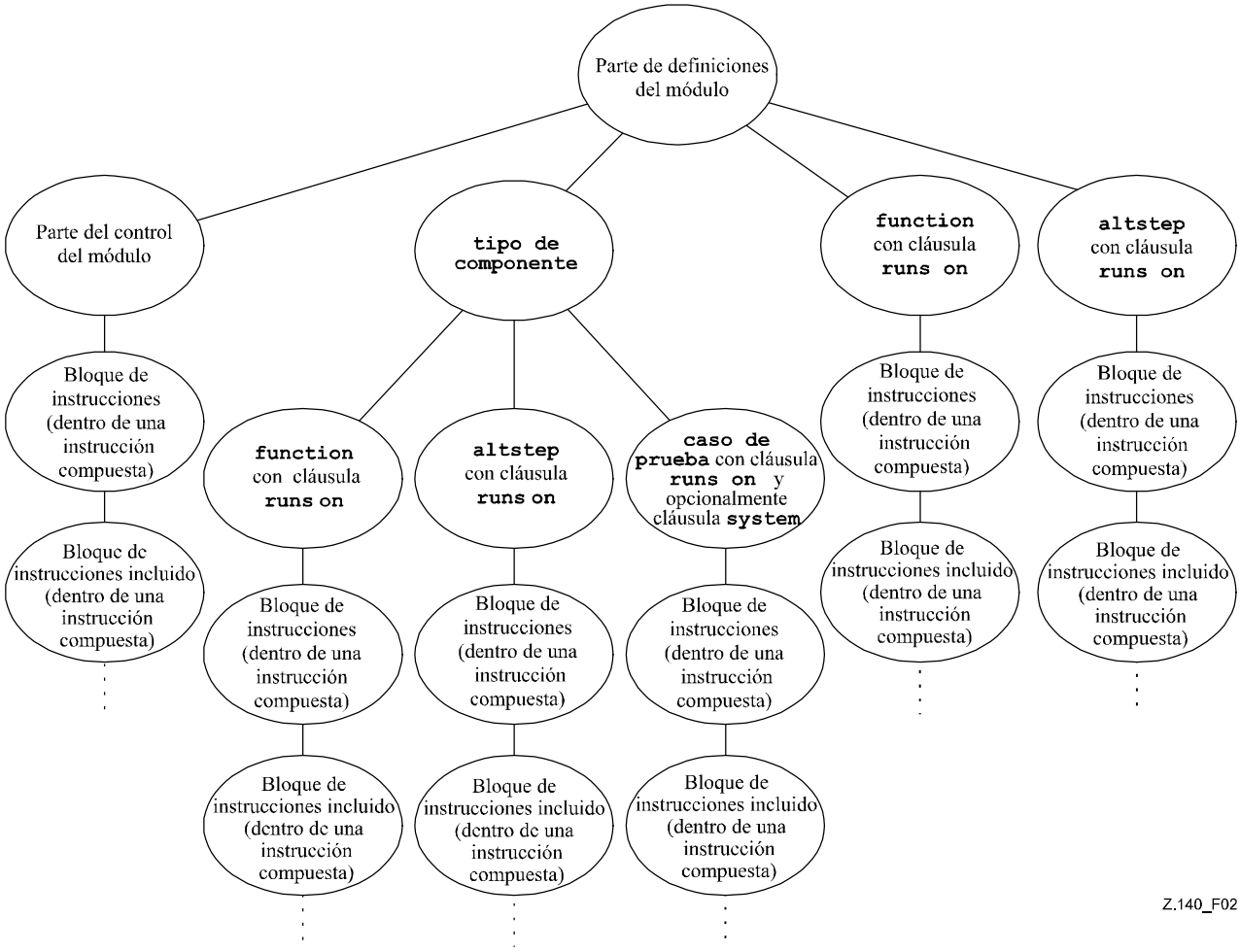


Figura 2/Z.140 – Jerarquía de unidades de ámbito

EJEMPLO:

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

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

    }
}
```

5.3.1 Ámbito de los parámetros formales

El ámbito de los parámetros formales de un elemento de lenguaje parametrizado (por ejemplo una solicitud de función) ha de limitarse a la definición en la que aparecen los parámetros, y a los niveles inferiores de ámbito de la misma jerarquía. Dicho de otra forma, se aplican las reglas de ámbito normales (véase 5.3.0).

5.3.2 Carácter unívoco de los identificadores

Los identificadores de la notación TTCN-3 han de ser únicos: todos los identificadores de la misma jerarquía de ámbito serán diferentes. Esto significa que una instrucción en un nivel de ámbito más bajo no podrá reutilizar el mismo identificador que una instrucción situada en un nivel más alto y en el mismo ramal de la jerarquía de ámbito. No es necesario que los identificadores para campos de tipos estructurados, valores de enumeración y grupos tengan un carácter único globalmente; ahora bien, en el caso de valores de enumeración, los identificadores sólo podrán reutilizarse para valores de enumeración en el contexto de otros tipos enumerados. Las reglas aplicables del carácter único de los identificadores valen igualmente para los identificadores de parámetros formales.

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 Identificadores y palabras clave

Los identificadores de la notación TTCN-3 son sensibles a mayúsculas y minúsculas. Las palabras clave de la TTCN-3 deberán escribirse enteramente con letras minúsculas (véase el anexo A). Las palabras clave de la TTCN-3 no podrán utilizarse como identificadores de objetos TTCN-3 o de objetos importados de módulos de otros lenguajes.

6 Tipos y valores

6.0 Consideraciones generales

La notación TTCN-3 soporta varios tipos básico previamente definidos: los tipos básicos asociados normalmente a un lenguaje de programación, tales como **integer**, **boolean** y tipos de cadena, y algunos tipos específicos de TTCN-3, tales como **verdicttype**. A partir de estos tipos básicos es posible construir tipos estructurados (por ejemplo los tipos **record**, **set** y **enumerated**).

El tipo de datos especial **anytype** representa la unión de todos los tipos de datos conocidos y el tipo dirección en un módulo.

Hay tipos especiales asociados a configuraciones de prueba, por ejemplo **address**, **port** y **component** que pueden utilizarse para definir la arquitectura del sistema de prueba (véase la cláusula 22).

El tipo de datos especial **default** puede utilizarse para hacer un tratamiento por defecto (véase la cláusula 21).

En el cuadro 3 se resumen los tipos de la notación TTCN-3.

Cuadro 3/Z.140 – Los tipos TTCN-3

Clase de tipo	Palabra clave	Subtipo
Tipos básicos simples	**integer**	gama, lista
	float	gama, lista
	boolean	lista
	objid	lista
	verdicttype	lista
Tipos de cadena básicos	**bitstring**	lista, longitud
	hexstring	lista, longitud
	octetstring	lista, longitud
	charstring	gama, lista, longitud, patrón
	universal charstring	gama, lista, longitud, patrón
Tipos estructurados	**record**	lista (véase la nota)
	record of	lista (véase la nota), longitud
	set	lista (véase la nota)
	set of	lista (véase la nota), longitud
	enumerated	lista (véase la nota)
	union	lista (véase la nota)
Tipos de datos especiales	**anytype**	lista (véase la nota)
Tipos de configuración especiales	**address**	
	port	
	component	
Tipo especial opción por defecto	**default**	
NOTA – La definición de subtipos de la lista de estos tipos es posible cuando se define un nuevo tipo condicionado a partir de un tipo progenitor existente, pero no directamente en la declaración del primer tipo progenitor.		

6.1 Tipos básicos y valores

6.1.0 Tipos básicos simples y valores

La notación TTCN-3 soporta los siguientes tipos básicos:

a) **integer**: un tipo cuyos valores diferenciados son los números enteros positivos y negativos, incluido cero.

   Los valores del tipo integer se indicarán con una o más cifras, y la primera cifra no será cero, a menos que el valor sea 0; el valor cero se representará con un solo cero.

b) **float**: el tipo de los números con coma flotante.

Por lo general, los números con coma flotante pueden definirse 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.

En TTCN-3, la notación de valor de números con coma flotante sólo puede utilizarse para una numeración base 10. Hay dos formas de expresar notaciones de valores de coma flotante:

– utilizando la notación decimal con una coma en una secuencia de números, por ejemplo 1,23 (que representa 123×10^{-2}), 2,783 (que representa 2783×10^{-3}) o –123,456789 (que representa $-123\ 456\ 789 \times 10^{-6}$); o

– utilizando dos números separados por E, donde el primer número especifica la mantisa y el segundo especifica el exponente, por ejemplo 12,3E4 (que representa 123×10^{3}) o –12,3E-4 (que representa -123×10^{-5}).

NOTA – A diferencia de la definición general de valores con coma flotante, la mantisa de la notación de valor de TTCN-3, al lado de los enteros, también permite números decimales.

c) **boolean**: un tipo que consiste en dos valores diferenciados.

Los valores del tipo booleano se indicarán mediante **true** (verdadero) y **false** (falso).

d) Vacío.

e) **verdicttype**: un tipo que se ha de utilizar para los veredictos de prueba y que consiste en 5 valores diferenciados. Los valores de **verdicttype** se indicarán mediante **pass**, **fail**, **inconc**, **none** y **error**.

6.1.1 Tipos de cadena básicos y valores

La notación TTCN-3 soporta los siguientes tipos de cadena básicos:

NOTA 1 – En la notación TTCN-3 el término general cadena o tipo de cadena se refiere a **bitstring**, **hexstring**, **octetstring**, **charstring** y **universal charstring**.

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

Los valores del tipo **bitstring** se indicarán mediante un número arbitrario (posiblemente cero) de 0 y 1, precedidos por la comilla simple (') y seguidos por el par de caracteres 'B.

EJEMPLO 1: `'01101'B`.

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

Los valores de tipo **hexstring** se indicarán mediante un número arbitrario (posiblemente cero) de los dígitos hexadecimales (las letras mayúsculas y minúsculas también pueden utilizarse como dígitos hexadecimales):

0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

precedidos por la comilla simple (') y seguidos por el par de caracteres 'H; cada dígito hexadecimal se utiliza para indicar el valor de un semiocteto utilizando una representación hexadecimal.

EJEMPLO 2:
```
'AB01D'H
'ab01d'H
'Ab01D'H
```

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

Los valores del tipo **octetstring** se indicarán mediante un número arbitrario, pero siempre un número par (posiblemente cero) de los dígitos hexadecimales (las letras mayúsculas y minúsculas también pueden utilizarse como dígitos hexadecimales):

0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

precedidos por la comilla simple (') y seguidos por el par de caracteres 'o; cada dígito hexadecimal se utiliza para indicar el valor de un semiocteto con una representación hexadecimal.

EJEMPLO 3:
```
'FF96'O
'ff96'O
'Ff96'O
```

d) **charstring**: son tipos cuyos valores diferenciados son caracteres de la Rec. UIT-T T.50 [9] ninguno, uno o más que satisfacen la versión internacional de referencia (IRV, *international reference version*) conforme a 8.2/T.50 [9].

NOTA 2 – La versión IRV del Alfabeto Internacional de referencia (anteriormente Alfabeto Internacional N.° 5 o IA5) que se describe en la Rec. UIT-T T.50 [9] es equivalente a la versión IRV de ISO/CEI 646.

Los valores del tipo **charstring** (cadena de caracteres) se indicarán mediante un número arbitrario (posiblemente cero) de caracteres del juego de caracteres pertinente, precedidos y seguidos por comillas (") o se calcularán aplicando la función de conversión predefinida int2char con el valor entero positivo de su codificación como argumento (véase la cláusula C.1).

NOTA 3 – La función de conversión predefinida puede devolver sólo valores de longitud de carácter simple.

Si es necesario definir cadenas que incluyen el carácter comillas ("), se escribirá un par de comillas en la misma línea sin caracteres intermedios.

EJEMPLO 4: `""abcd""` representa la cadena literal `"abcd"`.

e) El tipo cadena de caracteres precedido por la palabra clave **universal** indica tipos cuyos valores distinguidos son cero, uno o varios caracteres de ISO/CEI 10646 [10].

Los valores **universal charstring** también podrán indicarse mediante un número arbitrario (posiblemente cero) de caracteres del juego de caracteres pertinente, precedidos y seguidos por comillas ("), que se calculan empleando una función de conversión predefinida (véase C.3) con el valor entero positivo de su codificación como argumento o mediante un "cuádruplo".

NOTA 4 – La función de conversión predefinida puede devolver sólo valores de longitud de carácter simple.

Si es necesario definir cadenas que incluyen el carácter comillas ("), se escribirá un par de comillas en la misma línea sin caracteres intermedios.

Este "cuádruplo" sólo puede indicar un carácter, y lo hace mediante los valores decimales de su grupo, plano, fila y célula, conforme a la especificación ISO/CEI 10646 [10], precedido por la palabra clave **char** entre corchetes y separados por comas (por ejemplo, **char** (0, 0, 1, 113) indica al carácter húngaro "ű"). Si es necesario indicar el carácter de comillas (") en una cadena asignada por el primer método (entre comillas), se escribirá un par de comillas en la misma línea sin caracteres intermedios. Los dos métodos pueden combinarse en una sola notación para el valor de una cadena, utilizando el operador de concatenación.

EJEMPLO 5: La expresión "El carácter Braille " & **char** (0, 0, 40, 48) & "es así" representa la siguiente cadena literal: El carácter Braille ⣿ es así.

NOTA 5 – Los caracteres de control pueden indicarse mediante la función de conversión predefinida o los cuádruplos.

Por defecto **universal charstring** deberá ajustarse a la forma de representación codificada UCS-4 especificada en 14.2 de ISO/CEI 10646 [10].

NOTA 6 – UCS-4 es un formato de codificación que representa todos los caracteres UCS mediante un campo fijo de 32 bits.

Los atributos variant definidos pueden reemplazar esta codificación por defecto (véase 28.2.3). En el anexo E se definen los siguientes tipos útiles de cadena de caracteres que utilizan estos atributos: utf8string, bmpstring, utf16string e iso8859string.

6.1.2 Acceso a los elementos de una cadena

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

En el cuadro 4 se indican las unidades de longitud de los distintos elementos del tipo cadena.

Los índices comenzarán con el valor cero (0).

EJEMPLO:

```
// Dado
MyBitString := '11110111'B;
// la expresión
MyBitString[4] := '1'B;
// produce la cadena de bits '11111111'B
```

6.2 Definición de subtipos de los tipos básicos

6.2.0 Consideraciones generales

Para indicar los tipos definidos por el usuario se utilizará la palabra clave **type**. El usuario puede definir subtipos (tales como listas, gamas y restricciones de longitud) a partir de los tipos básicos, los tipos estructurados y **anytype** descritos en el cuadro 3.

6.2.1 Listas de valores

La notación TTCN-3 permite especificar una lista de valores diferenciados de tipos básicos, tipos estructurados y **anytype** del cuadro 3. Los valores de la lista han de ser del tipo raíz y han de constituir realmente un subconjunto de los valores definidos por el tipo raíz. Los valores de la lista serán los únicos valores permitidos para el subtipo definido por esta lista.

EJEMPLO:

```
type bitstring MyListOfBitStrings ('01'B, '10'B, '11'B);
type float pi (3.1415926);
type charstring MyStringList ("abcd", "rgy", "xyz");
type universal charstring SpecialLetters (char(0, 0, 1, 111),
char(0, 0, 1, 112), char(0, 0, 1,      113));
```

6.2.2 Gamas

6.2.2.0 Consideraciones generales

La notación TTCN-3 permite especificar condicionamientos de gama de los tipos **integer**, **charstring**, **universal charstring** y **float** (o derivaciones de estos tipos). En el caso de **integer** y **float** los valores de la gama, incluidos los límites inferior y superior, determinan un subtipo y son los únicos valores permitidos. En el caso de los tipos **charstring** y **universal charstring**, la gama restringe los valores permitidos para cada uno de los caracteres de las cadenas. Es condición que los límites se traduzcan en posiciones de carácter válidas conforme a la tabla o las tablas de caracteres codificados del tipo (por ejemplo, la posición considerada no puede estar vacía). Se considera que una posición vacía entre los límites inferior y superior no es un valor válido de la gama especificada.

EJEMPLO 1:

```
type integer MyIntegerRange (0 .. 255);
type float piRange (3.14 .. 3142E-3);
```

EJEMPLO 2:

```
type charstring MyCharString ("a" .. "z");
// Define un tipo de cadena de longitud indeterminada, de caracteres
// dentro de la gama especificada
type universal charstring MyUCharString1 ("a" .. "z");
// Define un tipo de cadena de longitud indeterminada, de caracteres
// dentro de la gama de a a z
// (códigos de carácter de 97 a 122), como "abxyz";
// se desaconsejan las cadenas que contengan cualquier otro carácter
// (incluidos los caracteres de control), como "abc2".
type universal charstring MyUCharString2 (char(0, 0, 1, 111) .. char(0, 0, 1, 113));
// Define un tipo de cadena de longitud indeterminada, de caracteres
// dentro de la gama especificada mediante la notación de cuádruplos
```

6.2.2.1 Gamas infinitas

Para especificar una gama infinita de enteros o valores de coma flotante se puede utilizar la palabra clave **infinity** en vez de un valor: indica que no hay un límite inferior o superior. El límite superior ha de ser igual al límite inferior o mayor.

EJEMPLO:

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

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

6.2.2.2 Mezclar listas y gamas

En blanco.

NOTA – La cláusula 6.2.5 sustituye esta cláusula.

6.2.3 Restricciones de longitud de cadena

La notación TTCN-3 permite especificar restricciones de longitud en tipos de cadenas. Los límites de longitud se basan en unidades diferentes dependiendo del tipo de cadena con el que se utilizan, pero siempre deberán traducirse en valores enteros no negativos (o valores **integer** derivados).

EJEMPLO:

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

En el cuadro 4 se especifican las unidades de longitud para diferentes tipos de cadena.

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
bitstring	caracteres

Para el límite superior también se puede utilizar la palabra clave **infinity**, para indicar que no hay ningún límite superior de longitud. El límite superior ha de ser igual al límite inferior o mayor.

6.2.4 Definición de subtipos de patrones de tipos de cadenas de caracteres

La notación TTCN-3 posibilita la aplicación de los patrones de caracteres que se especifican en B.1.5 a fin de condicionar los valores permitidos de los tipos **charstring** y **universal charstring**. La restricción de tipo debe utilizar la palabra de código **pattern** seguida por un patrón de caracteres. Los valores señalados por el patrón deben ser un subconjunto verdadero del tipo que se divide en subtipos.

NOTA – La definición de subtipos puede considerarse una forma especial de restricción de lista, cuyos miembros no se han definido mediante cadenas de caracteres específicas de la lista, sino a través de un mecanismo que genera elementos de la lista.

EJEMPLO:

```
type charstring MyString (pattern "abc*xyz");
// los valores permitidos de MyString tienen el prefijo abc y el sufijo xyz

type universal charstring MyUString (pattern "*\r\n")
// los valores permitidos de MyUString terminan en CR/LF

type charstring MyString2 (pattern "abc?\q{0,0,1,113}");
// introduce un error debido a que el carácter indicado por el cuádruplo
// {0,0,1,113} no es un carácter legítimo del tipo charstring de TTCN-3

type MyString MyString3 (pattern "d*xyz");
// introduce un error debido a que el tipo MyString no incluye un valor
// que comience con el carácter d
```

6.2.5 Combinación de mecanismos de definición de subtipos

6.2.5.1 Combinación de patrones, listas y gamas

En los tipos **integer** y **float** (o sus derivaciones) se permite la definición de subtipos para combinar listas y gamas. La superposición de diferentes restricciones no constituye un error.

EJEMPLO 1:

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

En las definiciones de subtipos de **charstring** y **universal charstring** se prohíbe la combinación de restricciones de patrones, listas o gamas.

EJEMPLO 2:

```
type charstring MyCharStr0 ('gr', 'xyz');
// contiene las cadenas de caracteres gr y xyz;

type charstring MyCharStr1 ('a'..'z');
// contiene cadenas de caracteres de longitud arbitraria que incluyen los caracteres a a z.

type charstring MyCharStr2 (pattern '[a-z]#(3,9)');
// contiene cadenas con longitud entre 3 y 9 caracteres que incluyen los caracteres a a z
```

6.2.5.2 Utilización de la restricción de longitud con otras restricciones

En las definiciones de subtipos de **bitstring**, **hexstring**, **octetstring** las restricciones de listas y longitud pueden combinarse en la misma definición de subtipo.

En las definiciones de subtipos de **charstring** y **universal charstring** se permite añadir una restricción de longitud a las restricciones que contienen definición de subtipos de lista, gama o patrón en la misma definición de subtipos.

Cuando la restricción de longitud se combina con otras restricciones, ésta debe ser el último elemento de la definición de subtipos. La restricción de longitud se aplica junto con otros mecanismos de definición de subtipos (es decir, el conjunto de valores del tipo consiste en el subconjunto común de los conjuntos de valores identificados por la definición de subtipos de lista, gama o patrón y la restricción de longitud).

EJEMPLO:

```
type charstring MyCharStr5 ('gr', 'xyz') length (1..9);
// contiene las cadenas de caracteres gr y xyz;

type charstring MyCharStr6 ( 'a'..'z') length (3..9);
// contiene cadenas con longitud entre 3 y 9 caracteres que incluyen los caracteres a a z

type charstring MyCharStr7 (pattern '[a-z]#(3,9)') length (1..9);
// contiene cadenas con longitud entre 3 y 9 caracteres que incluyen los caracteres a a z

type charstring MyCharStr8 (pattern '[a-z]#(3,9)') length (1..8);
// contiene cadenas con longitud entre 3 y 8 caracteres que incluyen los caracteres a a z

type charstring MyCharStr9 (pattern '[a-z]#(1,8)') length (1..9);
// contiene cadenas de cualesquiera caracteres con longitud entre 1 y 8 caracteres
// que incluyen los caracteres a a z

type charstring MyCharStr10 ('gr', 'xyz') length (4);
// no contiene valor (tipo vacío).
```

6.3 Tipos estructurados y valores

6.3.0 Consideraciones generales

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

Para expresar los valores de estos tipos se puede utilizar una notación de asignación explícita o una notación simplificada de lista de valores.

EJEMPLO 1:

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

// O
const MyRecordType MyRecordValue:= {'11001'B, true, "A string"}     // notación de lista
                                                                    // de valores
```

Si se utiliza la notación de asignación para especificar valores parciales (sólo se determina el valor de un subconjunto de los campos de una variable estructurada), sólo es necesario especificar los campos a los que hay que asignar valores. Los campos no mencionados quedan implícitamente no especificados. Asimismo, pueden dejarse campos explícitamente no especificados mediante la aplicación del símbolo "no se utiliza" "-". Si se utiliza la notación de lista

de valores se han de especificar todos los campos de la estructura, sea con un valor, con el símbolo "no se utiliza" "-" o con la palabra clave **omit**.

EJEMPLO 2:

```
var MyRecordType MyVariable:=                                   //notación de asignación
{
    field1 := '11001'B,
    //field2 implícitamente no especificado
    field3 := "A string"
}
//  O
var MyRecordType MyVariable:=                                   //notación de asignación
{
    field1 :='11001'B,
    field2 :=-,//field2 explícitamente no especificado
    field3 :="A string"
}
//O
var MyRecordType MyVariable:= {'11001'B, -, "A string"}     //notación de lista de valores
```

No se permite mezclar las dos notaciones de valores en el mismo contexto (inmediato).

EJEMPLO 3:

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

Tanto en la notación de asignación como en la notación de lista de valores hay que utilizar el valor explícito **omit** cuando se van a omitir campos facultativos. La palabra clave **omit** no deberá utilizarse para los campos obligatorios. Cuando se vuelve a asignar un valor inicializado anteriormente, el empleo del símbolo "no se utiliza" o el salto de un campo de la notación de asignación provocará que los campos pertinentes permanezcan sin cambios.

EJEMPLO 4:

```
var MyRecordType MyVariable :=
{
    field1 := '111'B,
    field2 := false,
    field3 := -
}

MyVariable := { '10111'B, -, - };
// tras lo anterior, MyVariable contiene { '10111'B, false /* unchanged */, // <undefined> }

MyVariable :=
{
    field2 := true
}
// tras lo anterior, MyVariable contiene { '10111'B, true, <undefined> }

MyVariable :=
{
    field1 := -,
    field2 := false,
    field3 := -
}
// tras lo anterior, MyVariable contiene { '10111'B, false, <undefined> }
```

6.3.1 El tipo record (registro) y sus valores

6.3.1.0 Consideraciones generales

La notación TTCN-3 soporta los tipos estructurados ordenados **record**. Los elementos de un tipo **record** pueden ser cualquiera de los tipos básicos o de los tipos de datos definidos por el usuario (otros registros, conjuntos o matrices). Los valores del tipo **record** han de ser compatibles con los tipos de los campos **record**. Los identificadores de elementos son de carácter local para el registro y han de ser únicos dentro de éste (no es preciso que sean globalmente únicos). Una constante de tipo **record** no podrá contener, directa o indirectamente, variables ni parámetros de módulo como valores de campo.

EJEMPLO 1:

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

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

Los registros pueden definirse sin campos (es decir, como registro vacíos).

EJEMPLO 2:

```
type record MyEmptyRecord { }
```

Los valores de **record** se asignan elemento por elemento. El orden de los valores de los campos en la notación de lista de valores será idéntico al orden de los campos en la definición de tipo relacionada.

EJEMPLO 3:

```
var integer MyIntegerValue:= 1;

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

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

El mismo valor especificado en una lista de valores.

EJEMPLO 4:

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

6.3.1.1 Referencia a los campos de un tipo record

Para hacer una referencia a los elementos de un tipo **record** deberá utilizarse la notación de puntos *TypeOrValueId.ElementId*, donde *TypeOrValueId* se traduce en el nombre de un tipo o una variable estructurados. *ElementId* se traduce en el nombre de un campo en un tipo estructurado.

EJEMPLO:

```
MyVar1 := MyRecord1.myElement1;
// En el caso de un registro jerarquizado dentro de otro tipo,
// la referencia tendría las siguientes características:
MyVar2 := MyRecord1.myElement1.myElement2;
```

6.3.1.2 Elementos facultativos en un registro

Los elementos facultativos en un **record** se especificarán utilizando la palabra **optional**.

EJEMPLO 1:

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

Los campos facultativos se omitirán utilizando el símbolo **omit**.

EJEMPLO 2:

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

// Obsérvese que es diferente de la siguiente instrucción
// MyRecordValue:= {MyIntegerValue, -, "A string"};
// que significa que no se modifica el valor de field2
```

6.3.1.3 Definiciones de tipos de subniveles de jerarquía para los tipos de campos

La notación TTCN-3 soporta la definición de tipos para los campos **record** en subniveles de jerarquía dentro de la definición de **record**. Existe la posibilidad tanto de la definición de nuevos tipos estructurados (**record**, **set**, **enumerated**, **set of** y **record of**) como de la especificación de restricciones de subtipos.

EJEMPLO:

```
// tipo de registro con definiciones de tipo estructurado en subniveles de jerarquía
type record MyNestedRecordType
{
    record
    {
          integer nestedField1,
          float nestedField2
} outerField1,
enumerated {
          nestedEnum1,
          nestedEnum2
    } outerField2,
    record of boolean outerField3
}

// tipo de registro con definiciones de subtipos en subniveles de jerarquía
type record MyRecordTypeWithSubtypedFields
{
    integer field1 (1 .. 100),
    charstring field2 length ( 2 .. 255 )
}
```

6.3.2 El tipo set (conjunto) y sus valores

6.3.2.0 Consideraciones generales

La notación TTCN-3 soporta los tipos estructurados no ordenados **set**. Estos tipos y los valores son similares a los registros, pero la ordenación de los campos **set** no es significativa.

EJEMPLO:

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

Los identificadores de campos son de carácter local para el conjunto y han de ser únicos dentro de éste (no es preciso que sean globalmente únicos).

No deberá utilizarse la notación de lista de valores para los tipos **set**.

6.3.2.1 Referencia a los campos de un tipo set

Para hacer una referencia a los elementos de un tipo **set** deberá utilizarse la notación de puntos (véase 6.3.1.1).

EJEMPLO:

```
MyVar3 := MySet1.myElement1;
// Referencia en el caso de un conjunto dentro de otro tipo
MyVar4 := MyRecord1.myElement1.myElement2;
// Obsérvese que el tipo set a cuyo campo se hace referencia
// con el identificador 'myElement2' está incluido en un tipo record
```

6.3.2.2 Elementos facultativos en un conjunto

Los elementos facultativos en un **set** se especificarán utilizando la palabra clave **optiona**l.

6.3.2.3 Definición de tipos de subniveles de jerarquía para los tipos de campos

La notación TTCN-3 soporta la definición de tipos para los campos **set** en subniveles de jerarquía dentro de la definición de **set**, de manera similar a los tipos record que se describen en 6.3.1.3.

6.3.3 Registros y conjuntos de un determinado tipo

6.3.3.0 Consideraciones generales

La notación TTCN-3 soporta la especificación de registros y conjuntos cuyos elementos son todos del mismo tipo. Para indicarlos se utiliza la palabra clave **of**. Estos registros y conjuntos no tienen identificadores de elementos y se pueden comparar 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**.

EJEMPLO 1:

```
type record length(10) of integer MyRecordOfType;
// es un registro que tiene exactamente 10 enteros

type record length(0..10) of integer MyRecordOfType;
// es un registro que tiene como máximo 10 enteros

type record length(10..infinity) of integer MyRecordOfType;
// es un registro que tiene como mínimo 10 enteros

type set of boolean MySetOfType;
// es un conjunto ilimitado de valores booleanos

type record length(0..10) of charstring StringArray length(12);
// es un registro que tiene como máximo 10 cadenas, cada una
// de 12 caracteres de longitud exactamente
```

Para los valores de **record of** y **set of** se utilizará una notación de lista de valores o una notación indexada para un determinado elemento (es la misma notación de valores utilizada para matrices; véase 6.5). Hay una excepción a esta regla general: en caso de definir plantillas modificadas, cuando se permite utilizar también la notación de asignación (véase 14.6.0).

Si se utiliza la notación de lista de valores, el primer valor de la lista se asigna al primer elemento, el segundo al segundo elemento y así sucesivamente. No se permiten asignaciones vacías (por ejemplo dos comas inmediatas o separadas por un espacio en blanco). Para no incluir un elemento en la asignación habrá que omitirlo o evitarlo explícitamente en la lista.

Se podrán utilizar notaciones de valor indexadas en los dos lados de las asignaciones. El índice del primer elemento ha de ser cero y se observará el valor límite de índice determinado por el subtipo de longitud. Se producirá un error de semántica o de ejecución si el valor de un elemento señalado por el índice a la derecha de una asignación no está definido. Si un operador de índice a la izquierda de una asignación señala un elemento que no existe, el valor de la derecha será asignado al elemento, y todos los elementos cuyo índice sea inferior al índice considerado y que no tengan ningún valor asignado serán creados con un valor no definido. Los elementos no definidos sólo están permitidos en estados temporales (mientras el valor permanezca invisible). El envío de un valor de **record** of con elementos no definidos da origen a un error de caso de prueba dinámico.

EJEMPLO 2:

```
// Dado
type record of integer MyRecordOf;
var integer MyVar;
var MyRecordOf MyRecordVar := { 0, 1, 2, 3, 4 };

MyVar := MyRecordVar[0];   // el primer elemento del valor de "record of" (entero 0)
                           // se asigna a MyVar

// También pueden darse valores indexados a la izquierda de la asignación:
MyRecordVar[1] := MyVar;   // MyVar se asigna al segundo elemento el valor
                           // de MyRecordVar es { 0, 0, 2, 3, 4 }

//La asignación
MyRecordVar := { 0, 1, -, 2, omit };
// cambiará el valor de MyRecordVar a{ 0, 1, 2 <unchanged>, 2};
// Obsérvese que el 3er elemento no estaría definido si no hubiera tenido
// un valor asignado anteriormente.
```

```
// La asignación
MyRecordVar[6] := 6;

// cambiaría el valor de MyRecordVar a { 0, 1, 2, 2, <undefined>, // <undefined>, 6 };
// Obsérvese que los elementos 5° y 6° (con índices 4 y 5) no tenían un valor asignado
// antes de esta última asignación y por consiguiente no están definidos.
```

NOTA – Siendo así es posible copiar valores de **record of** para cada elemento separadamente en un bucle "for". Por ejemplo, la siguiente función invierte los elementos de un valor de **record of**:

```
    function reverse(in MyRecord src) return MyRecord
          {
          var MyRecord dest;
          var integer I;
          for(I := 0; I < sizeof(src); I:= I + 1) {
          dest[sizeof(src) - 1 - I] := src[I];
          }
    return dest;
}
```

Los tipos **record of** y **set of** incrustados producirán una estructura de datos de las mismas características de matrices multidimensionales (véase 6.5).

EJEMPLO 3:

```
// Dado
type record of integer                    MyBasicRecordOfType;
type record of MyBasicRecordOfType        MyRecordOfType;

// la variable myRecordOfArray tendría atributos como una matriz bidireccional
var MyRecordOfType myRecordOfArray;
// y la referencia a un determinado elemento se haría así (valor del segundo elemento
// de la tercera construcción MyBasicRecordOfType')
// myRecordOfArray [2][1] := 1;
```

6.3.3.1 Definiciones de tipos con subniveles de jerarquía

La notación TTCN-3 soporta la definición del tipo agregado con subniveles de jerarquía en la definición de **record of** o **set of**. Tanto la definición de nuevos tipos estructurados (**record**, **set**, **enumerated**, **set of** y **record of**) como la especificación de restricciones de subtipo son posibles.

EJEMPLO:

```
type record of enumerated { red, green, blue } ColorList;
type record length (10) of record length (10) of integer Matrix;
type set of record { charstring id, charstring val } GenericParameters;
```

6.3.4 El tipo enumerated (enumerado) y sus valores

La notación TTCN-3 soporta los tipos **enumerated** que se utilizan para modelizar tipos que sólo tienen un conjunto de valores determinado y denominado. Estos valores determinados se conocen como enumeraciones. Cada enumeración ha de tener un identificador. Las operaciones que se realicen con tipos enumerados utilizarán exclusivamente estos identificadores y quedan restringidas por los operadores de asignación, equivalencia y ordenación. Los identificadores de enumeración serán únicos dentro del tipo enumerado (no es preciso que sean globalmente únicos) y, por tanto, sólo serán visibles dentro del contexto de ese tipo. Los identificadores de enumeración no podrán reutilizarse sino dentro de otras definiciones de tipos estructurados, y no podrán utilizarse para identificadores que tengan visibilidad local o global en el mismo nivel o un nivel inferior del mismo ramal de la jerarquía de ámbito (véase la jerarquía de ámbito en 5.3.0).

EJEMPLO 1:

```
type enumerated MyFirstEnumType {
  Monday, Tuesday, Wednesday, Thursday, Friday
};

type integer Monday;
// Esta definición no es válida porque el nombre del tipo tiene visibilidad local o global

type enumerated MySecondEnumType {
  Saturday, Sunday, Monday
};
// Esta definición es válida porque reutiliza el identificador de enumeración Monday dentro
// de otro tipo enumerado
```

```
type record MyRecordType {
  integer      Monday
};
// Esta definición es válida porque reutiliza el identificador de enumeración Monday
// dentro de un tipo estructurado particular como identificador de un determinado
// campo de ese tipo.

type record MyNewRecordType {
    MyFirstEnumType   firstField,
    integer           secondField
};

var MyNewRecordType newRecordValue := { Monday, 0 }
// MyFirstEnumType queda señalado de forma implícita por el elemento
// firstField de MyNewRecordType

const integer Monday := 7
// Esta definición no es válida porque reutiliza el identificador de enumeración Monday
// para un objeto diferente de la TTCN-3 dentro de la misma unidad de ámbito.
```

Puede asignarse a cada enumeración (es facultativo) un valor entero definido después del nombre de la enumeración entre paréntesis. Cada número entero asignado ha de ser diferente dentro de un determinado tipo **enumerated**. En el caso de enumeraciones sin valores enteros asignados, el sistema asocia sucesivamente a cada una un número entero en el orden textual de las enumeraciones empezando a la izquierda, partiendo de cero y con incrementos de 1, saltando los números que estén ocupados en una de las enumeraciones con un valor asignado manualmente. El sistema sólo utiliza estos valores para permitir la utilización de operadores relacionales.

NOTA 1 – El sistema también puede utilizar el valor entero para codificar/decodificar valores enumerados, pero esta utilización no entra en el marco de esta Recomendación (únicamente la asociación de atributos de codificación a elementos de la notación TTCN-3).

Es necesario crear una referencia implícita o explícita a un tipo **enumerated** para permitir la definición de ejemplares o la referencia a valores de este tipo.

NOTA 2 – Si el tipo enumerado es un elemento de un tipo estructurado definido por el usuario, hay una referencia implícita al tipo enumerado a través de ese elemento (mediante el identificador del elemento o la posición del valor en una notación de lista de valores) cuando se asigna el valor, se define una ejemplificación, etc.

EJEMPLO 2:

```
// Expresiones válidas de ejemplificaciones de MyFirstEnumType y MySecondEnumType
var MyFirstEnumType Today := Tuesday;
var MySecondEnumType Tomorrow := Monday;

// pero la siguiente instrucción no es válida porque los dos tipos de enumeración
// no son compatibles
Today := Tomorrow
```

6.3.5 Uniones

6.3.5.0 Consideraciones generales

La notación TTCN-3 soporta el tipo **union** que consiste en una serie de campos, cada uno con su identificador. En un valor efectivo del tipo **union** sólo aparecerá uno de los campos especificados. Los tipos union son útiles para modelizar una estructura que puede adoptar uno de los tipos conocidos (hay un número finito de tipos conocidos).

EJEMPLO:

```
type union MyUnionType
{
    integer           number,
    charstring        string
};

// La siguiente ejemplificación de MyUnionType sería válida
var MyUnionType age, oneYearOlder;
var integer ageInMonths;

age.number := 34;           // Notación de valor mediante una referencia al
                            // campo; obsérvese que esta notación señala ese
                            // campo como el campo elegido
oneYearOlder := {number := age.number+1};
ageInMonths := age.number * 12;
```

La notación de lista de valores no deberá utilizarse para los valores del tipo **union**.

6.3.5.1 Referencia a los campos de un tipo union

Para hacer una referencia a los campos de un tipo **union** deberá utilizarse la notación de puntos (véase 6.3.1.1).

EJEMPLO:

```
MyVar5 := MyUnion1.myChoice1;
// En el caso de un tipo union jerarquizado dentro de otro tipo, la referencia tendría
// las siguientes características:
MyVar6 := MyRecord1.myElement1.myChoice2;
// Obsérvese que se hace referencia al campo que tiene el identificador 'myChoice2'
// de un tipo union incluido dentro de un tipo record
```

6.3.5.2 Carácter facultativo en el caso de union

Dado que el tipo **union** no puede tener campos facultativos, la palabra clave **optional** no se podrá utilizar con los tipos **union**.

6.3.5.3 Definición de tipos con subniveles de jerarquía para tipos de campos

La notación TTCN-3 soporta la definición de tipos para los campos union con subniveles de jerarquía dentro de la definición de union, de manera similar a los tipos record que se describen en 6.3.1.3.

6.4 El tipo anytype (cualquier tipo)

El tipo especial **anytype** es una forma simplificada de indicar todos los *tipos de datos conocidos* y el tipo **address** en un módulo TTCN-3. Véase la definición de "tipos conocidos" en 3.1; el tipo **anytype** abarca todos los tipos de datos conocidos, salvo los tipos **port**, **component** y **default**. El tipo **address** se incluirá siempre que se haya definido explícitamente dentro de ese módulo.

Los nombres de los campos de **anytype** se han de identificar de forma única con los correspondientes nombres de tipo.

NOTA 1 – Como resultado de este requisito, los tipos importados con nombres confusos (con un identificador de una definición en el módulo importador o con un identificador importado de un tercer módulo) no podrán ser alcanzados a través del **anytype** del módulo importador.

EJEMPLO:

```
// La siguiente utilización de anytype es válida
var anytype MyVarOne, MyVarTwo;
var integer MyVarThree;

MyVarOne.integer := 34;
MyVarTwo := {integer := MyVarOne.integer + 1};

MyVarThree := MyVarOne.integer * 12;
```

El tipo **anytype** se define en un nivel local para cada módulo y, al igual que otros tipos predefinidos, no puede ser importado directamente por otro módulo. Ahora bien, sí es posible que otro módulo importe un tipo **anytype** que ha sido definido por el usuario.

NOTA 2 – Un tipo **anytype** definido por el usuario "contiene" todos los tipos importados en el módulo en el que se declara. Se debería considerar atentamente la importación de estos tipos definidos por el usuario en un módulo, por sus posibles efectos secundarios.

6.5 Matrices

En la notación TTCN-3, como en muchos lenguajes de programación, se considera que las matrices no son tipos, pero sí pueden especificarse en el caso de una declaración de variables. Es posible hacer una declaración de matriz simple o multidimensional. Las dimensiones de la matriz se deben especificar mediante expresiones constantes que se traducirán a valores **integer** positivos.

EJEMPLO 1:

```
var integer MyArray1[3];          // Ejemplares de una matriz de enteros de 3 elementos
                                  // con índices 0 a 2
var integer MyArray2[2][3];       // Ejemplares de una matriz de enteros bidimensional
                                  // de 2 × 3 elementos con índices de(0,0)a(1,2)
```

Se accede a los elementos de una matriz por medio de la notación de índice ([]), que debe especificar un índice válido dentro de la gama de la matriz. Se puede acceder a los elementos individuales de las matrices multidemensionales mediante la utilización reiterada de la notación de índice. El acceso a elementos externos a la gama de la matriz provocará un error de hora de compilación o de caso de prueba.

EJEMPLO 2:

```
MyArray1[1] := 5;
MyArray2[1][2] := 12;

MyArray1[4] := 12;    // ERROR: el índice debe estar entre 0 y 2
MyArray2[3][2] := 15; // ERROR: el primer índice debe ser 0 ó 1
```

Las dimensiones de una matriz también se pueden especificar utilizando gamas: los valores inferior y superior de la gama definen los valores de índices inferior y superior.

EJEMPLO 3:

```
var integer MyArray3[1 .. 5];          // Ejemplares de una matriz de enteros de 5 elementos
                                       // con índices 1 a 5
MyArray3[1] := 10; // Índice más bajo
MyArray3[5] := 50; // Índice más alto

var integer MyArray4[1 .. 5][2 .. 3 ]; // Ejemplares de una matriz de enteros bidimensional
                                       // de 5 × 2 elementos con índices de (1,2) a (5,3)
```

Los valores de elementos de una matriz tendrán que ser compatibles con la correspondiente declaración de variable. Es posible asignar los valores separadamente mediante una notación de lista de valores o una notación indexada, o asignar varios valores o todos utilizando una notación de lista de valores. Si se utiliza esta notación, el primer valor de la lista se asigna al primer elemento de la matriz (índice 0), el segundo al segundo elemento y así sucesivamente. Para excluir un elemento de la asignación es preciso omitirlo o evitarlo explícitamente en la lista. En el caso de asignación de valores a matrices multidimensionales, cada dimensión a la que se asigne tendrá que traducirse en un conjunto de valores presentados entre corchetes. Si se especifican valores de matrices multidimensionales, la dimensión más a la izquierda corresponde a la estructura más a la derecha del valor, y la dimensión más a la derecha corresponde a la estructura más interna. Se permite la utilización de sectores de matrices multidimensionales, es decir, cuando el número de índices del valor de la matriz es menor que el número de dimensiones en la definición de la matriz correspondiente. Los índices de los sectores de matriz deben corresponder a las dimensiones de la definición de la matriz de izquierda a derecha (es decir, el primer índice del sector corresponde a la primera dimensión de la definición). Los índices del sector deben ser conformes a las dimensiones de la definición de la matriz correspondiente.

EJEMPLO 4:

```
MyArray1[0]:= 10;
MyArray1[1]:= 20;
MyArray1[3]:= 30;

// otra posibilidad con lista de valores
MyArray1:= {10, 20, -, 30};

MyArray4:= {{1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10}};
    // El valor de la matriz está plenamente definido

var integer MyArray5[2][3][4] :=
{
  {
    {1, 2, 3, 4}, // asigna un valor al sector MyArray5 [0][0]
    {5, 6, 7, 8}, // asigna un valor al sector MyArray5 [0][1]
    {9, 10, 11, 12} // asigna un valor al sector MyArray5 [0][2]
  }, // termina las asignaciones al sector MyArray5 [0]
  {
    {13, 14, 15, 16}, {17, 18, 19, 20}, {21, 22, 23, 24}
  } // asigna un valor al sector MyArray5 [1]
};

MyArray4[2] := {20, 20};
    // arroja {{1, 2}, {3, 4}, {20, 20}, {7, 8}, {9, 10}};
MyArray5[1] := { {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}};
    // arroja {{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}},
    //         {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}}};

MyArray5[0][2] := {3, 3, 3, 3};
    // arroja {{{1, 2, 3, 4}, {5, 6, 7, 8}, {3, 3, 3, 3}},
    //         {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}}};
```

```
var integer MyArrayInvalid[2][2];
MyArrayInvalid := { 1, 2, 3, 4 }
    // no válida porque la dimensión de la notación de valor no corresponde
    // a las dimensiones de la definición
MyArrayInvalid[2] := { 1, 2 }
    // no válida porque el índice del sector debería ser 0 ó 1
```

NOTA – También es posible utilizar estructuras de datos multidimensionales mediante los tipos record, record of, set y set of.

EJEMPLO 5:

```
// 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 tendría esta característica
myRecordArray[1].field1 := 1;
```

6.6 Tipos recursivos

Los tipos de la notación TTCN-3 pueden definirse de forma recursiva si procede. Ahora bien, el usuario deberá asegurarse de que la recursión de todos los tipos se podrán traducir y que no se producirá un proceso recursivo infinito.

6.7 Compatibilidad de tipos

6.7.0 Consideraciones generales

En general, en la notación TTCN-3 deberá garantizarse la compatibilidad de tipos entre valores de asignaciones, ejemplificaciones y comparaciones.

Para las explicaciones que nos ocupan, el valor efectivo que se asigna y se transfiere como parámetro, etc., se denomina valor "b". El tipo de valor "b" se denomina tipo "B". El tipo de parámetro formal que permite obtener el valor efectivo del valor "b" se denomina tipo "A".

6.7.1 Compatibilidad de tipos no estructurados

En el caso de variables, constantes, plantillas, etc., de tipos básicos simples y tipos bitstring, hexstring y octetstring, el valor "b" es compatible con el tipo "A" si el tipo "B" se traduce en el mismo tipo raíz que el tipo "A" (por ejemplo, **`integer`**) y no infringe las condiciones de subtipos (por ejemplo, gamas, restricciones de longitud) del tipo "A".

EJEMPLO:

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

// Entonces
y := 5; // es una asignación válida

x := y;
// es una asignación válida porque y tiene el mismo tipo raíz que x
// y no hay infracción de formación de subtipos

x := 20; // es una asignación válida
y := x;
// NO es una asignación válida porque el valor de x está fuera de la gama
// de MyInteger

x := 5; // es una asignación válida
y := x;
// es una asignación válida porque este valor de x sí está dentro de la
// gama de MyInteger

//Siendo
type charstring MyChar length (1);
type charstring MySingleChar length (1);
var MyChar myCharacter;
var charstring myCharString;
var MySingleChar mySingleCharString := "B";
```

```
//Entonces
myCharString := mySingleCharString;
// es una asignación válida porque charstring restringida a la longitud 1
// es compatible con charstring.
myCharacter := mySingleCharString;
// es una asignación válida porque dos charstring con longitud de carácter
// simple son compatibles.

// Siendo
myCharString := "abcd";

// Entonces
myCharacter := myCharString[1];
// es válida porque la notación r.h.s. apunta a un solo elemento de la cadena

// Siendo
var charstring myCharacterArray [5] := {"A", "B", "C", "D", "E"}

//Entonces
myCharString := myCharacterArray[1];
// es válida y asigna el valor "B" a myCharString;
```

En el caso de variables, constantes, plantillas etc. de tipo **charstring**, el valor 'b' es compatible con una **universal charstring** tipo 'A' a menos que infrinja un tipo de especificación de restricción (gama, lista o longitud) de tipo "A".

En el caso de variables, constantes, plantillas etc. de tipo **universal charstring**, el valor 'b' es compatible con una charstring tipo 'A' si todos los caracteres del valor 'b' tienen caracteres correspondientes (es decir, el mismo carácter de control o gráfico emplea el mismo código de carácter) en el tipo **charstring** y no infringen ninguna especificación de restricción de tipo (gama, lista o longitud) de tipo "A".

6.7.2 Compatibilidad de tipos estructurados

6.7.2.0 Consideraciones generales

Cuando se trata de tipos estructurados (excepto el tipo **enumerated**) un valor "b" de tipo "B" es compatible con el tipo "A" si las estructuras de valores efectivos de los tipos "B" y "A" son compatibles; entonces es posible hacer asignaciones, ejemplificaciones y comparaciones.

6.7.2.1 Compatibilidad de los tipos enumerated

Los tipos enumerated nunca son compatibles con otros tipos básicos o estructurados (en el caso de los tipos enumerated es necesaria una determinación rigurosa de tipos).

6.7.2.2 Compatibilidad de los tipos record y record of

En el caso de los tipos **record**, las estructuras de valores efectivos son compatibles si son idénticas las características de número y aspecto facultativo de los campos en el orden textual de definición, si los tipos de cada campo son compatibles y si el valor de cada campo existente del valor "b" es compatible con el tipo del campo correspondiente en el tipo "A". El valor de cada campo del valor "b" se asigna al campo correspondiente en el valor del tipo "A".

EJEMPLO 1:

```
// Siendo
type record AType {
    integer     a(0..10)    optional,
    integer     b(0..10)    optional,
    boolean     c
}

type record BType {
    integer     a           optional,
    integer     b(0..10)    optional,
    boolean     c
}

type record CType {         // tipo con nombres de campos diferentes
integer         d           optional,
integer         e           optional,
boolean         f

}
```

```
type record DType {           // tipo cuyo campo c es facultativo
        integer         a           optional,
        integer         b           optional,
        boolean         c           optional
}

type record EType {           // tipo con un campo adicional d
        integer         a     optional,
        integer         b     optional,
        boolean         c,
        float           d     optional
}

        var AType MyVarA := { -, 1, true};
        var BType MyVarB := { omit, 2, true};
        var CType MyVarC := { 3, omit,  true};
        var DType MyVarD := { 4, 4, true};
        var EType MyVarE := { 5, 5, true, omit};

        // Entonces

        MyVarA := MyVarB;     // es una asignación válida,
                              // el valor de MyVarA es ( a := <undefined>, b:= 2, c:= true)
        MyVarC := MyVarB;     // es una asignación válida
                              // el valor de MyVarC es ( d := <undefined>, e:= 2, f:= true)
        MyVarA := MyVarD;     // NO es una asignación válida porque no coincide el carácter
                              // facultativo de los campos
        MyVarA := MyVarE;     // NO es una asignación válida porque no coincide el número de campos

        MyVarC := { d:= 20 };// valor efectivo de MyVarC es { d:=20, e:=2, f:= true }
        MyVarA := MyVarC      // NO es una asignación válida porque el campo 'd' de MyVarc
                              // infringe la norma de subtipos del campo 'a' de AType
```

En el caso de tipos **record of** y matrices, las estructuras de valores efectivos son compatibles si los tipos componentes son compatibles y el valor "b" del tipo "B" no infringe ninguna norma de longitud de subtipos del tipo **record of** ni de dimensión de la matriz de tipo "A". Los valores de los elementos del valor "b" se asignarán de forma secuencial al ejemplar del tipo "A", incluidos los elementos no definidos.

Los tipos **record of** y las matrices unidimensionales son compatibles con los tipos **record** si sus estructuras de valores efectivos son compatibles y si el valor "b" del tipo **record of** "B", o la dimensión de la matriz "b", tiene exactamente el mismo número de elementos que el tipo **record** "A". El carácter facultativo de los campos del tipo **record** no es un factor de compatibilidad: no afecta el total de campos (los campos facultativos se han de incluir siempre en el total). Los valores de elementos del tipo **record of** o la matriz deberán asignarse al ejemplar de un tipo **record** en el orden textual de la correspondiente definición del tipo **record**, incluidos los elementos no definidos. Si un elemento que no tiene un valor definido se asigna a un elemento facultativo de **record**, se omitirá el elemento facultativo. Se producirá un error si se intenta asignar un elemento que no tiene un valor definido a un elemento obligatorio de **record**.

NOTA – Si no hay ninguna restricción de longitud para el tipo **record of** o la restricción de longitud es superior al número de elementos del tipo **record** comparado, y el índice de cualquiera de los elementos definidos del valor **record of** es igual al número de elementos del tipo **record** menos uno, o es inferior, habrá compatibilidad en todos los casos.

De otra parte, es posible asignar los valores de un tipo **record** a un ejemplar de un tipo **record of** o una matriz unidimensional si no se infringe ninguna norma de longitud del tipo **record of** o la dimensión de la matriz es igual al número de elementos del tipo **record** o superior. Los elementos facultativos que faltan en el valor **record** se deberán asignar como elementos con valores no definidos.

EJEMPLO 2:

```
// Siendo
    type record HType {
    integer     a,
    integer     b optional,
    integer     c
}

type record of integer IType

var HType MyVarH := { 1, omit, 2};
var IType MyVarI;
var integer MyArrayVar[2];
```

```
// Entonces

MyArrayVar := MyVarH;
// es una asignación válida porque el tipo de MyArrayVar es compatible con HType

MyVarI := MyVarH;
// es una asignación válida porque los tipos son compatibles y no se infringe ninguna
// norma de subtipos

MyVarI := { 3, 4 };
MyVarH := MyVarI;
// NO es una asignación válida porque el campo obligatorio "c" de Htype no recibe
// ningún valor
```

6.7.2.3 Compatibilidad de los tipos set y set of

Los tipos **set** sólo son compatibles con otros tipos **set** y **set of**. Para los tipos **set** y **set of** se aplicarán las mismas reglas de compatibilidad de los tipos **record** y **record of**.

NOTA 1 – Por tanto, aunque no se conoce el orden de los elementos en el envío y la recepción, el orden textual de los campos en la definición del tipo es determinante para la compatibilidad de los tipos **set**.

NOTA 2 – El orden de los campos en los valores **set** puede ser arbitrario, aunque esto no afecta la compatibilidad de tipos porque los nombres de campos identifican de modo inequívoco la correspondencia entre los campos de los tipos **set** y los valores **set**.

EJEMPLO:

```
// Siendo
type set FType {
    integer     a     optional,
    integer     b     optional,
    boolean     c
}

type set GType {
    integer     d     optional,
    integer     e     optional,
    boolean     f
}

var FType MyVarF := { a:=1, c:=true };
var GType MyVarG := { f:=true, d:=7};

// Entonces

MyVarF := MyVarG; // es una asignación válida porque los tipos FType y GType son compatibles

MyVarF := MyVarA; // NO es una asignación válida porque MyVarA es un tipo record
```

6.7.2.4 Compatibilidad entre subestructuras

Las reglas de compatibilidad de tipos estructurados de esta cláusula también valen para la subestructura de dichos tipos.

EJEMPLO:

```
// Siendo
type record JType {
HType    H,
integer  b     optional,
integer  c
}

var JType MyVarJ

// Considerando las anteriores declaraciones,

MyVarJ.H := MyVarH;
// es una asignación válida porque el tipo del campo H de JType es compatible con HType

MyVarI := MyVarJ.H;
// es una asignación válida porque Itype es compatible con el tipo del campo H de JType
```

6.7.3 Compatibilidad de los tipos component

La compatibilidad de los tipos componente ha de tenerse en cuenta en dos condiciones diferentes.

1) Compatibilidad entre un valor de referencia componente y un tipo componente (por ejemplo, cuando se pasa una referencia componente como un parámetro efectivo a una función o un altstep, o cuando se

asigna un valor de referencia componente a una variable de un tipo componente distinto): una referencia componente "b" del tipo componente "B" es compatible con el tipo componente "A" si todas las definiciones de "A" tienen definiciones idénticas en "B".

2) Compatibilidad de **runs on**: una función o altstep que hace referencia a un tipo componente "A" en su cláusula **runs on** puede invocarse o arrancarse en un ejemplar de componente de tipo 'B' si todas las definiciones de "A" tienen definiciones idénticas en "B".

La identidad de las definiciones en 'A' con definiciones de 'B' se determina basándose en las siguientes reglas:

- Para los ejemplares de puertos, es una condición que sean idénticos el tipo y el identificador.
- Para los ejemplares de temporizador los identificadores deben ser idénticos y ambos tendrán duraciones iniciales idénticas o bien no tendrán duración inicial.
- Para los ejemplares de variables y las definiciones constantes, los identificadores, los tipos y los valores de inicialización deben ser idénticos (en el caso de las variables, significa que faltan en ambas definiciones o que son idénticos).
- Para definiciones de plantilla local los identificadores, los tipos, las listas de parámetros formales y los valores de campo de plantilla asignada o de plantilla deben ser idénticos.

6.7.4 Compatibilidad de tipos para operaciones de comunicación

Las operaciones de comunicación (véase la cláusula 23) **`send`**, **`receive`**, **`trigger`**, **`call`**, **`getcall`**, **`reply`**, **`getreply`** y **`raise`** son excepciones a la regla de tolerancia de compatibilidad de tipos y requieren una tipificación rigurosa. De otra parte, los tipos de valores o plantillas utilizados directamente como parámetros de estas operaciones deben definirse explícitamente en la definición de tipo de puerto asociada. La condición de tipificación rigurosa vale también para el almacenamiento del valor, la dirección o la referencia componente recibidos durante una operación **`receive`** o **`trigger`**.

6.7.5 Conversión de tipos

Si es necesario convertir valores de un tipo a valores de otro tipo, y estos tipos no se derivan del mismo tipo básico, se utilizará una de las funciones de conversión predefinidas del anexo C o una función definida por el usuario.

EJEMPLO:

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

7 Módulos

7.0 Consideraciones generales

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

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

7.1 Denominación de los módulos

Los nombres de los módulos se forman con un identificador TTCN-3. Además, una especificación de módulo puede transportar un atributo facultativo que se identifica mediante la palabra clave **`language`** que permite identificar la versión del lenguaje TTCN-3 en el que se especificó el módulo. Hoy en día se soportan las siguientes cadenas de lenguaje: "TTCN-3:2001" para una especificación de módulo que cumple con la versión TTCN-3 2001, "TTCN-3:2003" para la versión 2003, "TTCN-3:2005" para la versión 2006.

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

EJEMPLO:

```
module SIPTestSuite language "TTCN-3:2003"
{ … }
```

7.2 Parámetros de módulos

7.2.0 Consideraciones generales

La lista de parámetros de módulo define un conjunto de valores que son suministrados por el entorno de prueba en el momento de ejecución y que serán tratados como constantes durante la ejecución de la prueba. Los parámetros de módulo se declaran especificando el tipo y mencionando sus identificadores después de la palabra clave **modulepar**. Los parámetros del módulo no deben ser de tipo port, default o component. Un parámetro de módulo debe ser sólo del tipo address si dicho tipo está definido explícitamente en el módulo asociado. Los parámetros de módulo sólo podrán declararse dentro de la parte definición del módulo. Puede haber varias declaraciones de parámetros de módulo, pero cada parámetro sólo podrá declararse una vez (no se permite redefinir un parámetro de módulo).

EJEMPLO:

```
module MyModulewithParameters
{
    modulepar integer TS_Par0, TS_Par1;
    modulepar boolean TS_Par2;
    modulepar hexstring TS_Par3;
}
```

7.2.1 Valores de parámetros de módulo por defecto

Se permite especificar valores por defecto para parámetros de módulo, mediante una asignación en la lista de parámetros de módulo. El valor por defecto sólo puede ser un valor literal y se tiene que asignar en el lugar de la declaración del parámetro. Si el sistema de prueba no notifica un valor efectivo de ese parámetro en el momento de ejecución, la prueba se hará con el valor por defecto y en otros casos con el valor efectivo notificado por el sistema de prueba.

EJEMPLO:

```
module MyModuleDefaultParameter
{
    modulepar integer TS_Par0 := 0, TS_Par1;
    modulepar boolean TS_Par2 := true;
    :
}
```

7.3 Parte de definiciones del módulo

7.3.0 Consideraciones generales

La parte de definiciones del módulo especifica las definiciones más generales del módulo y puede importar identificadores de otros módulos. Véanse en 5.3 las reglas de ámbito para las declaraciones que se hacen en la parte de definiciones del módulo y las declaraciones importadas. En el cuadro 1 se indican los elementos de lenguaje que pueden definirse en un módulo TTCN-3. Las definiciones del módulo pueden ser importadas por otros módulos.

EJEMPLO:

```
module MyModule
{   // Este módulo sólo contiene definiciones
    :
    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, funciones, altsteps o tipos component.

NOTA – La notación TTCN-3 no soporta la declaración de variables en la parte de definiciones del módulo. Por tanto, no es posible definir variables globales en la TTCN-3. Sin embargo, las variables definidas en un componente de prueba podrán utilizarse en todos los casos de prueba, funciones, etc. que se ejecuten en ese componente; las variables definidas en la parte de control permiten mantener los valores independientemente de la ejecución del caso de prueba.

7.3.1 Grupos de definiciones

Las definiciones pueden reunirse en grupos denominados en la parte de definiciones del módulo. Es posible especificar un grupo de declaraciones cuando se permite una sola declaración. Los grupos pueden ser jerarquizados, es decir, un grupo puede contener otros grupos. El creador de una sucesión de pruebas puede utilizar esta posibilidad para estructurar conjuntos de datos de prueba o funciones que describen el comportamiento de la prueba, entre otras cosas.

La agrupación se hace para que el módulo sea más legible y para mejorar su estructura lógica, si es necesario. No se asocia un ámbito a los grupos o grupos jerarquizados *excepto* en el contexto de identificadores y atributos que se asignan al grupo mediante una instrucción **`with`** asociada. Por tanto:

- Los identificadores de grupo no tienen que ser necesariamente únicos en todo el módulo. Ahora bien, todos los identificadores de grupo de subgrupos de un solo grupo tienen que ser únicos. Si es necesario se utilizará la notación de puntos para identificar de forma única los subgrupos en la jerarquía, por ejemplo para importar un determinado subgrupo.
- Véanse en 28.4 las reglas de reemplazo de atributos.

EJEMPLO:

```
module MyModule {
    :
    // Un conjunto de definiciones
    group MyGroup {
          const integer MyConst:= 1;
           :
          type record MyMessageType { … };
          group MyGroup1 {  // Subgrupo con definiciones
                type record AnotherMessageType { … };
                const boolean MyBoolean := false
          }
    }


    // Un grupo de altsteps
    group MyStepLibrary {
          group MyGroup1 {  // Subgrupo que tiene el mismo nombre que el subgrupo
                            // con definiciones
                altstep MyStep11() { … }
                altstep MyStep12() { … }
                 :
                altstep MyStep1n() { … }
          }
          group MyGroup2 {
                altstep MyStep21() { … }
                altstep MyStep22() { … }
                :
                altstep MyStep2n() { … }
          }
    }
    :
}

// Instrucción para importar MyGroup1 en MyStepLibrary
import from MyModule {
    group MyStepLibrary.MyGroup1
}
```

7.4 Parte de control del módulo

La parte de control del módulo puede contener definiciones locales y describe el orden de ejecución (posiblemente repetitivo) de los casos de prueba efectivos. Los casos de prueba se definen en la parte de definiciones del módulo y se invocan 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; por tanto, puede ejecutarse
```

```
        control
        {
              var boolean MyVariable; // variable de control local
              :
              execute( MyTestCase1()); // ejecución secuencial de casos de prueba
              execute( MyTestCase2());
              :
        }
    }
```

7.5 Importación desde un módulo

7.5.0 Consideraciones generales

Es posible reutilizar definiciones especificadas en otros módulos utilizando la instrucción **`import`**. Como en la notación TTCN-3 no hay construcciones de exportación explícitas, por defecto pueden importarse todas las definiciones de módulo en la parte de definiciones del módulo. La instrucción **`import`** puede utilizarse 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 una instrucción **`with`**), se importarán también los atributos. En la cláusula 28.6 se explican los mecanismos para modificar atributos de definiciones importadas.

NOTA – Si el módulo tiene atributos globales, estos atributos serán asociados a definiciones que no los tienen.

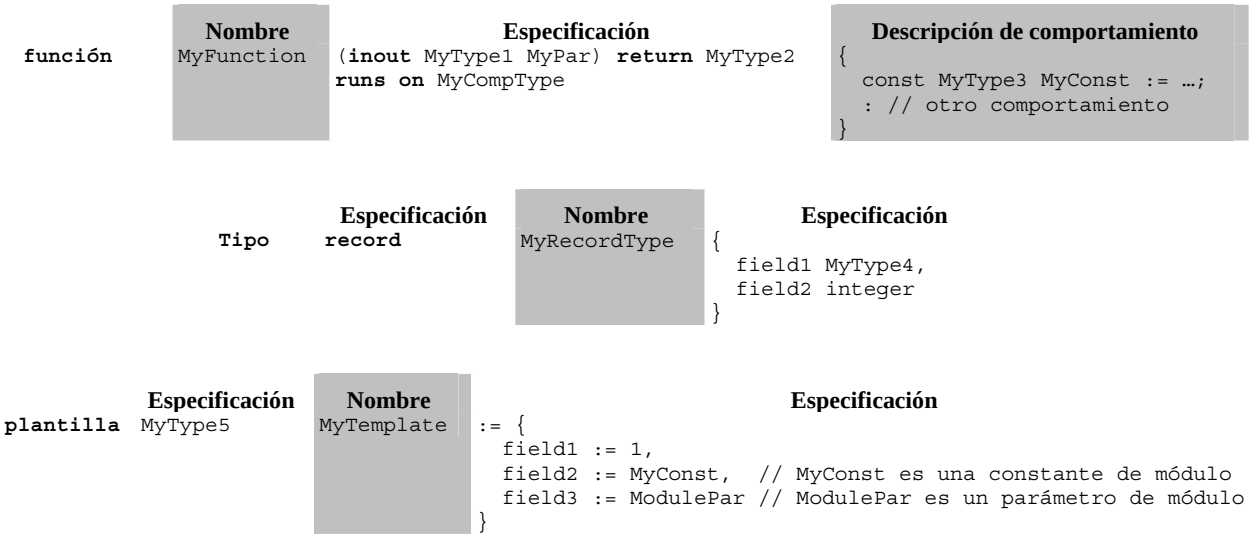
EJEMPLO:

```
module MyModuleA
{       // Este módulo contiene definiciones y definiciones importadas
        :
        const integer MyConstant := 1;
        import from MyModuleB all;       // El ámbito de las definiciones importadas es
                                         // global para MyModuleA
        import from MyModuleC {
            type MyType1, MyType2;
            template all
        }
        type record MyMessageType { … }
        :
        function MyBehaviourC()
        {
                  const integer MyConstant := 2;
                  // en este caso no se puede importar
                   :
        }
        :
        control
        {   // en este caso no se puede importar
            :
        }
}
```

7.5.1 Estructura de definiciones que pueden importarse

La notación TTCN-3 soporta la importación de las siguientes definiciones: parámetros de módulo, tipos definidos por el usuario, firmas, constantes, constantes externas, plantillas de datos, plantillas de firmas, funciones, funciones externas, alternativas (altsteps) y casos de prueba. Cada definición tiene un *nombre* (que corresponde al identificador de la definición, por ejemplo un nombre de función) y una *especificación* (por ejemplo, una especificación de tipo o una firma de una función); las funciones, las alternativas (altsteps) y los casos de prueba tienen además una *descripción de comportamiento* asociada.

EJEMPLO:



Las descripciones de comportamiento no influyen en el mecanismo de importación porque se considera que estas características son invisibles para el importador de las funciones, las alternativas (altsteps) o los casos de prueba. Por eso no se consideran en las siguientes descripciones.

La parte de especificación de una definición que se puede importar contiene *definiciones locales* (por ejemplo, nombres de campos en definiciones de tipos estructurados, o valores de tipos enumerados) y *definiciones por referencia* (por ejemplo, referencias a definiciones de tipos, plantillas, constantes o parámetros de módulo). Para los anteriores ejemplos sería:

	Nombre	Definiciones locales	Definiciones por referencia
función	MyFunction	MyPar	MyType1, MyType2, MyCompType
tipo	MyRecordType	field1, field2	MyType4, integer
plantilla	MyTemplate		MyType5, field1, field2, field3, MyConst, ModulePar

NOTA 1 – La columna de definiciones locales se refiere únicamente a identificadores diferentes definidos en la definición que se puede importar. También pueden considerarse como definiciones locales los valores asignados a los distintos campos de estas definiciones, por ejemplo en definiciones de plantillas, pero éstos no son importantes para la explicación del mecanismo de importación.

NOTA 2 – Las definiciones por referencia field1, field2 y field3 de la plantilla MyTemplate son los nombres de campos de MyType5: se hace referencia a ellos mediante MyType5.

Las definiciones por referencia también pueden importarse: como se ha dicho, el origen de una definición por referencia puede estructurarse en un nombre y una parte de especificación, y esta parte también contiene definiciones locales y por referencia. Dicho de otra forma, las definiciones de importación se pueden construir de forma recursiva a partir de otras definiciones del mismo tipo.

El mecanismo de importación de la notación TTCN-3 se aplica a las definiciones locales y por referencia utilizadas en la parte de especificación de las definiciones que pueden importarse. En el cuadro 5 se indican las definiciones locales y por referencia posibles en este tipo de definiciones.

Cuadro 5/Z.140 – Definiciones locales y por referencia posibles de las definiciones que pueden importarse

Definición que puede importarse	Definiciones locales posibles	Definiciones por referencia posibles
Parámetro de módulo		Tipo parámetro de módulo
Tipo definido por el usuario (todos)	Nombres de parámetros	Tipo de parámetro
• tipo enumerado	Valores concretos	
• tipo estructurado	Nombres de campos	Tipos de campos
• tipo puerto		Tipos de mensajes, firmas
• tipo componente	Nombres de constantes, de variables, de temporizadores y de puertos	Tipos de constantes, de variables y de puertos
Firma	Nombres de parámetros	Tipos de parámetros, tipo de respuestas, tipos de excepciones
Constante		Tipo de constante
Constante externa		Tipo de constante
Plantilla de datos	Nombres de parámetros	Tipo de plantilla y de parámetros, constantes, parámetros de módulo, funciones
Plantilla de firma		Definición de firma, constantes, parámetros de módulo, funciones
Función	Nombres de parámetros	Tipos de parámetros, de respuesta y de componente (cláusula **`runs`** on)
Función externa	Nombres de parámetros	Tipos de parámetros y de respuesta
Alternativa (altstep)	Nombres de parámetros	Tipos de parámetros y de componente (cláusula **`runs on`**)
Caso de prueba	Nombres de parámetros	Tipos de parámetros y de componente (cláusula **`runs on`** y **`system`**)

El mecanismo de importación de la notación TTCN-3 distingue entre el *identificador de una definición por referencia* y la *información necesaria para utilizar una definición por referencia* dentro de la definición importada. El identificador de una definición por referencia no se importa automáticamente porque no es necesario para utilización.

7.5.2 Reglas sobre la utilización de importaciones

La importación está sometida a las siguientes reglas:

a) Sólo pueden importarse las definiciones del nivel superior del módulo. No se importarán definiciones de un ámbito más bajo (por ejemplo, constantes locales definidas en una función).

b) Sólo se permite importar directamente del módulo de origen de una definición (en el que reside la definición efectiva para el identificador señalado por referencia en la instrucción **`import`**).

c) Con la definición se importan su nombre y todas las definiciones locales.

NOTA 1 – Una definición local, por ejemplo un nombre de campo de un tipo record definido por el usuario, sólo es significativa en el contexto de las definiciones correspondientes, por ejemplo, un nombre de campo de un tipo record sólo podrá utilizarse para acceder a un campo de este tipo y no fuera de este contexto.

d) Con la definición se importa toda la información de definiciones por referencia que son necesarias para utilizar este tipo de definición.

NOTA 2 – Las instrucciones de importación son transitivas, es decir, si un módulo A importa una definición de un módulo B que utiliza una referencia de tipo definida en el módulo C, la información correspondiente necesaria para utilizar ese tipo será importada automáticamente en el módulo A.

e) Los identificadores de definiciones referenciadas no son importados automáticamente.

NOTA 3 – Si se quieren utilizar las definiciones referenciadas en el módulo que realiza la importación, es preciso importarlas explícitamente del módulo de origen.

f) En el caso de importación de una función, una alternativa (altstep) o un caso de prueba, las correspondientes especificaciones de comportamiento y todas las definiciones utilizadas dentro de ellas serán invisibles para el módulo que realiza la importación.

g) Las importaciones cíclicas están prohibidas.

EJEMPLO:

```
module ModuleONE {

    modulepar integer ModPar1, ModPar2 := 7


    type record RecordType_T1 {
          integer     Field1_T1,
          boolean     Field2_T1
    }

    type record RecordType_T2 {
          RecordType_T1     Field1_T2,  // Utilización de RecordType_T1
          RecordType_T1     Field2_T2,
          integer           Field3_T2
    }

    const integer MyConst := 13;

    template RecordType_T2 Template_T2 (RecordType_T1 TempPar_T2):= { // parameterized
                                                                      // template
          Field1_T2 := TempPar_T2,            // Referencia a parámetro de plantilla
          Field2_T2 := {MyConst, true},       // Referencia a constante de módulo
          Field3_T2 := ModPar1                // Referencia a parámetro de módulo
    }

} // fin del módulo ModuleONE

module ModuleTWO {

  import from ModuleONE {
    template Template_T2
    }

    // Sólo los nombres Template_T2 y TempPar_T2 serán visibles en ModuleTWO.
    // Obsérvese que el identificador TempPar_T2 sólo puede utilizarse en el
    // contexto de Template_T2, por ejemplo para comunicar un valor de
    // parámetro efectivo. Toda la información necesaria para utilizar
    // Template_T2, por ejemplo para verificación del tipo, será importada
    // para las definiciones referenciadas RecordType_T2, RecordType_T1,
    // Field1_T2, Field2_T2,Field3_T3, MyConst y ModPar1, pero sus
    // identificadores no son visibles en ModuleTWO. Entonces, por ejemplo,
    // no es posible utilizar la constante MyConst ni declarar una variable
    // del tipo RecordType_T1 o RecordType_T2 en ModuleTWO sin importar
    // explícitamente estos tipos.

    import from ModuleONE {
          modulepar ModPar2
    }

    // El parámetro de módulo ModPar2 de ModuleONE será importado de ModuleONE
    // y puede utilizarse como una constante de número entero

} // fin del módulo ModuleTWO

module ModuleTHREE {

  import from ModuleONE all; // importa todas las definiciones de ModuleONE

  type port MyPortType {
    inout RecordType_T2
  }

  type component MyCompType {
    var integer MyComponentVar := ModPar2;    // Referencia a un parámetro de módulo,
                                              // de ModuleONE
    port MyPortType MyPort
  }

  function MyFunction () return integer {
    return MyConst    // Devuelve una constante de módulo definida en ModuleONE
    }

  testcase MyTestCase (out RecordType_T2 MyPar) runs on MyCompType {

    var integer MyTCVar := ModPar2;     // Referencia a un parámetro de módulo, de ModuleONE
```

```
    MyPort.send(Template_T2);     // Envío de una plantilla definida en ModuleONE
    MyPort.receive(RecordType_T2 : ?) -> value MyPar; // El valor recibido se asigna al
                                                      // parámetro out MyPar.

  } // fin del caso de prueba MyTestCase

} // fin de ModuleTHREE

module ModuleFOUR {

  import from ModuleTHREE {
   testcase MyTestCase
  }

    // Sólo los nombres MyTestCase y MyPar serán visibles y disponibles para utilización
    // en ModuleFOUR. La información de tipo para RecordType_T2 es importada a través
    // de ModuleTHREE desde ModuleONE, y la información de tipo para MyCompType es importada
    // de ModuleTHREE. Todas las definiciones utilizadas en la parte de comportamiento
    // de MyTestCase quedan ocultas para el usuario de ModuleFOUR.

} // fin de ModuleFOUR
```

7.5.3 En blanco

7.5.4 Importación de definiciones separadamente

Es posible importar una definición separadamente.

EJEMPLO:

```
import from MyModuleA {
  type MyType1       // importar una definición de tipo de MyModuleA
}

import from MyModuleB {
  type MyType2, Mytype3, MyType4;      // importa tres tipos
  template MyTemplate1;                // importa una plantilla
  const MyConst1, MyConst2             // importa dos constantes
}
```

7.5.5 Importación de todas las definiciones de un módulo

Es posible importar todas las definiciones de la parte de definiciones de un módulo si el nombre del módulo se acompaña de la palabra clave **`all`**. Cuando se importan de esta forma todas las definiciones de un módulo no deberá utilizarse ninguna otra forma de importación (una definición separadamente, de la misma clase, etc.) para la misma instrucción **`import`**.

EJEMPLO 1:

```
import from MyModule all;
```

Es posible excluir determinadas declaraciones en la importación, indicando las clases y los identificadores en la lista de excepciones entre corchetes y después de la palabra clave **except**.

EJEMPLO 2:

```
import from MyModule all except {
  type MyType3, MyType5
    // excluye de la instrucción de importación las declaraciones de tipo MyType3
    // y MyType5 pero importa las demás declaraciones de MyModule
}
```

La palabra clave **`all`** también puede utilizarse en la lista de excepciones: excluye de la instrucción de importación todas las declaraciones de la misma clase.

EJEMPLO 3:

```
import from MyModule all except {
  type MyType3, MyType5;   // excluye los dos tipos en la instrucción de importación
  template all             // excluye todas las plantillas declaradas en MyModule en
                           // la instrucción de importación
}
```

7.5.6 Importación de grupos

Es posible importar grupos de definiciones.

EJEMPLO 1:

```
import from MyModule {
    group MyGroup
}
```

La importación agrupada tiene el mismo efecto que una instrucción **import** que incluyera todas las definiciones que pueden importarse de este grupo (incluyendo subgrupos).

Los grupos de la notación TTCN-3 sólo se utilizan para estructurar y no son unidades de ámbito. Por tanto, es posible importar directamente un subgrupo (grupo definido en otro grupo), sin el grupo que lo contiene. Si el nombre del subgrupo a importar no es único en el mismo módulo (véase 7.3.1) se utilizará la notación de puntos para identificar de forma única el subgrupo a importar.

Es posible excluir de la importación determinadas definiciones de un grupo, indicando las clases y los identificadores en la lista de excepciones entre corchetes y precedidos de la palabra clave **except**.

EJEMPLO 2:

```
import from MyModule {
    group MyGroup except {
          type MyType3, MyType5
                // excluye las definiciones de tipo MyType3 y MyType5 en
                // la instrucción de importación, pero importa las demás
                // definiciones de MyGroup
    }
}
```

La palabra clave **all** también puede utilizarse en la lista de excepciones: excluye de la instrucción de importación todas las definiciones de la misma clase.

EJEMPLO 3:

```
import from MyModule {
    group MyGroup except {
          type MyType3, MyType5;  // excluye los dos tipos en la instrucción de importación
                                  // y excluye todas las plantillas definidas en MyGroup
          template all            // en la instrucción de importación
    }
}
```

7.5.7 Importación de definiciones de la misma clase

La palabra clave **all** permite importar todas las definiciones de un módulo que son de la misma clase. La palabra clave all que se emplea con la palabra clave constant permite identificar todas las constantes y todas las constantes externas declaradas en la parte de definiciones del módulo al que se refiere la instrucción import. De manera similar, la palabra clave **all** que se emplea con la palabra clave function permite identificar todas las funciones y todas las funciones externas que se definen en el módulo señalado por la instrucción import.

EJEMPLO 1:

```
import from MyModule {
    type all;         // importa todos los tipos de MyModule
    template all      // importa todas las plantillas de MyModule
}
```

Es posible no incluir determinadas declaraciones de una clase en la importación, indicando los identificadores después de la palabra clave **except**.

EJEMPLO 2:

```
import from MyModule {
  type all except MyType3, MyType5;    // importa todos los tipos excepto MyType3 y MyType5
  template all                         // importa todas las plantillas definidas en Mymodule
}
```

7.5.8 Tratamiento de conflictos de nombres al importar

Todos los módulos de la notación TTCN-3 tendrán su propio espacio de nombre en el cual todas las definiciones serán identificadas de manera única. La importación puede producir conflictos de nombres: importación de módulos diferentes. Para resolver los conflictos de nombres se añadirá a la definición importada (que origina el conflicto de nombres), como prefijo, el identificador del módulo del cual es importada. El prefijo y el identificador estarán separados por un punto (.).

Cuando no hay ambigüedades no es necesario añadir este prefijo (pero puede añadirse) para la utilización de definiciones importadas. Si la definición está referenciada en el mismo módulo en que se define, puede utilizarse igualmente el identificador del módulo (módulo actual) como prefijo del identificador de la definición.

EJEMPLO:

```
module MyModuleA  {
   :
  type bitstring MyTypeA;
  import from SomeModuleC {
    type  MyTypeA,          // Donde MyTypeA es cadena de caracteres
          MyTypeB           // Donde MyTypeB es cadena de caracteres
  }
    :
  control {
     :
    var   SomeModuleC.MyTypeA MyVar1 := "Test String"; // Prefijo obligatorio
    var MyTypeA MyVar2 := '10110011'B; // Es el MyTypeA original
     :
    var MyTypeB MyVar3 := "Test String"; // Prefijo no obligatorio …
    var SomeModuleC.MyTypeB MyVar3 := "Test String";// … pero puede añadirse
     :
  }
}
```

NOTA – Se supone que las definiciones que tienen el mismo nombre y están definidas en distintos módulos siempre son diferentes, incluso si las definiciones en los diferentes módulos efectivamente son idénticas. Por ejemplo, si se importa un tipo que ya está definido localmente, incluso con el mismo nombre, habrá dos tipos diferentes en el módulo.

7.5.9 Tratamiento de múltiples referencias a la misma definición

La utilización de **`import`** para una determinada definición, 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. El sistema deberá resolver estos casos e importar una sola vez la definición.

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

Se considera que todas las instrucciones **`import`** y las definiciones de estas instrucciones se tratan separadamente, una tras otra en el orden de presentación. Es importante señalar que una instrucción **`except`** no excluye las definiciones indicadas de toda importación en general; todas las instrucciones de importación de definiciones de la misma clase son una forma de notación simplificada de una lista equivalente de identificadores de definiciones consideradas separadamente. La instrucción **`except`** sólo excluye las definiciones de esta lista.

EJEMPLO :

```
import from MyModule {
    type all except MyType3;      // importa todos los tipos de MyModule excepto MyType3
    type MyType3                  // importa MyType3 explícitamente
}
```

7.5.10 Importación de definiciones de módulos que no son TTCN

Cuando se importan definiciones de fuentes que no son módulos TTCN es preciso especificar el lenguaje (eventualmente también la versión) de la fuente (módulo, paquete, biblioteca, incluso fichero) de importación de las definiciones. Para ello se introduce la palabra clave **`language`** y una declaración textual del lenguaje indicado. No es obligatorio especificar el lenguaje cuando se importa de un módulo TTCN-3 de la misma edición del módulo importador. Cuando se detecta incompatibilidad entre la identificación de lenguaje (incluida la identificación implícita al omitir la especificación de lenguaje) y la sintaxis del módulo del que se importan las definiciones, las herramientas disponibles permitirán resolver el conflicto con un esfuerzo razonable.

Se definen los siguientes identificadores de lenguaje TTCN-3:

- 'TTCN-3:2001' – se empleará con módulos conformes a la versión 2001 de esta Recomendación (véase la Bibliografía).
- TTCN-3:2003' – se empleará con módulos conformes a la versión 2003 de esta Recomendación (véase la Bibliografía).
- 'TTCN-3:2005' – se empleará con módulos conformes a esta Recomendación.

EJEMPLO:

```
import from MyModule language "TTCN-3:2003" {
    type MyType
}
```

NOTA – El mecanismo de importación permite reutilizar definiciones de otros módulos TTCN-3 o de módulos de otros lenguajes. Si se importan definiciones de especificaciones escritas en otros lenguajes, por ejemplo paquetes SDL, en unos casos se podrán aplicar las reglas de la TTCN-3 y en otros habrá que definir reglas particulares.

8 Configuraciones de prueba

8.0 Consideraciones generales

La notación TTCN-3 permite la especificación (dinámica) de configuraciones de prueba concurrentes (que llamaremos simplemente configuraciones). 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.

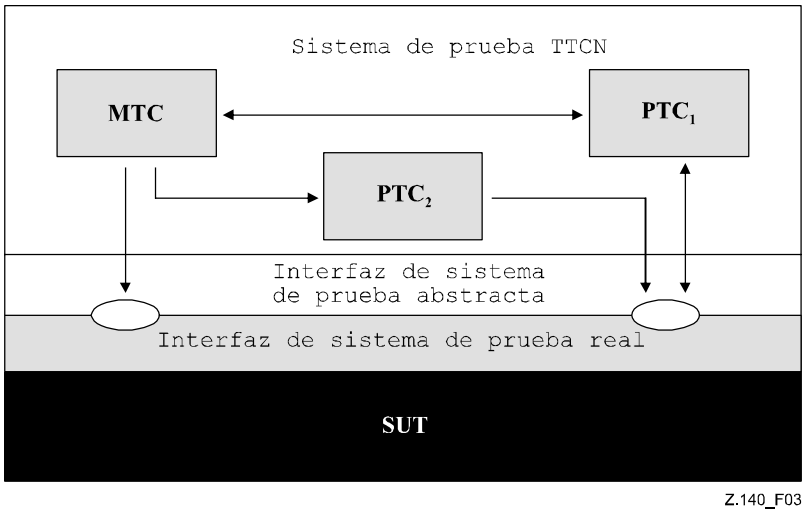


Figura 3/Z.140 – Visión conceptual de la configuración de prueba TTCN-3

Dentro de cada configuración habrá solamente un componente de prueba principal (MTC, *main test component*). Los otros son componentes de prueba paralelos (PTC, *parallel test component*). El sistema deberá crear automáticamente el MTC al empezar a ejecutar cada caso de prueba. El comportamiento definido en la descripción principal del caso de prueba será ejecutado en este componente. Durante la ejecución de un caso de prueba pueden crearse dinámicamente otros componentes, utilizando explícitamente la operación **`create`**.

La ejecución del caso de prueba terminará cuando termine el MTC. Todos los demás PTC reciben el mismo tratamiento, es decir, no hay relación jerárquica explícita entre ellos y la terminación de un solo PTC no termina otros componentes ni el MTC. Al terminar el MTC el sistema tiene que detener todos los PTC que no hubieran terminado al finalizar la ejecución del caso de prueba.

La comunicación se realiza a través de puertos de comunicación, sea entre componentes de prueba o entre los componentes y la interfaz del sistema de prueba (véase 8.1).

Los tipos de componentes de prueba y los tipos de puertos, indicados por las palabras clave **component** y **por**t, serán definidos en la parte de definición del módulo. Para definir la configuración efectiva de los componentes y las conexiones entre ellos se ejecutan las operaciones **create** y **connect** dentro del comportamiento del caso de prueba. Para conectar los puertos de componentes a los puertos de la interfaz del sistema de prueba se realiza la operación **map** (véase 22.2).

8.1 Modelo de comunicación a través de puertos

Los componentes de prueba están conectados a través de sus puertos: las conexiones entre componentes y entre un componente y la interfaz del sistema de prueba se efectúan por puertos. El modelo de cada puerto es una cola FIFO (el primero en entrar es el primero en salir) infinita que almacena los mensajes entrantes o las solicitudes de procedimientos hasta que son procesados por el componente de ese puerto.

NOTA – En principio el número de puertos TTCN-3 es infinito, pero en un sistema de prueba real puede haber desbordamiento: esta situación ocasionaría un error de caso de prueba (véase 25.2.1).

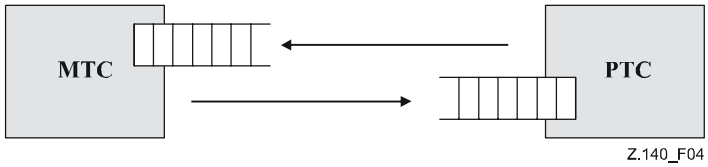


Figura 4/Z.140 – Modelo de puertos de comunicación de TTCN-3

8.2 Restricciones relativas a las conexiones

En la notación TTCN-3 hay conexiones puerto a puerto, y conexiones puerto-interfaz del sistema de prueba (véase la figura 5). El número de conexiones de un componente no está restringido, y también se permiten conexiones uno a muchos (por ejemplo, figuras 5(g) o 5(h)).

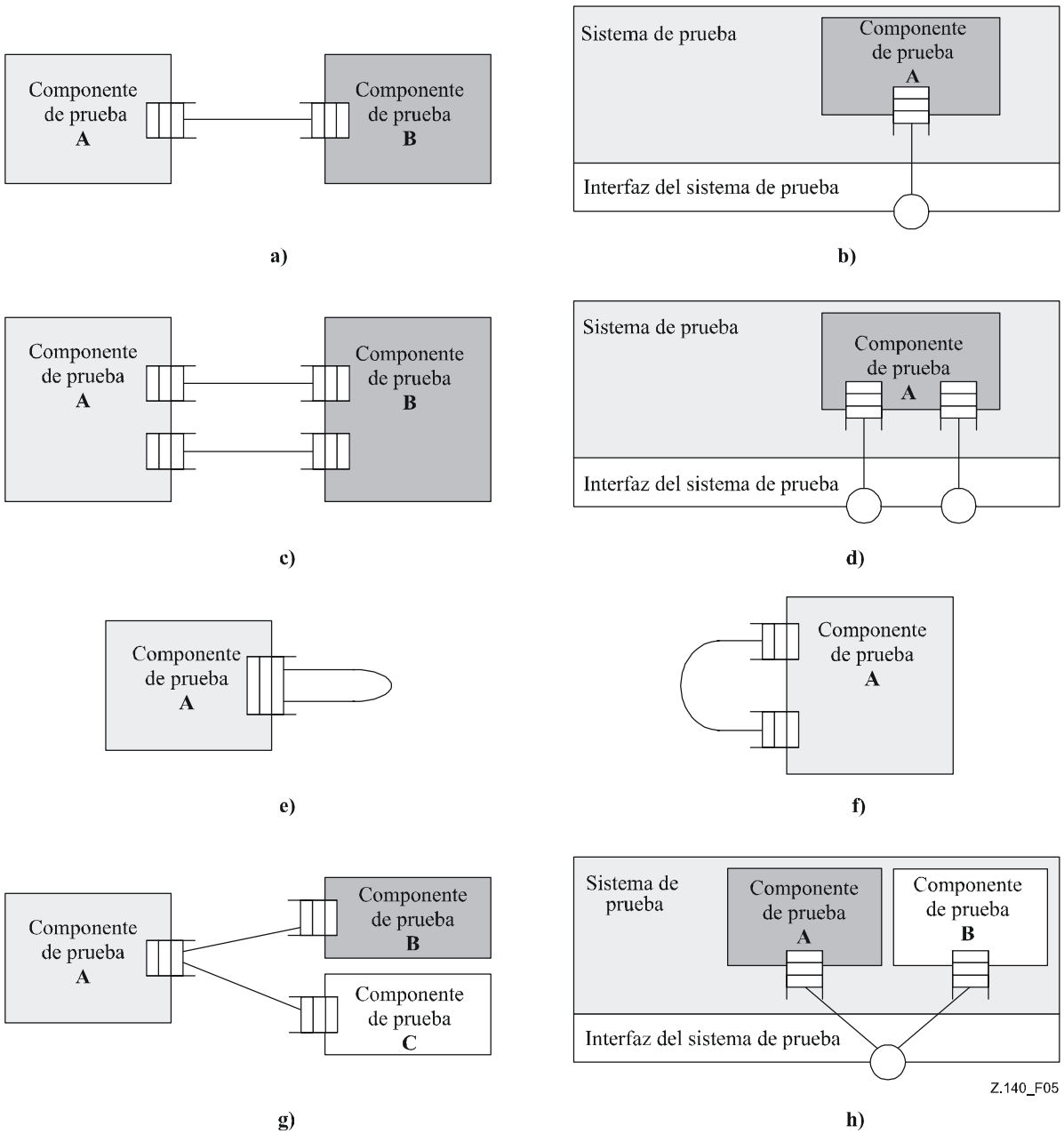


Figura 5/Z.140 – Las conexiones permitidas

Las siguientes conexiones no están permitidas:

- No se podrá conectar un puerto de un componente A con dos o más puertos del mismo componente (figuras 6(a) y 6(e)).
- No se podrá conectar un puerto de un componente A con dos o más puertos del componente B (véase la figura 6(c)).
- Entre un puerto del componente A y la interfaz del sistema de prueba sólo puede haber conexión uno a uno. Por este motivo no están permitidas las conexiones representadas en la figuras 6(b) y 6(d).
- No puede haber conexiones dentro de la interfaz del sistema de prueba (véase la figura 6(f)).
- Un puerto que está conectado no debe tener una relación y un puerto que tiene una relación no debe estar conectado (véase la figura 6(g)).

Como la notación TTCN-3 permite configuraciones y direcciones dinámicas, no siempre es posible verificar las restricciones de conexiones al hacer la compilación. Habrá que hacer la verificación en el momento de ejecución y si procede se producirá un error de caso de prueba.

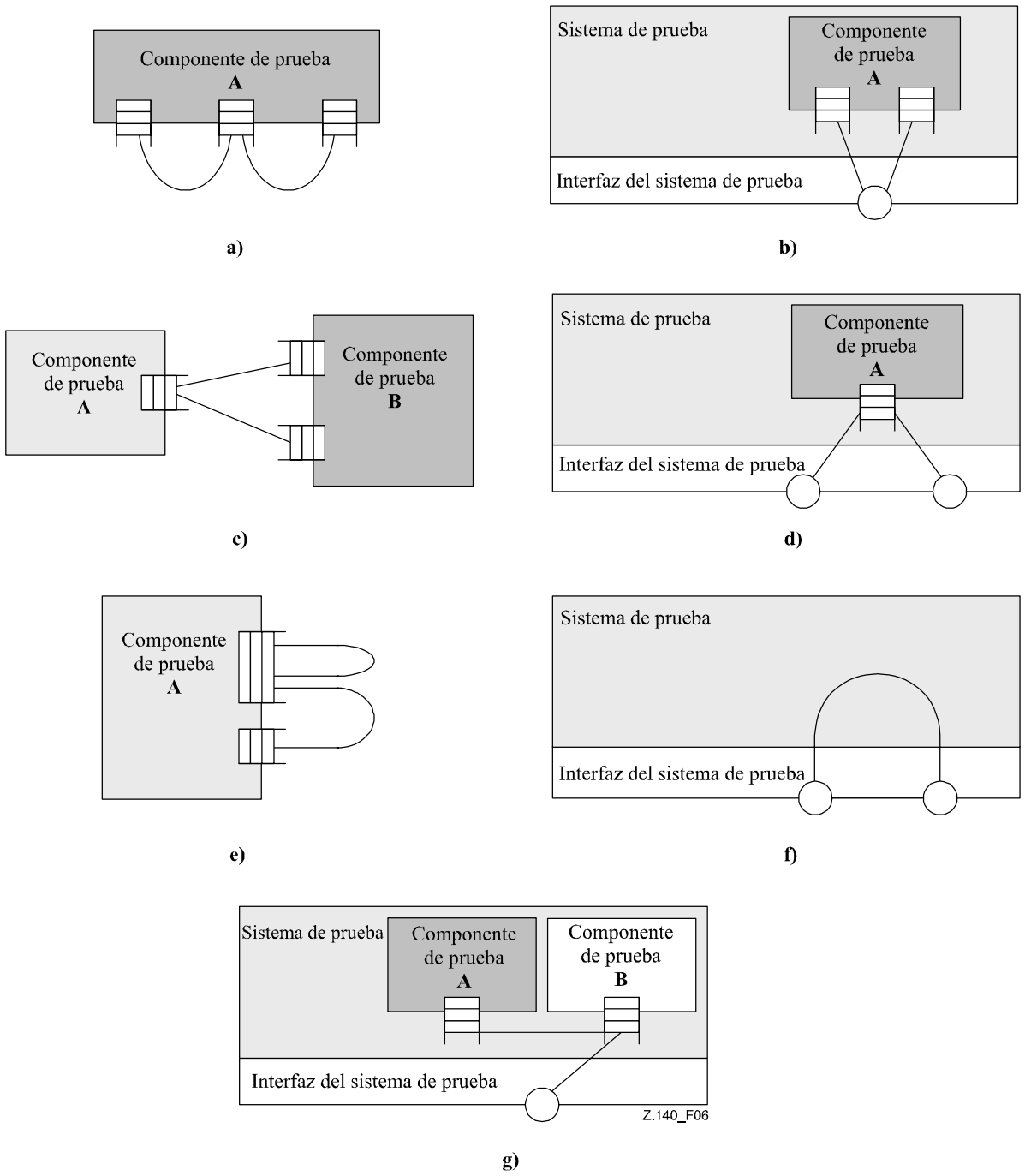


Figura 6/Z.140 – Las conexiones que no están permitidas

8.3 Interfaz de sistema de prueba abstracta

La notación TTCN-3 se utiliza para probar implementaciones. El objeto que se prueba es la implementación sometida a prueba (IUT, *implementation under test*). Puede ser una IUT que tiene interfaces directas para la prueba, pero también puede ser una IUT que forma parte del sistema; en el segundo caso se utiliza la denominación sistema sometido a prueba (SUT, *system under test*). En el caso mínimo, la IUT y el SUT son equivalentes. En la presente Recomendación se utiliza el término SUT de manera general para significar el SUT o la IUT.

Los casos de prueba tienen que comunicar con el SUT en un entorno de prueba real, pero la especificación de la conexión física está fuera del ámbito de la TTCN-3. Ahora bien, hay que asociar una interfaz de sistema de prueba bien definida (pero abstracta) a cada caso de prueba. La definición de interfaz de sistema de prueba es idéntica a la definición de componente: una lista de todos los puertos de comunicación posibles a través de los cuales se conecta el caso de prueba con el SUT.

La interfaz del sistema de prueba define estáticamente el número y el tipo de conexiones por puertos con el SUT durante una prueba. Sin embargo, las conexiones entre la interfaz del sistema de prueba y los componentes de prueba TTCN-3 son dinámicas y pueden modificarse durante una prueba mediante las operaciones **map** y **unmap** (véanse 22.2 y 22.3).

8.4 Definición de tipos de puertos de comunicación

8.4.0 Consideraciones generales

Los puertos facilitan la comunicación entre componentes de prueba y, de otra parte, entre estos componentes y la interfaz del sistema de prueba.

La notación TTCN-3 soporta puertos de comunicación por mensajes o por procedimientos y hay que definir si se trata de un puerto del primer tipo o del segundo (o mixto como se indica en 8.4.1). Para identificar el primer tipo de puertos se utilizará la palabra clave **message** y para el segundo tipo la palabra clave **procedure** dentro de la definición del tipo de puerto asociada.

Los puertos son bidireccionales y tienen un sentido de comunicación que se indica con las palabras clave **in** (entrada), **out** (salida) e **inout** (ambos sentidos). Cada definición de tipo de puerto tendrá una o más listas que indiquen el conjunto permitido de tipos (mensajes) y/o procedimientos, así como el sentido de comunicación autorizado.

Cuando se define una firma (véase asimismo la cláusula 13) en el sentido "out" de un puerto por procedimiento, los tipos de todos sus parámetros **inout** y **out**, su tipo return y sus tipos exception automáticamente pasan a formar parte del sentido "in" de este puerto. Cuando se define una firma en el sentido in de un puerto por procedimiento, los tipos de todos sus parámetros **inout** y **out**, su tipo return y sus tipos exception automáticamente pasan a formar parte del sentido out de este puerto.

EJEMPLO 1:

```
// Puerto por mensajes que puede recibir los 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 por procedimientos que permite una solicitud 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 – Como el término mensaje puede significar mensajes definidos por plantillas o valores efectivos resultantes de expresiones, la lista que restringe las condiciones de utilización de un puerto basado en mensajes es simplemente una lista de nombres de tipos.

8.4.1 Puertos mixtos

Es posible definir un puerto que permita las dos clases de comunicación, utilizando la palabra clave **mixed**. Las listas para estos puertos mixtos también serán mixtas, con firmas y tipos. En la definición no hay ninguna separación.

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

Un puerto mixto en TTCN-3 es una forma de notación simplificada para dos puertos: un puerto por mensajes y un puerto por procedimientos con el mismo nombre. Las operaciones de comunicación distinguen entre los dos puertos en el momento de ejecución.

Cuando se utilice un identificador de puerto mixto, las operaciones utilizadas para controlar puertos, es decir, **start**, **stop** y **clear** (véase 23.5) se ejecutarán en ambas colas (en un orden arbitrario).

8.5 Definición de tipos component

8.5.0 Consideraciones generales

El tipo **component** define los puertos que están asociados a un componente. Estas definiciones deberán hacerse en la parte de definiciones del módulo. Los nombres de puertos en una definición de componente son locales para ese componente, lo que significa que otro componente puede tener puertos con los mismos nombres. Todos los puertos de un componente tendrán nombres únicos. El hecho de definir un componente no significa que hay conexiones entre componentes a través de esos puertos.

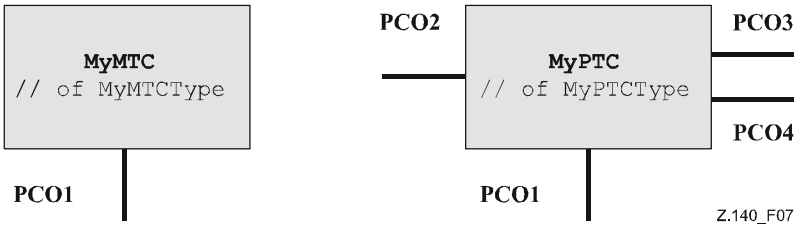


Figura 7/Z.140 – Los componentes típicos

EJEMPLO:

```
type component MyMTCType
{
    port MyMessagePortType              PCO1
}

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

8.5.1 Declaración de variables, constantes y temporizadores locales en un componente

Es posible declarar constantes, variables y temporizadores que son locales para un componente determinado.

EJEMPLO:

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

Estas declaraciones son visibles para todos los casos de prueba, todas las funciones y alternativas (altsteps) 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 al ejemplar del componente y se ajustan a las reglas de ámbito definidas en 5.3. Por tanto, todo nuevo ejemplar de componente tendrá su propio conjunto de variables y temporizadores especificado en la definición del componente (incluidos valores iniciales si se indican).

NOTA – Los componentes utilizados como interfaces del sistema de prueba (véase 8.8) no podrán utilizar ninguna de las constantes, variables y temporizadores declarados en el componente.

8.5.2 Definición de componentes con matrices de puertos

Es posible definir matrices de puertos en las definiciones del tipo component (véase también 22.12).

EJEMPLO:

```
type component My3pcoCompType
{
    port MyMessageInterfaceType PCO[3]
    port MyProcedureInterfaceType PCOm[3][3]
    // Define un tipo de componente que tiene una matriz de 3 puertos de mensaje
    // y una matriz bidimensional con 9 puertos de procedimiento.
}
```

8.5.3 Ampliación de los tipos component

Se pueden definir tipos component como ampliación de otros tipos component, gracias a la palabra clave extends.

EJEMPLO 1:

```
type component MyExtendedMTCType extends MyMTCType
{
    var float MyLocalFloat;
    timer MyOtherLocalTimer;
    port MyMessagePortType PCO2;
}
```

En esta definición se hace referencia a la nueva definición de tipo como *extended type* (*tipo ampliado*), y a la definición de tipo que sigue a la palabra clave **`extends`** como *parent type* (*tipo progenitor*).

Esta definición provocará que el tipo extended incluya además implícitamente todas las definiciones del tipo parent. Por consiguiente, la definición anterior equivale a escribir (y por lo cual se denomina *definición de tipo effective*):

EJEMPLO 2:

```
// efectivamente, la definición del ejemplo 1 es equivalente a ésta:
type component MyExtendedMTCType
{
    /* las definiciones de MyMTCType */
    var integer MyLocalInteger;
    timer MyLocalTimer;
    port MyMessagePortType PCO1

    /* las definiciones adicionales */
    var float MyLocalFloat;
    timer MyOtherLocalTimer;
    port MyMessagePortType PCO2;
}
```

Es válido ampliar tipos component que se definan mediante ampliación, siempre que no se cree una cadena cíclica de definiciones.

EJEMPLO 3:

```
type component MTCTypeA extends MTCTypeB { /* … */ };
type component MTCTypeB extends MTCTypeC { /* … */ };
type component MTCTypeC extends MTCTypeA { /* … */ };  // ERROR - ampliación cíclica
type component MTCTypeD extends MTCTypeD { /* … */ };  // ERROR - ampliación cíclica
```

Si se definen tipos component por ampliación, no se producirán conflictos de nombre entre las definiciones tomadas del tipo parent y las definiciones que se añaden al tipo extended, es decir, no se declarará un identificador de puerto, variable, constante, temporizador o plantilla tanto en el tipo parent (directamente o por ampliación) como en el tipo extended.

EJEMPLO 4:

```
type component MyExtendedMTCType extends MyMTCType
{
  var integer MyLocalInteger; // ERROR - already defined in MyMTCType (see example 2)
  var float MyLocalTimer;     // ERROR - timer with that name exists in MyMTCType
  port MyOtherMessagePortType PCO1; // ERROR - port with that name exists in MyMTCType
}

type component MyBaseComponent { timer MyLocalTimer };
type component MyInterimComponent extends MyBaseComponent { timer MyOtherTimer };
type component MyExtendedComponent extends MyInterimComponent
{
  timer MyLocalTimer; // ERROR - already defined in MyInterimComponent via extension
    }
```

Es válido que en una definición un tipo component amplíe varios tipos parent, siempre que en la definición se especifique como una lista de tipos separados por comas:

EJEMPLO 5:

```
type component MyCompA extends MyCompB, MyCompC, MyCompD {
  /* definiciones adicionales para MyCompA */
}
```

Asimismo, cualquiera de los tipos parent podrá definirse mediante una ampliación.

La definición de tipo component efectiva del tipo ampliado se obtiene como la colección de todas las definiciones de constante, variable, temporizador, puerto y plantilla derivadas de los tipos parent (determinadas recurrentemente si un tipo parent se define también por medio de una ampliación) y las definiciones declaradas directamente en el tipo extended. La definición de tipo component efectiva no debe tener conflictos de nombres. Para cumplir con este requisito, dentro del conjunto de los tipos parent que se emplean en la definición del tipo extended, todas las definiciones deben tener nombres únicos que diferirán de cualquiera de los nombres de las definiciones que se hayan declarado directamente en el tipo ampliado.

NOTA 1 – No se considera que haya declaraciones diferentes y por lo tanto no se produce un error si la misma definición de tipo ampliado se deriva de diferentes tipos de parent (a través de trayectos de ampliación diferentes).

EJEMPLO 6:

```
type component MyCompB { timer T };
type component MyCompC { var integer T };
type component MyCompD extends MyCompB, MyCompC {}
  // ERROR – confusión de nombres entre MyCompB y MyCompC

// MyCompB se definió antes
type component MyCompE extends MyCompB {
    var integer MyVar1 := 10;
}

type component MyCompF extends MyCompB {
    var float MyVar2 := 1.0;
}

type component MyCompG extends MyCompB, MyCompE, MyCompF {
  // Sin confusión de nombres.
  // Los tres tipos parent de MyCompG tienen un temporizador T, directamente o a través
  // de una ampliación de MyCompB; porque todos provienen (directamente o a través
  // de una ampliación) del temporizador T que se declaró en MyCompB, lo que valida
  // esta forma de colisión.
  /* aquí deben colocarse las definiciones adicionales */
}
```

La semántica de los tipos component con ampliaciones se define sustituyendo simplemente cada definición de tipo component por su definición de tipo component efectiva como un paso de pretratamiento antes de utilizarlo.

NOTA 2 – Para la compatibilidad de tipos component, esto quiere decir que una referencia de componente c de tipo CT1, que amplía CT2, es compatible con CT2, y que los casos de prueba, funciones y altsteps (alternativas) que especifican CT2 en sus cláusulas **runs on** pueden ejecutarse en c (véase 6.7.3).

8.6 Direccionamiento de entidades dentro del SUT

Un SUT puede estar formado por varias entidades y hay que direccionar cada una separadamente. Hay un tipo datos de dirección que se utiliza en operaciones de puertos para direccionar las entidades del SUT. Si se define con las palabras clave **to**, **from** y **sende**r, este tipo datos de dirección sólo se podrá utilizar en operaciones de recepción y envío de puertos que tienen una relación con la interfaz del sistema de prueba. La representación efectiva de datos de **address** se podrá traducir con una definición de tipo explícita dentro de la serie de pruebas, pero también se puede utilizar una traducción externa del sistema de prueba (tipo **address** abierto en la especificación TTCN-3), lo que permite especificar casos de prueba abstractos independientemente de cualquier mecanismo de dirección real específico del SUT.

En un módulo TTCN-3 sólo se podrán generar direcciones SUT explícitas si el tipo está definido dentro del módulo. Si el tipo no está definido dentro del módulo, las direcciones SUT explícitas sólo se transferirán como parámetros o se recibirán en campos de mensaje o como parámetros de solicitudes distantes de procedimientos.

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

EJEMPLO:

```
// Asocia el tipo integer con la dirección de tipo abierto
type integer address;
 :
// nueva variable de dirección inicializada con null
var address MySUTentity := null;
 :
// recibir un valor de dirección y asignarlo a la variable MySUTentity
PCO.receive(address:*) -> value MySUTentity;
 :
```

```
// utilización de la dirección recibida para enviar la plantilla MyResult
PCO.send(MyResult) to MySUTentity;
 :
// utilización de la dirección recibida para recibir una plantilla de confirmación
PCO.receive(MyConfirmation) from MySUTentity;
```

8.7 Referencias de componentes

Se trata de referencias únicas a los componentes de prueba creados durante la ejecución de un caso de prueba. Esta referencia única a un componente es generada por el sistema de prueba cuando se crea el componente, es decir, es el resultado de una operación **create** (véase 22.1). Algunas funciones predefinidas también devuelven referencias de componente: **system** (devuelve la referencia de componente para identificar los puertos de la interfaz del sistema de prueba), **mtc** (devuelve la referencia de componente del MTC) y **self** (devuelve la referencia del componente en el cual se invoca **self**).

Las referencias de componentes se utilizan en las operaciones de configuración **connec**t, **map** y **start** (véase la cláusula 22) para establecer configuraciones de prueba, y en las partes **from**, **to** y **sender** de operaciones de comunicación de los puertos conectados a componentes de prueba que no sean la interfaz del sistema de prueba (véanse la cláusula 23 y la figura 5) para direccionamiento.

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

El sistema de prueba deberá traducir externamente la representación efectiva de datos de referencias de componentes. Esto permite especificar casos de prueba abstractos independientemente de los entornos TTCN-3 reales en el momento de ejecución; en otras palabras, la notación TTCN-3 no restringe la implementación de un sistema de prueba con condiciones relativas al tratamiento y la identificación de componentes de prueba.

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

EJEMPLO:

```
// Definición de tipo de componente
type component MyCompType {
    port PortTypeOne PCO1;
    port PortTypeTwo PCO2
}

// Declaración de una variable para tratar referencias a componentes de tipo MyCompType
// y crear un componente de este tipo
var MyCompType MyCompInst := MyCompType.create;

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

// Utilización 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 en el que se tiene que deducir el emisor.
// Se puede utilizar el siguiente esquema:
 :
var M1 MyMessage, MyResult;
var MyCompType1 MyInst1 := null;
var MyCompType2 MyInst2 := null;
var MyCompType3 MyInst3 := null;
 :
alt {
  [] PCO1.receive(M1:?) from MyInst1 -> value MyMessage sender MyInst1 {}
  [] PCO1.receive(M1:?) from MyInst2 -> value MyMessage sender MyInst2 {}
  [] PCO1.receive(M1:?) from MyInst3 -> value MyMessage sender MyInst3 {}
}
 :
```

```
MyResult := MyMessageHandling(MyMessage);    // deducción de un 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};
 :
```

8.8 Definición de la interfaz del sistema de prueba

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

NOTA – Las variables, temporizadores y constantes que se hayan declarado en los tipos component y que se emplean como interfaces del sistema de prueba no tendrán efecto alguno.

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

En general, para cada caso de prueba que utilice más de un componente de prueba habrá que asociar una referencia de tipo de componente que defina la interfaz del sistema de prueba. Los puertos de interfaz del sistema de prueba deberán automáticamente ser ejemplificados por el sistema junto con el MTC al empezar a ejecutar el caso de prueba.

La operación **system** devuelve la referencia de componente de la interfaz del sistema de prueba. Deberá utilizarse para direccionar a los puertos del sistema de prueba.

EJEMPLO:

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

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

9 Declaración de constantes

Es posible declarar y utilizar constantes en la parte de definiciones del módulo, en definiciones de tipo de componente, la parte de control del módulo, los casos de prueba, las funciones y las alternativas (altsteps). La palabra clave **const** indica una definición de constante. Las constantes no deben ser del tipo puerto (port). El valor de la constante será asignado en el lugar de declaración.

NOTA – El único valor que puede asignarse a las constantes de los tipos default y component es el valor especial **null**.

EJEMPLO 1:

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

El valor se puede asignar a la constante dentro del módulo o externamente. En el segundo caso será una declaración de constante externa indicada por la palabra clave **external**.

EJEMPLO 2:

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

El tipo de una constante externa puede ser arbitrario, salvo los tipos port, default o component, pero tiene que ser conocido en el módulo: un tipo raíz o un tipo definido por el usuario definidos en el módulo o importados de otro módulo. La concordancia del tipo con la representación externa de una constante externa también está fuera del ámbito de la presente Recomendación, al igual que el mecanismo para transferir a un módulo el valor de una constante externa.

10 Declaración de variables

10.0 Consideraciones generales

Los tipos de las variables pueden ser: básico simple, cadena básica, de datos especiales (incluidos los subtipos que se derivan de éstos), address, component o default.

NOTA – Las variables de tipo estructurado y component pueden ser declaradas basándose sólo en los tipos de usuario definidos.

Es posible declarar y utilizar variables en la parte de control del módulo, en casos de prueba, en funciones y en alternativas (altsteps). También se pueden declarar variables en las definiciones del tipo component. Estas variables podrán utilizarse en casos de prueba, alternativas (altsteps) y funciones que se ejecutan en el tipo de componente considerado. No se declararán ni se utilizarán en la parte de definiciones del módulo (es decir, la TTCN-3 no soporta variables globales).

La utilización de variables no inicializadas o parcialmente inicializadas en posiciones distintas del lado izquierdo de las asignaciones o como parámetros efectivos trasladados a parámetros formales out, provocará un error.

10.1 Variables de valor

Es posible declarar una variable de valor mediante la palabra clave **var** seguida por un identificador de tipo y otro de variable. En el momento de la declaración se puede asignar un valor inicial. Las variables de valor deben almacenar sólo valores y pueden aprovecharse en las expresiones en los lados derecho e izquierdo de las asignaciones, a continuación de la palabra clave **return** en el cuerpo de las funciones con una cláusula return en sus encabezamientos, y pueden ser transferidas a parámetros de valor así como a parámetros formales de tipo template.

EJEMPLO:

```
var integer MyVar0:
var integer MyVar1 := 1;
var boolean MyVar2 := true, MyVar3 := false;
```

10.2 Variables tipo template (plantilla)

Es posible declarar variables template mediante la palabra clave **var template** seguida por un identificador de tipo y otro de variable. En el momento de la declaración se puede asignar un contenido inicial. Las variables template pueden almacenar, además de valores, mecanismos de concordancia (véase 14.3), que pueden ser útiles en los lados derecho e izquierdo de las asignaciones, a continuación de la palabra clave **return** en los cuerpos de las funciones que definen un valor return de tipo template en sus encabezamientos, y pueden ser transferidas como parámetros efectivos a parámetros formales de tipo template. Cuando se emplean en el lado derecho de las asignaciones no pueden ser operandos de operadores TTCN-3 (véase la cláusula 15) y la variable en el lado izquierdo también debe ser del tipo template. Asimismo, se permite asignar un ejemplar de plantilla a una variable template o a un campo de variable template.

NOTA – Las variables template, de forma similar a las plantillas locales y globales, se deben especificar plenamente a fin de que puedan ser empleadas en las operaciones de transmisión y recepción.

EJEMPLO:

```
template MyRecord MyTempl ( template boolean par_bool ) :=
    { field1 := par_bool, field2 := * }
:
function Myfunc () return template MyRecord {
  var template integer MyVarTemp1 := ?;
  var template MyRecord MyVarTemp2 := { field1 := true, field2 := * },
                MyVarTemp3 := { field1 := ?, field2 := MyVarTemp1 };
  MyVarTemp2 := MyTempl (?);
:
  return MyVarTemp2
}
```

No está permitido aplicar directamente operaciones de TTCN-3 a las variables template, pero se permite utilizar las notaciones de punto y de índice para supervisar y modificar los campos de las variables template. En 14.3.1 se presentan las reglas que han de aplicarse cuando dichas notaciones tratan de alcanzar campos fuera del mecanismo de concordancia.

11 Declaración de temporizadores

11.0 Consideraciones generales

Es posible declarar y utilizar temporizadores en la parte de control del módulo, en casos de prueba, funciones y alternativas (altsteps). También se pueden utilizar en casos prueba, funciones y alternativas (altsteps) que se ejecutan en el tipo de componente considerado. Es posible asignar un valor de duración por defecto facultativo a una declaración de temporizador. El temporizador se activará con este valor si no se especifica ningún otro. Ha de ser un valor **`float`** no negativo (es decir, 0,0 ó superior) con la base en segundos.

EJEMPLO 1:

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

timer MyTimer2;             // declaración de MyTimer2 sin un valor por defecto, es decir,
                            // hay que asignar un valor al activar el temporizador
```

No sólo es posible definir un ejemplar de temporizador, también matrices de temporización. Se utilizará una matriz de valores para asignar la duración (o duraciones) por defecto de los elementos de una matriz de temporización. Se utilizará la notación de valor de matriz que se especifica en 6.5 para asignar la duración (o duraciones) por defecto. Para no asignar una duración por defecto a determinados elementos de la matriz de temporización habrá que declararlo explícitamente mediante el símbolo "no se utiliza" ("-").

EJEMPLO 2:

```
timer t_Mytimer1[5] := { 1.0, 2.0, 3.0, 4.0, 5.0 }
  // duración por defecto para todos los elementos de la matriz de temporización.

timer t_Mytimer2[5] := { 1.0, -, 3.0, 4.0,5.0}
  // el segundo temporizador (t_Mytimer2[1]) no tiene duración por defecto.
```

11.1 Temporizadores como parámetros

Los temporizadores sólo se pueden transferir por referencia a funciones y alternativas (altsteps). Los temporizadores transferidos a una función o alternativa (altstep) son conocidos dentro de la definición de comportamiento correspondiente.

Un temporizador transferido como parámetro por referencia se puede utilizar como cualquier otro temporizador (no es necesario declararlo). También es posible transferir un temporizador activo a una función o una alternativa (altstep). El temporizador sigue su marcha (la operación no lo detiene implícitamente). Así pues, es posible tratar un evento de expiración de temporización dentro de la función o la alternativa (altstep) a las que se ha transferido 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

Una de las principales características de la notación TTCN-3 es la capacidad de enviar y recibir mensajes complejos en los puertos de comunicación definidos por la configuración de prueba. Pueden ser mensajes asignados explícitamente a la prueba del SUT o de coordinación interna, o mensajes 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, *protocol data units*) 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.

13 Declaración de firmas de procedimientos

13.0 Consideraciones generales

Es necesario utilizar firmas de procedimientos (que llamaremos simplemente firmas) en la comunicación por procedimientos. Puede utilizarse esta técnica para las comunicaciones dentro del sistema de prueba (entre componentes de prueba) o entre el sistema de prueba y el SUT. En el segundo caso es posible invocar un procedimiento en el SUT (es decir, el sistema de prueba hace la solicitud) o invocarlo en el sistema de prueba (es decir, el SUT hace la solicitud). Todos los procedimientos utilizados para comunicación entre componentes de prueba, solicitados desde el SUT o solicitados desde el sistema de prueba necesitan la definición de una firma de procedimiento (**signature)** en el módulo TTCN-3.

13.1 Firmas para la comunicación bloqueante y no bloqueante

La notación TTCN-3 soporta la comunicación *bloqueante* y *no bloqueante*. Las definiciones de firmas para la comunicación no bloqueante han de utilizar la palabra clave **noblock**, sólo podrán tener parámetros **in** (véase 13.2) y no devolverán ningún valor (véase 13.3), pero sí pueden plantear excepciones (véase 13.4). Por defecto, se supone que las definiciones de firmas que no tienen la palabra clave **noblock** se utilizan para la comunicación por procedimientos bloqueante.

EJEMPLO:

```
signature MyRemoteProcOne ();          // MyRemoteProcOne se utilizará para una comunicación
                                       // por procedimientos bloqueante. No tiene parámetros
                                       // y no hay ningún valor en respuesta.

signature MyRemoteProcTwo () noblock;  // MyRemoteProcTwo se utilizará para una comunicación
                                       // por procedimientos no bloqueante. No tiene
                                       // parámetros y no hay ningún valor en respuesta.
```

13.2 Parámetros de firmas de procedimientos

Las definiciones de firmas pueden incluir parámetros. La lista de parámetros de una definición **signature** puede incluir identificadores de parámetros, tipos de parámetros y el sentido de transmisión **in**, **out** o **inout**. Los sentidos **out** e **inout** indican que estos parámetros se utilizan para recabar información del procedimiento distante. Obsérvese que la dirección de los parámetros es la que percibe la parte *llamada*, no la parte *llamante*.

EJEMPLO:

```
signature MyRemoteProcThree (in integer Par1, out float Par2, inout integer Par3);
// MyRemoteProcThree se utilizará para una comunicación por procedimientos bloqueante.
// El procedimiento tiene tres parámetros: Par1 (parámetro in de tipo integer),
// Par2 (parámetro out de tipo float) y Par3 (parámetro inout de tipo integer).
```

13.3 Procedimientos distantes que devuelven un valor

Un procedimiento distante puede devolver un valor al terminar. Hay que especificar el tipo del valor en respuesta mediante una cláusula **return** en la definición de firma correspondiente.

EJEMPLO:

```
signature MyRemoteProcFour (in integer Par1) return integer;
// MyRemoteProcFour se utilizará para una comunicación por procedimientos bloqueante.
// El procedimiento tiene el parámetro in Par1 de tipo integer y devuelve un valor
// de tipo integer al terminar.
```

13.4 Especificación de excepciones

Las excepciones que pueden plantear los procedimientos distantes se representan como valores de un tipo específico en la notación TTCN-3. Esto permite utilizar plantillas y mecanismos de concordancia para especificar o verificar valores de respuesta de procedimientos distantes.

NOTA – Cada herramienta y sistema tiene su propia solución para convertir las excepciones generadas por el SUT o enviadas al SUT, en la correspondiente representación de tipo o SUT TTCN-3. Por eso está fuera del ámbito de la presente Recomendación.

Las excepciones se definen incluyendo una lista de excepciones en la definición de **signature**. En esta lista se definen todos los tipos diferentes posibles asociados a las excepciones posibles (habitualmente es la representación mediante valores específicos de estos tipos la que distingue el significado de las excepciones).

EJEMPLO:

```
signature MyRemoteProcFive (inout float Par1) return integer
        exception (ExceptionType1, ExceptionType2);
// MyRemoteProcFive se utilizará para comunicación por procedimientos bloqueante.
// Devuelve un valor float en el parámetro inout Par1, y un valor entero, o puede
// plantear excepciones de tipo ExceptionType1 o ExceptionType2

signature MyRemoteProcSix (in integer Par1) noblock
        exception (integer, float);
// MyRemoteProcSix se utilizará para comunicación por procedimientos no bloqueante.
// Si termina de forma insatisfactoria, MyRemoteProcSix plantea excepciones de tipo
// entero o float.
```

14 Declaración de plantillas

14.0 Consideraciones generales

Se utilizan plantillas para transmitir un conjunto de valores particulares o comprobar si un conjunto de valores recibidos concuerda con la especificación de una plantilla. Se pueden definir plantillas globalmente en una parte de definiciones del módulo, localmente en un caso de prueba, función, altstep (alternativa) o bloque de instrucciones, o en línea como argumentos de una operación de comunicación o de un parámetro efectivo de un caso de prueba, función o llamada altstep (alternativa).

Las plantillas proporcionan las siguientes posibilidades:

a) son una manera de organizar y reutilizar datos de pruebas, incluida una forma simple de herencia;

b) se pueden parametrizar;

c) permiten mecanismos de concordancia;

d) se pueden utilizar para comunicaciones por mensajes o por procedimientos.

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

En la declaración de plantilla hay que especificar un conjunto de valores básicos o símbolos de concordancia para cada uno de los campos definidos en la definición pertinente de tipo o de firma (especificación completa). Una declaración de plantilla modificada (véase 14.6) sólo especifica los campos a modificar en la plantilla de base (es una especificación parcial). El símbolo "no se utiliza" ("-") debe incluirse sólo en plantillas de firmas para parámetros que no son pertinentes, y también en las declaraciones de plantilla modificada y en las plantillas en línea modificadas para indicar que no se cambia un determinado campo o elemento.

Las funciones que se emplean en las expresiones están sujetas a determinadas restricciones (véase 16.1.4) cuando se especifican plantillas o campos de plantilla;

14.1 Declaración de plantillas de mensajes

14.1.0 Consideraciones generales

Pueden utilizarse plantillas para especificar ejemplares de mensajes con valores efectivos. Cabe considerar que una plantilla es un conjunto de instrucciones para construir un mensaje que se ha de enviar o para establecer la concordancia con un mensaje recibido.

Se pueden especificar plantillas para cualquier tipo TTCN-3 definido en el cuadro 3, salvo para los tipos **`port`** y **`default`**.

EJEMPLO:

```
// Si se utiliza en una operación de recepción, esta plantilla concordará con cualquier
// valor entero
//
template integer Mytemplate := ?;
// Esta plantilla sólo concordará con los valores enteros 1, 2 ó 3
template integer Mytemplate := (1, 2, 3);
```

14.1.1 Plantillas para enviar mensajes

Una plantilla utilizada en una operación **send** define un conjunto completo de valores de campos que comprenden el mensaje que se ha de transmitir por un puerto de prueba. La plantilla ha de estar totalmente definida al ejecutar la operación **send**, es decir, todos los campos deberán traducirse en valores efectivos y no se utilizará, directa ni indirectamente, ningún mecanismo de concordancia en los campos de plantilla.

NOTA – En el caso de plantillas de envío, la omisión de un campo facultativo es considerada como una notación de valor y no como un mecanismo de concordancia.

EJEMPLO:

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

// una plantilla de mensaje podría ser
template MyMessageType MyTemplate:=
{
    field1 := omit,
    field2 := "My string",
    field3 := true
}

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

14.1.2 Plantillas para recibir mensajes

Una plantilla utilizada en una operación **receive**, **trigger** o **check** es el modelo de los datos con los que ha de concordar un mensaje entrante. En las plantillas de recepción se pueden utilizar los mecanismos de concordancia definidos en el anexo B. No se producirá ninguna vinculación de los valores entrantes con la plantilla.

EJEMPLO:

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

// una plantilla de mensaje podría ser
template MyMessageType MyTemplate:=
{
    field1 := ?,
    field2 := pattern "abc*xyz",
    field3 := true
}

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

14.2 Declaración de plantillas de firma

14.2.0 Consideraciones generales

Es posible especificar listas de parámetros de procedimiento utilizando plantillas. Pueden definirse plantillas para cualquier procedimiento mediante una referencia a la definición de firma asociada. Una plantilla de firma permite definir sólo los valores y los mecanismos de concordancia de los parámetros del procedimiento pero no el valor return. Los valores y mecanismos correspondientes a return habrán de definirse dentro de la operación **reply** o **getreply** (véanse 23.3.3 y 23.3.4 respectivamente).

EJEMPLO:

```
// definición de firma para un procedimiento distante
signature RemoteProc(in integer Par1, out integer Par2, inout integer Par3) return integer;
```

```
// ejemplo de plantillas asociadas a una 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 invocar procedimientos

Una plantilla utilizada en una operación **call** o **reply** define un conjunto completo de valores de campos para todos los parámetros **in** e **inout**. Cuando se ejecute la operación **call**, todos los parámetros **in** e **inout** de la plantilla se traducirán en valores efectivos, y en estos campos no se utilizarán, directa ni indirectamente, mecanismos de concordancia. Si hubiera una especificación de plantilla para parámetros **out**, simplemente no se tiene en cuenta, por lo que se permite especificar mecanismos de concordancia para estos campos u omitirlos (véase el anexo B).

EJEMPLO:

```
// Considérense los ejemplos en la cláusula 14.2.0

// Esta invocación es válida porque todos los parámetros in e inout tienen
// un valor particular
MyPCO.call(RemoteProc:Template1);

// Esta invocación es válida porque todos los parámetros in e inout tienen
// un valor particular
MyPCO.call(RemoteProc:Template2);

// Esta invocación no es válida porque el parámetro inout Par3 tiene
// un atributo de concordancia, no un valor
MyPCO.call(RemoteProc:Template3);

// Las plantillas nunca devuelven valores. En el caso de Par2 y Par3,
// para extraer los valores que devuelve la operación call es necesario
// utilizar una cláusula de asignación al final de la instrucción call
```

14.2.2 Plantillas para aceptar invocaciones de procedimientos

Una plantilla utilizada en una operación **getcall** es un modelo de datos con los que han de concordar los campos de parámetros entrantes. En todas las plantillas empleadas por esta operación pueden utilizarse los mecanismos de concordancia definidos en el anexo B. No se producirá ninguna vinculación de valores entrantes con la plantilla. En el proceso de concordancia no se tendrán en cuenta los parámetros **out**.

EJEMPLO:

```
// Considérense los ejemplos la cláusula 14.2.0

// getcall válida, concordará si Par1 == 1 y Par3 == 3
MyPCO.getcall(RemoteProc:Template1);


// getcall válida, concordará si Par1 == 1 y Par3 == 3
MyPCO.getcall(RemoteProc:Template2);

// getcall válida, concordará si Par1 == 1 y con cualquier valor de Par3
MyPCO.getcall(RemoteProc:Template3);
```

14.3 Mecanismos de concordancia de plantillas

14.3.0 Consideraciones generales

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

También es posible utilizar mecanismos de concordancia y comodines en línea, sólo en eventos de recepción (es decir, operaciones **`receive`**, **`trigger`**, **`getcall`**, **`getreply`** y **`catch`**). Pueden aparecer en valores explícitos.

EJEMPLO 1:

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

El identificador de tipo es facultativo cuando el valor identifica el tipo de forma inequívoca.

EJEMPLO 2:

```
MyPCO.receive("AAAA"O);
```

NOTA – Los siguientes tipos son facultativos: integer, float, boolean, objid, bitstring, hexstring, octetstring.

Ahora bien, el tipo de plantilla en línea tiene que estar en la lista de puertos por los cuales se recibe la plantilla. Cuando hay ambigüedad entre el tipo de la lista y el tipo del valor proporcionado (formación de subtipos) es necesario incluir el nombre de tipo en la instrucción de recepción.

Hay cuatro grupos de mecanismos de concordancia:

a) valores específicos:
   - una expresión que se traduce en un valor específico;
   - **omit**: se omite el valor;

b) símbolos especiales que se pueden utilizar *en vez* de valores:
   - **(…)**: una lista de valores;
   - **complement (...)**: complemento de una lista de valores;
   - **?**: comodín para cualquier valor;
   - *****: comodín para cualquier valor o ningún valor (valor omitido)
   - **(***lowerBound..upperBound***)**: una gama de valores enteros o float entre los límites inferior y superior, incluidos éstos;
   - **superset**: al menos todos los elementos enumerados, es decir, posiblemente más;
   - **subset**: como máximo los elementos enumerados, es decir, posiblemente menos;

c) símbolos especiales que se pueden utilizar *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 (elemento omitido);
   - **permutation**: todos los elementos enumerados pero con un orden arbitrario (cabe señalar que, ? y * también son válidos como elementos de la lista de permutaciones);

d) símbolos especiales que describen *atributos* de valores:
   - **length**: restricción de longitud de cadena (tipos de cadena) o número de elementos de **`record of`**, **`set of`** y matrices;
   - **ifpresent**: para concordancia de valores de campos facultativos (si no se omite).

En el cuadro 6 se indican los mecanismos de concordancia soportados y los símbolos asociados a ellos (en su caso) así como el ámbito de aplicación. En la columna de la izquierda se indican todos los tipos de TTCN-3 a los cuales se aplican estos mecanismos de concordancia. En el anexo B se describen en detalle los mecanismos de concordancia.

Cuadro 6/Z.140 – Mecanismos de concordancia de TTCN-3

Se utiliza con valores de	Valor		Valores de sustitución							Valores internos				Atributos
	Valor específico	OmitValue	Lista complementada	Lista de valores	AnyValue(?)	AnyValueOrNone(*)	Gama	Superconjuntos	Subconjuntos	AnyElement(?)	AnyElementsOrNone(*)	Permutación	LengthRestriction	IfPresent
boolean	Sí	Sí	Sí	Sí	Sí	Sí[a)]								Sí[b)]
integer	Sí	Sí	Sí	Sí	Sí	Sí[a)]	Sí							Sí[b)]
float	Sí	Sí	Sí	Sí	Sí	Sí[a)]	Sí							Sí[b)]
bitstring	Sí	Sí	Sí	Sí	Sí	Sí[a)]				Sí	Sí		Sí	Sí[b)]
octetstring	Sí	Sí	Sí	Sí	Sí	Sí[a)]				Sí	Sí		Sí	Sí[b)]
hexstring	Sí	Sí	Sí	Sí	Sí	Sí[a)]				Sí	Sí		Sí	Sí[b)]
character strings	Sí	Sí	Sí	Sí	Sí	Sí[a)]	Sí			Sí	Sí		Sí	Sí[b)]
record	Sí	Sí	Sí	Sí	Sí	Sí[a)]								Sí[b)]
record of	Sí	Sí	Sí	Sí	Sí	Sí[a)]				Sí	Sí	Sí	Sí	Sí[b)]
array	Sí	Sí	Sí	Sí	Sí	Sí[a)]				Sí	Sí		Sí	Sí[b)]
set	Sí	Sí	Sí	Sí	Sí	Sí[a)]								Sí[b)]
set of	Sí	Sí	Sí	Sí	Sí	Sí		Sí	Sí	Sí	Sí		Sí	Sí[b)]
enumerated	Sí	Sí	Sí	Sí	Sí	Sí[a)]								Sí[b)]
union	Sí	Sí	Sí	Sí	Sí	Sí[a)]								Sí[b)]
anytype	Sí	Sí	Sí	Sí	Sí	Sí[a)]								Sí[b)]

[a)] Si se emplea, se aplica únicamente a los campos facultativos de los tipos record y set (sin restricción en el tipo de ese campo).

[b)] Si se emplea, se aplica únicamente a los campos record y set (sin restricción en el tipo de ese campo).

14.3.1 Referencia a elementos de plantillas o de campos de plantilla

14.3.1.1 Referencia a elementos de una cadena particular

Se prohíbe hacer referencia a elementos de una cadena particular dentro de plantillas o campos de plantilla.

EJEMPLO:

```
var template charstring t_Char1 := 'MYCHAR';
var template charstring t_Char2;

t_Char2 := t_Char1[1];
// provocará un error en virtud de que la referencia a elementos
// particulares de la cadena está prohibida;
```

14.3.1.2 Referencia a campos record y set

Las plantillas y las variables de plantilla permiten hacer referencia a los subcampos dentro de una definición de plantilla mediante la notación de punto. No obstante, el campo referenciado puede ser un subcampo de un campo estructurado al que se ha asignado un mecanismo de concordancia. En esta cláusula se presentan las reglas correspondientes a esos casos.

- Omit, AnyValueOrNone, listas de valores y listas complementadas: si se hace referencia a un subcampo dentro de un campo estructurado al cual se ha asignado **omit**, AnyValueOrNone (*), una lista de valores o una lista complementada, se provocará un error.

EJEMPLO 1:

```
type record R1 {
    integer f1 optional,
    R2      f2 optional
}
type record R2 {
    integer g1,
    R2      g2 optional
}

:
var template R1 t_R1 := {
    f1 := 5,
    f2 := omit
}
var template R2 t_R2 := t_R1.f2.g2;
    // provoca un error debido a que se ha asignado omit a t_R1.f2
t_R1. f2 := *
t_R2 := t_R1.f2.g2;
    // provoca un error ya que se ha asignado * a t_R1.f2

t_R1 := ({f1:= omit, f2:={g1:=0, g2:= omit }},{f1:=5, f2:={g1:=1, g2:={g1:=2,
g2:= omit }}});

t_R2 := t_R1.f2;
t_R2 := t_R1.f2.g2;
t_R2 := t_R1.f2.g2.g2;
    // todas estas asignaciones provocan un error debido a que se ha asignado una lista
    // de valores a t_R1

t_R1 :=
    complement({f1:= omit, f2:={g1:=0, g2:= omit }},{f1:=5, f2:={g1:=1, g2:={g1:=2,
    g2:=omit }})

t_R2 := t_R1.f2;
t_R2 := t_R1.f2.g2;
t_R2 := t_R1.f2.g2.g2;
    // todas estas asignaciones provocan un error debido a que se ha asignado una lista
    // complementada a t_R1
```

- AnyValue: si se hace referencia a un subcampo dentro de un campo estructurado al cual se ha asignado AnyValue (?), en el lado derecho de una asignación, se devolverá AnyValue (?) a los subcampos obligatorios y AnyValueOrNone a los subcampos facultativos.

  Si se hace referencia a un subcampo dentro de un campo estructurado al cual se ha asignado AnyValue (?), en el lado izquierdo de una asignación, el campo estructurado tendrá que ser ampliado recurrentemente hasta alcanzar la profundidad del subcampo referenciado. Durante el proceso de ampliación se asignará AnyValue (?) a los subcampos obligatorios y AnyValueOrNone a los subcampos facultativos. Tras la ampliación, el valor o mecanismo de concordancia en el lado derecho de la asignación se asignará al subcampo referenciado.

EJEMPLO 2:

```
t_R1 := {f1:=0, f2:=?}
t_R2 := t_R1.f2.g2;
    // tras la asignación t_R2 será {g1:=?, g2:=*}
t_R1.f2.g2.g2 := ({g1:=1, g2:=omit},{g1:=2, g2:=omit});
    // en primer lugar el campo t_R1.f2 se ha ampliado hipotéticamente a
    // {g1:=?,g2:={g1:=?,g2:=*}}
    // por consiguiente, tras la asignación t_R1 será:
    // {f1:=0, f2:={g1:=?,g2:={g1:=?,g2:=({g1:=1, g2:=omit},{g1:=2, g2:=omit})}}}
```

- Atributo ifpresent: referenciar un subcampo dentro de un campo estructurado al cual se ha adjuntado el atributo **`ifpresent`** provocará un error (independientemente del valor o del mecanismo de concordancia al cual se ha adjuntado **`ifpresent`**).

14.3.1.3 Referencia a elementos record of y set of

Las plantillas y las variables de plantilla permiten hacer referencia a elementos de una plantilla o campo **`record of`** o **`set of`** mediante la notación de índice. No obstante, puede estar asignado un mecanismo de concordancia a la plantilla o campo dentro del cual está referenciado el elemento. Esta cláusula proporciona las reglas para tratar dichos casos.

- Omit, AnyValueOrNone, listas de valores, listas complementadas, subconjunto y superconjunto: referenciar un elemento dentro de un campo record of o set of al cual se ha asignado **`omit`**,

AnyValueOrNone (*) con o sin un atributo de longitud, una lista de valores, una lista complementada, un subconjunto o un superconjunto, provocará un error.

EJEMPLO 1:

```
type record of integer RoI;
type record of RoI RoRoI;

:
var template RoI t_RoI;
var template RoRoI t_RoRoI;
var template integer t_Int;
:
t_RoRoI := ({},{0},{0,0},{0,0,0});
t_RoI := t_RoRoI[0];
    // provocará un error debido a que se ha asignado una lista de valores a t_RoRoI;
```

- AnyValue: si se hace referencia a un elemento de una plantilla o campo **record of** o **set of** al cual se ha asignado AnyValue (?) (sin un atributo de longitud), en el lado derecho de una asignación, se devolverá AnyValue (?). Si se adjunta un atributo de longitud a AnyValue (?), el índice de la referencia no infringirá el atributo de longitud.

  Si se hace referencia a un elemento dentro de una plantilla o campo **record of** o **set of** al cual se ha asignado AnyValue (?) (sin un atributo de longitud), en el lado izquierdo de una asignación, el valor o mecanismo de concordancia en el lado derecho de la asignación se asignará al elemento referenciado, AnyElement(?) se asignará a todos los elementos antes del elemento referenciado (si los hubiere) y un solo AnyElementsOrNone(*) se añadirá al final. Si se adjunta un atributo de longitud a AnyValue(?), dicho atributo será conducido al nuevo campo o plantilla de forma transparente. El índice no infringirá las restricciones de tipo en ninguno de los casos anteriores.

EJEMPLO 2:

```
type record of integer RoI;
type record of RoI RoRoI;

:
var template RoI t_RoI;
var template RoRoI t_RoRoI;
var template integer t_Int;
:
t_RoI := ?;
t_Int := t_RoI[5];
    // tras la asignación t_Int será AnyValue(?);

t_RoRoI := ?;
t_RoI := t_RoRoI[5];
    // tras la asignación t_RoI será AnyValue(?);
t_Int := t_RoRoI[5].[3];
    // tras la asignación t_Int será AnyValue(?);

t_RoI := ? length (2..5);
t_Int := t_RoI[3];
    // tras la asignación t_Int será AnyValue(?);
t_Int := t_RoI[5];
    // provocará un error debido a que el índice referenciado queda fuera del atributo
    // de longitud (obsérvese que el índice 5 haría referencia al 6° elemento);

t_RoRoI[2] := {0,0};
    // tras la asignación t_RoRoI será {?,?,{0,0},*};
t_RoRoI[4] := {1,1};
    // tras la asignación t_RoRoI será {?,?,{0,0},?,{1,1},*};
t_RoI[0] := -5;
    // tras la asignación t_RoI será {-5,*} length (2..5);
t_RoI := ? length (2..5);
t_RoI[1] := 1;
    // tras la asignación t_RoI será {?,1,*} length(2..5);
t_RoI[3] := ?
    // tras la asignación t_RoI será {?,1,?,?,*}length(2..5);
t_RoI[5] := 5
    // tras la asignación t_RoI será {?,1,?,?,?,5,*}length(2..5);
    // obsérvese que t_RoI se convierte en un conjunto vacio aunque ello no provoca
    // un error;
```

- Permutación: si se hace referencia a un elemento de una plantilla o campo **record of** localizado dentro de una permutación (basándose en su índice), esto provocará un error. Los índices de elementos resguardados por una permutación serán determinados en función del número de elementos de

permutación. AnyValueOrNone como elemento de permutación provoca que la permutación proteja a todos los **record of** índices de los elementos.

EJEMPLO 3:

```
t_RoI := {permutation(0,1,3,?),2,?}
t_Int := t_RoI[5];
    // tras la asignación t_Int será AnyValue(?)

t_RoI := { permutation (0,1,3,?),2,*}
t_Int := t_RoI[5];
    // tras la asignación t_Int será * (AnyValueOrNone)
t_Int := t_RoI[2];
    // provoca un error debido a que el tercer elemento (con índice 2) se encuentra dentro
    // de la permutación

t_RoI := { permutation (0,1,3,*),2,?}
t_Int := t_RoI[5];
    // provoca un error debido a que la permutación contiene AnyValueOrNone(*)que puede
    // abarcar cualquier registro de índices
```

- Atributo Ifpresent: referenciar un elemento dentro de un campo **record of** o **set of** al cual se ha adjuntado el atributo **ifpresent** provocará un error (independientemente del valor o el mecanismo de concordancia al cual se adjuntó **ifpresent**).

14.4 Parametrización de plantillas

14.4.0 Consideraciones generales

Las plantillas para las operaciones de envío y recepción se pueden parametrizar. Los parámetros efectivos de una plantilla pueden incluir valores y plantillas, funciones y símbolos de concordancia especiales. Es obligatorio aplicar las reglas para listas de parámetros formales y efectivos, definidas en 5.2.

EJEMPLO:

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

// se podría utilizar así
pco1.send(MyTemplate(123));
```

14.5 En blanco

14.6 Plantillas modificadas

14.6.0 Consideraciones generales

Normalmente una plantilla especifica un conjunto de valores básicos o supletorios, o símbolos de concordancia para cada uno de los campos definidos en la definición de tipo o firma apropiada. Es posible especificar una plantilla modificada si sólo hay que hacer ligeras modificaciones. La plantilla modificada especifica, directa o indirectamente, las modificaciones de determinados campos de la plantilla original.

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

Las modificaciones se producen por vinculación, refiriendo posiblemente a la plantilla original. Si en la plantilla modificada se especifica un campo así como el valor o el símbolo de concordancia correspondientes, este valor o símbolo de concordancia sustituyen al especificado en la plantilla progenitora. Si en la plantilla modificada no se especifica un campo ni el valor o el símbolo de concordancia, se utilizará el valor o el símbolo de concordancia de la plantilla progenitora. Si el campo a modificar forma parte de la jerarquía de un campo de plantilla que también está estructurado, sólo se modificarán los campos señalados explícitamente y ningún otro del campo estructurado.

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

EJEMPLO 1:

```
// Dado
type record MyRecordType
{
    integer field,
    charstring field2,
    boolean field3
}

template MyRecordType MyTemplate1 :=
{
    field1 := 123,
    field2 := "A string",
    field3 := true
}
// esta instrucción
template MyRecordType MyTemplate2 modifies MyTemplate1 :=
{
    field1 := omit,         // field1 es facultativo, pero está en MyTemplate1
    field2 := "A modified string"
                            // field3 no se modifica
}
// es igual a la siguiente
template MyRecordType MyTemplate2 :=
{
    field1 := omit,
    field2 := "A modified string",
    field3 := true
}
```

Cuando se quieren cambiar los valores individuales de una plantilla modificada o de un campo de plantilla modificada de un tipo **`record of`**, y sólo en estos casos, también puede utilizarse la notación de asignación de valor, donde el lado izquierdo de la asignación representa el índice del elemento que ha de modificarse.

EJEMPLO 2:

```
template MyRecordOfType MyBaseTemplate := { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
template MyRecordOfType MyModifTemplate modifies MyBaseTemplate := { [2] := 3, [3] := 2 };
// MyModifTemplate debe concordar con la secuencia de valores { 0, 1, 3, 2, 4, 5, 6, 7,
// 8, 9 }
```

14.6.1 Parametrización de plantillas modificadas

Si una plantilla de base tiene una lista de parámetros formales, las siguientes reglas se aplican a todas las plantillas modificadas derivadas de esa plantilla de base sea en una sola o en varias operaciones de modificación:

a) la plantilla derivada no omitirá parámetros que se hayan definido en una de las operaciones de modificación entre la plantilla de base y esta plantilla considerada;

b) una plantilla derivada puede tener parámetros adicionales (anexados);

c) la lista de parámetros formales debe mantener el nombre de plantilla para cada 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 está parametrizado en Template1 y se mantiene parametrizado en Template2
    field2 := "A modified string",
}
```

14.6.2 Plantillas modificadas en línea

La notación TTCN-3 permite crear plantillas modificadas denominadas explícitamente y también definir plantillas modificadas en línea.

EJEMPLO:

```
// Dado
template MyMessageType Setup :=
{   field1 := 75,
    field2 := "abc",
    field3 := true
}

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

14.7 Modificación de campos de plantillas

En las operaciones de comunicación (por ejemplo, **send**, **receive**, **call**, **getcall**, etc.), los campos de plantillas sólo se pueden modificar mediante parametrización o mediante plantillas derivadas en línea. Los efectos de estos cambios en el valor del campo de plantilla no se mantienen en la plantilla después del evento de comunicación correspondiente.

Para fijar o extraer valores de plantillas en eventos de comunicación no se utilizará la notación de puntos *MyTemplateId.Fieldid* sino el símbolo "->" (véase la cláusula 23).

14.8 Operación Match

La operación **match** permite comparar el valor de una variable o parámetro con una plantilla. La operación devuelve un valor booleano. Si los tipos de la plantilla y la variable no son compatibles (véase 6.7), la respuesta de la operación es "falso". Si los tipos son compatibles, en la respuesta de la operación se indica si el valor de la variable es conforme a la plantilla especificada.

EJEMPLO:

```
template integer LessThan10 := (-infinity..9);

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

    if(match( RxValue, LessThan10)) { … }
// verdadero si el valor efectivo de Rxvalue es inferior a 10; si no es falso
     :
}
```

14.9 Operación Value of

La operación **valueof** permite asignar a los campos de una variable el valor especificado dentro de una plantilla. La variable y la plantilla han de ser de tipos compatibles (véase 6.7) y cada uno de los campos de la plantilla debe traducirse en un solo valor.

EJEMPLO:

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

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

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

15 Operadores

15.0 Consideraciones generales

La notación TTCN-3 soporta varios operadores predefinidos que se pueden utilizar en los términos de expresiones de TTCN-3. Hay siete categorías de operadores predefinidos:

a) operadores aritméticos;

b) operadores de cadena;

c) operadores relacionales;

d) operadores lógicos;

e) operadores para bits;

f) operadores de desplazamiento;

g) operadores de permutación.

Véase una lista de operadores en el cuadro 7.

Cuadro 7/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" lógico	**and**
	"O" lógico	**or**
	"O exclusivo" lógico	**xor**
Operadores de bits	"No" para bits	**not4b**
	"Y" para bits	**and4b**
	"O" para bits	**or4b**
	"O exclusivo" para bits	**xor4b**
Operadores de desplazamiento	desplazamiento izquierda	**<<**
	desplazamiento derecha	**>>**
Operadores de permutación	permutación izquierda	**<@**
	permutación derecha	**@>**

En el cuadro 8 se indica la precedencia de estos operadores. Todos los operadores de una fila tienen la misma precedencia. Si en una expresión aparece más de un operador de la misma precedencia, las operaciones son evaluadas de izquierda a derecha. Se pueden utilizar paréntesis para agrupar operandos en expresiones: una expresión entre paréntesis tienen la precedencia más alta para evaluación.

Cuadro 8/Z.140 – Precedencia de operadores

Prioridad	Tipo de operador	Operador
La más alta		**(…)**
	Unario	**+, -**
	Binario	***, /, mod, rem**
	Binario	**+, -, &**
	Unario	**not4b**
	Binario	**and4b**
	Binario	**xor4b**
	Binario	**or4b**
	Binario	**<<, >>, <@, @>**
	Binario	**<, >, <=, >=**
	Binario	**==, !=**
	Unario	**not**
	Binario	**and**
	Binario	**xor**
La más baja	Binario	**or**

15.1 Operadores aritméticos

Los operadores aritméticos representan las operaciones de adición, sustracción, multiplicación, división, módulo y residuo. Los operandos de estos operadores serán de tipo **integer** (incluidas derivaciones de **integer**) o **float** (incluidas derivaciones de **float**), pero en el caso de **mod** y **rem** sólo podrán utilizarse tipos **integer** (incluidas derivaciones de **integer**).

El resultado de las operaciones aritméticas con tipos **integer** es **integer**. El resultado de las operaciones aritméticas con tipos **float** es **float**.

Las reglas para operandos valen igualmente cuando se utiliza más (+) o menos (–) como operador unario. El resultado de utilizar el operador menos es el valor negativo de un operando positivo, y viceversa.

Resultado de la operación de división (/) entre dos valores:

a) si son valores **integer**, es el valor entero **integer** resultante de dividir el primer **integer** por el segundo (es decir, se descartan las fracciones);

b) si son valores **float**, es el valor **float** resultante de dividir el primer valor con coma decimal por el segundo (es decir, no se descartan las fracciones).

Los operadores **rem** y **mod** se aplican sobre operandos de tipo **integer** y tienen un resultado de tipo **integer**. Las operaciones `x rem y` y `x mod y` dan el residuo de la división de enteros de `x` por `y`. Por tanto, sólo se definen para operandos `y` distintos de cero. Cuando tanto `x` como `y` son positivos, las operaciones `x rem y` y `x mod y` dan el mismo resultado; el resultado es diferente si los argumentos son negativos.

Definición formal de los operadores **mod** y **rem**:

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

En el cuadro 9 se indica la diferencia entre los operadores **mod** y **rem**:

Cuadro 9/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 de cadena predefinidos concatenan valores de tipos de cadena compatibles. La operación es una simple concatenación de izquierda a derecha. No implica ninguna forma de adición aritmética. El resultado será del tipo raíz de los operandos.

EJEMPLO:

```
'1111'B & '0000'B & '1111'B es '111100001111'B
```

15.3 Operadores relacionales

Los operadores relacionales predefinidos representan las relaciones de igualdad (==), menor que (<), mayor que (>), no igualdad (!=) mayor que o igual (>=) y menor que o igual (<=). Los tipos de los operandos de igualdad y de no igualdad pueden ser arbitrarios, pero deben ser compatibles; ahora bien, en el caso del tipo **enumerated** los operandos han de ser ejemplares del mismo tipo. Todos los demás operadores relacionales sólo tendrán operandos de tipo **integer** (incluidas derivaciones de **integer**), **float** (incluidas derivaciones de **float**) o ejemplares del mismo tipo **enumerated**. El resultado de estas operaciones es de tipo **boolean**.

Dos valores **charstring** o **universal charstring** son iguales únicamente si tienen la misma longitud y los mismos caracteres en todas las posiciones. La misma regla de igualdad se aplica a los valores de tipos **bitstring**, **hexstring** u **octetstring**, excepto que las fracciones que han de coincidir en todas las posiciones son, respectivamente, bits, cifras hexadecimales o pares de cifras hexadecimales.

Dos valores **record**, **set**, **record of** o **set of** son iguales únicamente si las estructuras de valores efectivos son compatibles (véase 6.7) y los valores de todos los campos correspondientes son iguales. Los valores **record** también pueden compararse con valores **record of**, y los valores set con valores set of, aplicando la misma regla que se aplica para comparar dos valores **record** o **set**.

NOTA – En el caso de "Todos los campos", los campos facultativos que no se han incluido en el valor efectivo de un tipo **record** serán considerados como valores no definidos. Si se compara con un valor de otro tipo **record** sólo puede haber igualdad con un campo facultativo que falta (considerado también como un valor no definido); si se compara con un valor de tipo **record** of sólo puede haber igualdad con un elemento de valor no definido. El mismo principio vale para la comparación de valores de dos tipos **set**, o un tipo **set** y un tipo **set of**.

Dos valores de tipo **union** son iguales únicamente si los tipos de los campos elegidos en los dos valores son compatibles, y los valores efectivos de los campos elegidos son iguales.

EJEMPLO:

```
// Dado
type      set   SetA{
          integer     a1    optional,
          integer     a2    optional,
          integer     a3    optional
          };

type      set   SetB{
          integer     b1    optional,
          integer     b2    optional,
          integer     b3    optional
          };

type      set   SetC{
          integer     c1    optional,
          integer     c2    optional,
          };

type      set of integer    SetOf;

type      union UniD{
          integer     d1,
          integer     d2,
          };

type      union UniE{
          integer     e1,
          integer     e2,
          };
```

```
type      union UniF{
          integer     f1,
          integer     f2,
          boolean     f3,
          };

// Y
const     SetA  conSetA1    :=    { a1 := 0, a2:= omit, a3 := 2 };
          // Obsérvese que el orden de definición de valores de los campos
          // es indiferente
const     SetB  conSetB1    :=    { b1 := 0, b3 := 2, b2:= omit };
const     SetB  conSetB2    :=    { b2 := 0, b3 := 2, b1:= omit };
const     SetC  conSetC1    :=    { c1 := 0, c2 :=2 };
const     SetOf conSetOf1   :=    { 0, omit, 2 };
const     SetOf conSetOf2   :=    { 0, 2 };
const     UniD  conUniD1    :=    { d1:= 0 };
const     UniE  conUniE1    :=    { e1:= 0 };
const     UniE  conUniE2;   :=    { e2:= 0 };
const     UniF  conUniF1;   :=    { f1:= 0 };

// Entonces
conSetA1 == conSetB1;
    // devuelve true
conSetA1 == conSetB2;
// devuelve false porque no hay igualdad de a1 ni a2 con la contraparte
// (y no se omite el elemento correspondiente)
conSetA1 == conSetC1;
    // devuelve false porque las estructuras del valor efectivo de SetA
    // y SetC no son compatibles
conSetA1 == conSetOf1;
// devuelve true
conSetA1 == conSetOf2;
// devuelve false porque la contraparte de a2 (omitido) es 2,
// pero la contraparte de a3 no está definida
conSetC1 == conSetOf2;
// devuelve true
conUniD1 == conUniE1;
// devuelve true
conUniD1 == conUniE2;
    // devuelve false porque el campo elegido e2 no es la contraparte
    // del campo d1 de UniD1
conUniD1 == conUniF1;
    // devuelve false porque las estructuras del valor efectivo de UniD1
    // y UniF no son compatibles
```

15.4 Operadores lógicos

Los operadores **boolean** predefinidos ejecutan las operaciones de negación, **and** lógico, **or** lógico y **xor** lógico. Los operandos han de ser de tipo **boolean**. El resultado de las operaciones lógicas es de tipo **boolean**.

La respuesta del operador **not** lógico unario será el valor **true** si el valor del operando es **false**, y la respuesta será el valor **false** si el valor del operando es **true**.

La respuesta del operador **and** lógico será el valor **true** si los dos operandos son **true**; en los demás casos será el valor **false**.

La respuesta del operador **or** lógico será el valor **true** si por lo menos uno de sus operandos es **true**; será el valor **false** únicamente cuando ambos operandos son **false**.

La respuesta del operador **xor** lógico será el valor **true** si uno de sus operandos es **true**; será el valor **false** si ambos operandos son **false** o ambos operandos son **true**.

Para las expresiones booleanas se emplea la evaluación en cortocircuito, es decir, la evaluación de operandos de los operadores lógicos se detiene cuando se conoce el resultado total: en el caso del operador **and**, si el argumento izquierdo se traduce a **false**, el argumento derecho no se evalúa y toda la expresión se traduce a **false**. En el caso del operador **or**, si el argumento izquierdo se traduce a **true**, el argumento derecho no se evalúa y toda la expresión se traduce a true.

15.5 Operadores para bits

Los operadores para bits predefinidos ejecutan las operaciones **not**, **and**, **or** y **xor**. Las denominaciones de estos operadores para bits son, respectivamente **not4b**, **and4b**, **or4b** y **xor4b**.

NOTA – Estas denominaciones significan, al pronunciarlas en inglés: "no para bits", "y para bits", etc.

Los operandos han de ser de tipo **bitstring**, **hexstring** u **octetstring** y han de ser de tipos compatibles en el caso de **and4b**, **or4b** y **xor4b**. El resultado de los operadores para bits será del mismo tipo raíz que los operandos.

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

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

EJEMPLO 1:

```
el resultado de not4b  '1010'B es  '0101'B
el resultado de not4b  '1A5'H es  'E5A'H
el resultado de not4b  '01A5'O es 'FE5A'O
```

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

```
el resultado de '1'B and4b '1'B es  '1'B
el resultado de '1'B and4b '0'B es  '0'B
el resultado de '0'B and4b '1'B es  '0'B
el resultado de '0'B and4b '0'B es  '0'B
```

EJEMPLO 2:

```
el resultado de '1001'B and4b '0101'B es '0001'B
el resultado de 'B'H and4b '5'H es '1'H
el resultado de 'FB'O and4b '15'O es '11'O
```

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

```
el resultado de '1'B or4b '1'B es  '1'B
el resultado de '1'B or4b '0'B es  '1'B
el resultado de '0'B or4b '1'B es  '1'B
el resultado de '0'B or4b '0'B es  '0'B
```

EJEMPLO 3:

```
el resultado de '1001'B or4b '0101'B es '1101'B
el resultado de '9'H or4b '5'H es 'D'H
el resultado de 'A9'O or4b 'F5'O es 'FD'O
```

El operador para bits **xor4b** acepta dos operandos de la misma longitud. 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 del bit resultante es 1. Es decir:

```
el resultado de '1'B xor4b '1'B es  '0'B
el resultado de '0'B xor4b '0'B es  '0'B
el resultado de '0'B xor4b '1'B es  '1'B
el resultado de '1'B xor4b '0'B es  '1'B
```

EJEMPLO 4:

```
el resultado de '1001'B xor4b '0101'B es '1100'B
el resultado de '9'H xor4b '5'H es 'C'H
el resultado de '39'O xor4b '15'O es '2C'O
```

15.6 Operadores de desplazamiento

Los operadores de desplazamiento predefinidos ejecutan las operaciones de desplazamiento a la izquierda (**<<**) y desplazamiento a la derecha (**>>**). El operando a la izquierda ha de ser de tipo **bitstring**, **hexstring** u **octetstring**. El operando a la derecha a de ser de tipo **integer**. El resultado de estos operadores será del mismo tipo que el operando a la izquierda.

El comportamiento de los operadores de desplazamiento dependerá del tipo de operando a la izquierda. Si este operando es de tipo:

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

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

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

El operador de desplazamiento a la izquierda (**<<**) acepta dos operandos. Desplaza a la izquierda el operando de la izquierda en un número de unidades especificado por el operando de la derecha. Se descartan las unidades de desplazamiento en exceso (bits, dígitos hexadecimales u octetos). Por cada unidad de desplazamiento a la izquierda se inserta un cero ('0'B, '0'H, ó '00'O, según el tipo del operando de la izquierda) del lado derecho del operando de la izquierda.

EJEMPLO 1:

```
el resultado de '111001'B  << 2 es '100100'B
el resultado de '12345'H  << 2 es '34500'H
el resultado de '1122334455'O  << (1+1) es '3344550000'O
```

El operador de desplazamiento a la derecha **(>>)** acepta dos operandos. Desplaza a la derecha el operando de la izquierda en un número de unidades especificado por el operando de la derecha. Se descartan las unidades de desplazamiento en exceso (bits, dígitos hexadecimales u octetos). Por cada unidad de desplazamiento a la derecha se inserta un 0 ('0'B, '0'H, ó '00'O según el tipo del operando de la izquierda) del lado izquierdo del operando de la izquierda.

EJEMPLO 2:

```
el resultado de '111001'B  >> 2 es '001110'B
el resultado de '12345'H  >> 2 es '00123'H
el resultado de '1122334455'O  >> (1+1) es '0000112233'O
```

15.7 Operadores de permutación

Los operadores de permutación predefinidos ejecutan las operaciones de permutación a la izquierda (**<@**) y permutación a la derecha (**@>**). El operando de la izquierda ha de ser de tipo **bitstring**, **hexstring**, **octetstring**, **charstring** o **universal charstring**. El operando de la derecha será de tipo **integer**. El resultado de estos operadores será del mismo tipo que el operando de la izquierda.

El comportamiento de los operadores de permutación depende del tipo del operando de la izquierda. Si el tipo del operando de la izquierda es:

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

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

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

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

El operador de permutación izquierda (**<@**) acepta dos operandos. Permuta a la izquierda el operando de la izquierda en el número de unidades de desplazamiento especificado por el operando de la derecha. Las unidades de desplazamiento en exceso (bits, dígitos hexadecimales, octetos o caracteres) se reinsertan a la derecha en el operando de la izquierda.

EJEMPLO 1:

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

El operador de permutación derecha (**@>**) acepta dos operandos. Permuta a la derecha el operando de la izquierda en el número de unidades de desplazamiento especificado por el operando de la derecha. Las unidades de desplazamiento en exceso (bits, cifras hexadecimales, octetos o caracteres) se reinsertan a la izquierda en el operando de la izquierda.

EJEMPLO 2:

```
el resultado de '100001'B  @> 2 es '011000'B
el resultado de '12345'H  @> 2 es '45123'H
el resultado de '1122334455'O  @> (1+2) es '3344551122'O
el resultado de "abcdefg" @> 3 es "efgabcd"
```

16 Funciones y alternativas (altsteps)

En la notación TTCN-3 se utilizan funciones y alternativas (altsteps) para especificar y estructurar un comportamiento de pruebas, definir el comportamiento por defecto, estructurar cálculos en un módulo, etc. como se indica en las siguientes cláusulas.

16.1 Funciones

16.1.0 Consideraciones generales

En TTCN-3 las funciones se utilizan para expresar comportamientos de prueba, organizar la ejecución de la prueba o 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 o una plantilla. La devolución de un valor se indica mediante la palabra clave **`return`** seguida por un identificador de tipo. La devolución de una plantilla se señala mediante las palabras de código **`return template`** seguidas por un identificador de tipo.

Cuando se utiliza la palabra clave **`return`** en el cuerpo de una función que tiene definido en el encabezamiento el valor que ha de devolver, después de la misma hay que indicar siempre una expresión descriptiva del valor de respuesta. La función debe devolver un tipo compatible con el tipo especificado (**`return`**). Cuando se emplea la palabra clave **`return`** en el cuerpo de la función que tiene definida en su encabezamiento la devolución de una plantilla, hay que indicar siempre un ejemplar de expresión o de plantilla que represente la plantilla de respuesta, cuyo tipo debe ser compatible con el tipo de plantilla de respuesta.

La instrucción return en el cuerpo de la función hace que la función devuelva, al terminar, un valor en respuesta a la posición que la ha invocado.

EJEMPLO 1:

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

  return 7;     // devuelve el valor entero 7 cuando termina la función
}
// Definición de las funciones que pueden devolver símbolos o plantillas
// concordantes
function MyFunction2() return template integer
{
:
  return ?;     // devuelve el mecanismo de concordancia AnyValue
}
function MyFunction3() return template octetstring
{
:
  return "FF??FF"O;  // devuelve una cadena de octetos (octetstring)
                     //con AnyElement incorporado
}
```

Es posible definir una función dentro de un módulo o declarar que se ha definido externamente (es decir, **`external`**). Para una función externa sólo hay que proporcionar la interfaz correspondiente en el módulo TTCN-3. La realización de la función externa está fuera del ámbito de la presente Recomendación. Las funciones externas no pueden contener operaciones de puertos. Está prohibido el empleo de las funciones externas para devolver plantillas.

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

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

En un módulo, el comportamiento de una función puede definirse utilizando las instrucciones de programa y las operaciones descritas en la cláusula 18. Si una función utiliza variables, constantes, temporizadores y puertos que se declaran en una definición del tipo de componente, habrá que referenciar este tipo de componente utilizando las palabras clave **`runs on`** en el encabezamiento de la función. La única excepción a esta regla es el caso en que toda la información necesaria del componente se transfiere dentro de la función como parámetros.

EJEMPLO 2:

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

Las funciones que no tengan la cláusula **`runs on`** no podrán invocar en ningún caso una función o una alternativa (altstep), ni activar una alternativa (altstep) supletoria con una cláusula **`runs on`** localmente.

Todas las funciones que se lanzan mediante la operación de activación del componente de prueba (**start**) han de tener una cláusula **runs on** (véase 22.5), y se considera que han sido invocadas en el componente que se activa (no localmente). Ahora bien, es posible invocar la operación de activación de componente de prueba (**start**) en funciones que no tienen la cláusula **runs on**.

NOTA – Las restricciones relativas a la cláusula **runs on** valen sólo para funciones y alternativas (altsteps), no para casos de prueba.

Las funciones que se utilicen en la parte de control de un módulo TTCN-3 no podrán tener la cláusula **runs on**, pero sí pueden ejecutar casos de prueba.

16.1.1 Parameterización de funciones

Las funciones se pueden parametrizar. Es necesario aplicar las reglas definidas en 5.2 para las listas de parámetros formales.

EJEMPLO:

```
function MyFunction2(inout integer MyPar1) {
                                        // MyFunction2 no devuelve ningún valor
   MyPar1 := 10 * MyPar1;               // pero sí modifica el valor de MyPar1
}                                       // que se transfiere por referencia
```

16.1.2 Invocación de funciones

Para invocar una función se hace referencia a su nombre y se proporciona la lista de parámetros efectivos. Hay que invocar directamente las funciones que no devuelven valores. Las funciones que devuelven valores se pueden invocar directamente o dentro de expresiones. Se aplicarán las reglas definidas en 5.2 para las listas de parámetros efectivos.

EJEMPLO:

```
MyVar := MyFunction4();     // El valor que devuelve MyFunction4 se asigna a MyVar.
                            // Este valor y MyVar tienen que ser compatibles

MyFunction2(MyVar2);        // MyFunction2 no devuelve ningún valor y se solicita
                            // con el parámetro efectivo MyVar2 que se puede transferir
                            // por referencia

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

Hay restricciones especiales para las funciones que se vinculan a un componente de prueba mediante la operación de activación del componente (**start**). Estas restricciones se describen en 22.5.

16.1.3 Funciones predefinidas

La notación TTCN-3 contiene varias funciones predefinidas (incorporadas) que no tienen que ser declaradas antes de su utilización.

Cuadro 10/Z.140 – Lista de las funciones predefinidas de TTCN-3

Categoría	Función	Palabra clave
Funciones de conversión	Convertir valor **integer** en valor **charstring**	**int2char**
	Convertir valor **integer** en valor **universal charstring**	**int2unichar**
	Convertir valor **integer** en valor **bitstring**	**int2bit**
	Convertir valor **integer** en valor **hexstring**	**int2hex**
	Convertir valor **integer** en valor **octetstring**	**int2oct**
	Convertir valor **integer** en valor **charstring**	**int2str**
	Convertir valor **integer** en valor **float**	**int2float**
	Convertir valor **float** en valor **integer**	**float2int**
	Convertir valor **charstring** en valor **integer**	**char2int**
	Convertir valor **charstring** en valor **octetstring**	**char2oct**
	Convertir valor **universal charstring** en valor **integer**	**unichar2int**
	Convertir valor **bitstring** en valor **integer**	**bit2int**
	Convert valor **bitstring** en valor **hexstring**	**bit2hex**
	Convertir valor **bitstring** en valor **octetstring**	**bit2oct**
	Convertir valor **bitstring** en valor **charstring**	**bit2str**
	Convertir valor **hexstring** en valor **integer**	**hex2int**
	Convertir valor **hexstring** en valor **bitstring**	**hex2bit**
	Convertir valor **hexstring** en valor **octetstring**	**hex2oct**
	Convertir valor **hexstring** en valor **charstring**	**hex2str**
	Convertir valor **octetstring** en valor **integer**	**oct2int**
	Convertir valor **octetstring** en valor **bitstring**	**oct2bit**
	Convertir valor **octetstring** en valor **hexstring**	**oct2hex**
	Convertir valor **octetstring** en valor **charstring**	**oct2str**
	Convertir valor **octetstring** en valor **charstring**	**oct2char**
	Convertir valor **charstring** en valor **integer**	**str2int**
	Convertir valor **charstring** en valor **octetstring**	**str2oct**
	Convertir valor **charstring** en valor **float**	**str2float**
Funciones de longitud/tamaño	Devolver la longitud de un valor de cualquier tipo de cadena	**lengthof**
	Devolver el número de elementos de un **record**, **record of**, **template**, **set**, **set of** o **array**	**sizeof**
	Devolver el número de elementos en un tipo estructurado	**sizeoftype**
Funciones de presencia/opción	Determinar si está presente un campo facultativo en un **record**, **record of**, **template**, **set** o **set of**	**ispresent**
	Determinar cuál es la opción de un tipo **union**	**ischosen**
Funciones de cadenas	Devolver una parte de la cadena de entrada que concuerda con la descripción de plantilla especificada	**regexp**
	Devolver la porción especificada de la cadena de entrada	**substr**
	Sustituir una subcadena de una cadena o insertar la cadena de entrada en una cadena	**replace**
Otras funciones	Crear un número de coma decimal (float) aleatorio	**rnd**

Cuando se invoca una función predefinida:

1) el número de parámetros efectivos ha de ser igual al número de parámetros formales; y
2) cada parámetro efectivo tendrá que traducirse en un elemento del tipo del parámetro formal correspondiente; y
3) todas las variables que aparecen en la lista de parámetros estarán vinculadas.

Véase la descripción completa de las funciones predefinidas en el anexo C.

16.1.4 Restricciones para las funciones solicitadas de sitios específicos

Las funciones que devuelven valores pueden ser solicitadas durante las operaciones de comunicación (en plantilla, campos de plantilla o plantillas en línea) o durante la evaluación del estado puntual (en guardas booleanas de instrucciones alt o altsteps (véase 20.1.1) y en el momento de inicialización de las definiciones locales de altsteps (véase 16.2.2)). A fin de evitar efectos secundarios que provoquen el cambio de estado del componente o del estado puntual efectivo e impedir resultados diferentes de evaluaciones subsiguientes de un estado puntual no modificado, no deben utilizarse las siguientes operaciones en las funciones solicitadas en los casos antes especificados:

- Operaciones componente, es decir, **create**, **start** (componente), **stop** (componente), **kill**, **running** (componente), **alive**, **done** y **killed** (véanse las notas 1, 3, 4 y 6).
- Operaciones port, es decir, **start** (puerto), **stop** (puerto), **halt**, **clear**, **send**, **receive**, **trigger**, **call**, **getcall**, **reply**, **getreply**, **raise**, **catch**, **check**, **connect**, **map** (véanse las notas 1, 2, 3 y 6).
- La operación **action** (véanse las notas 2 y 6).
- Operaciones timer, es decir, **start** (temporizador), **stop** (temporizador), **running** (temporizador), **read**, **timeout** (véanse las notas 4 y 6).
- Solicitud de funciones externas (véanse las notas 4 y 6).
- Solicitud de la función predefinida **rnd** (véanse las notas 4 y 6).
- Modificación de variable de componente, es decir, utilización de variables de componente en el lado derecho de las asignaciones y en la creación de ejemplares de los parámetros **out** e **inout** (véanse las notas 4 y 6).
- Solicitud de la operación **setverdict** (véanse las notas 4 y 6).
- Activación y desactivación de opciones por defecto, es decir, las instrucciones **activate** y **deactivate** (véanse las notas 5 y 6).
- Solicitud de funciones con los parámetros **out** o **inout** (véanse las notas 7 y 8).

NOTA 1 – La ejecución de las operaciones **start**, **stop**, **done**, **killed**, **halt**, **clear**, **receive**, **trigger**, **getcall**, **getreply**, **catch** y **check** puede provocar cambios del estado puntual efectivo.

NOTA 2 – Las operaciones **send**, **call**, **reply**, **raise** y **action** habrán de evitarse para fines de legibilidad, es decir, todas las comunicaciones deben llevarse a cabo explícitamente y no como efecto secundario de otra operación de comunicación o de la evaluación de un estado puntual.

NOTA 3 – Las operaciones **map**, **unmap**, **connect**, **disconnect**, **create** habrán de evitarse para fines de legibilidad, es decir, todas las operaciones de configuración deben llevarse a cabo explícitamente y no como efecto secundario de otra operación de comunicación o de la evaluación de un estado puntual.

NOTA 4 – La solicitud de las funciones externas, **rnd**, **running**, **alive**, **read**, **setverdict**, y la escritura a variables de componente deberán evitarse porque pueden conducir a resultados diferentes de evaluaciones subsiguientes del mismo estado puntual, por ejemplo, imposibilitando la detección de situaciones de bloqueo.

NOTA 5 – Las operaciones **activate** y **deactivate** habrán de evitarse porque modifican el conjunto de opciones por defecto que se tiene en cuenta durante la evaluación del estado puntual efectivo.

NOTA 6 – Las restricciones salvo la limitación del uso de la parametrización de **out** o **inout** se aplicarán recurrentemente, es decir, se prohíbe utilizarlas directamente o a través de una cadena larga y arbitraria de solicitudes de funciones.

NOTA 7 – La restricción de funciones de solicitud con los parámetros **out** o **inout** no se aplica recurrentemente, es decir, son válidas las funciones de solicitud que por sí mismas invocan funciones con los parámetros **out** o **inout**.

NOTA 8 – La utilización de los parámetros **out** o **inout** habrá de evitarse porque puede conducir también a resultados distintos de evaluaciones subsiguientes del mismo estado puntual.

16.2 Alternativas (Altsteps)

16.2.0 Consideraciones generales

La notación TTCN-3 utiliza alternativas (altsteps) para especificar el comportamiento por defecto o estructurar las alternativas de una instrucción **alt**. Las alternativas (altsteps) son unidades ámbito similares a las funciones. En el cuerpo del altstep aparece un conjunto facultativo de definiciones locales y un conjunto de alternativas (*alternativas principales*) que constituye el cuerpo propiamente dicho de altstep. Para estas alternativas principales se aplican las mismas reglas de sintaxis de las alternativas de instrucciones **alt**.

El comportamiento de un altstep puede describirse utilizando las instrucciones y las operaciones de programa que se indican en la cláusula 18. Si un altstep incluye operaciones de puertos o utiliza variables, constantes o temporizadores de componentes, es necesario referenciar el tipo de componente asociado utilizando las palabras clave **runs on** en el encabezamiento del altstep. La única excepción a esta regla es el caso en que todos los puertos, variables, constantes y temporizadores utilizados en el altstep se transfieren como parámetros.

EJEMPLO:

```
// Dado
type component MyComponentType {
    var integer MyIntVar := 0;
    timer MyTimer;
    port MyPortTypeOne PCO1, PCO2;
    port MyPortTypeTwo PCO3;
}
```

```
// Puede definirse así un altstep que utiliza PCO1, PCO2, MyIntVar y MyTimer,
// y es de tipo MyComponentType

altstep AltSet_A(in integer MyPar1) runs on MyComponentType {
    [] PCO1.receive(MyTemplate(MyPar1, MyIntVar) {
       setverdict(inconc);
       }
    [] PCO2.receive {
          repeat
       }
    [] MyTimer.timeout {
       setverdict(fail);
          stop
       }
}
```

Un altstep puede invocar funciones y otros altsteps, o activar altsteps supletorios. Un altstep que no tenga la cláusula **runs on** no podrá invocar en ningún caso una función o un altstep, ni activar un altstep supletorio con una cláusula **runs on** localmente.

16.2.1 Parametrización de altsteps

Es posible parametrizar los altsteps. Un altstep que esté activado como opción por defecto sólo podrá tener parámetros **in**, parámetros **port** y parámetros **time**r. Un altstep invocado sólo como alternativa en una instrucción **alt** o por instrucción autónoma en una descripción de comportamiento TTCN-3 puede tener parámetros **in**, **out** e **inout**. Son de aplicación las reglas para listas de parámetros formales definidas en 5.2.

16.2.2 Definiciones locales en altsteps

16.2.2.0 Consideraciones generales

Los altsteps pueden definir localmente constantes, variables y temporizadores. Hay que hacer las definiciones locales antes del conjunto de alternativas.

EJEMPLO:

```
altstep AnotherAltStep(in integer MyPar1) runs on MyComponentType {
    var integer MyLocalVar := MyFunction();  // variable local
    const float MyFloat := 3.41;             // constante local
    [] PCO1.receive(MyTemplate(MyPar1, MyLocalVar) {
       setverdict(inconc);
       }
    [] PCO2.receive {
       repeat
       }
}
```

16.2.2.1 Restricciones para la inicialización de definiciones locales en altsteps

Se señala la posibilidad de efectos secundarios de una inicialización de definiciones locales mediante una solicitud de funciones que devuelven valores. Para evitar estos efectos secundarios que son una causa de incoherencia entre el estado puntual (*snapshot*) y el estado del componente e impedir resultados distintos de evaluaciones subsiguientes de un estado puntual no modificado, habrán de aplicarse las restricciones establecidas en 16.1.4 para la inicialización de las definiciones locales.

16.2.3 Invocación de altsteps

En todos los casos hay una relación entre la invocación de un altstep y una instrucción **alt**. Puede invocarse implícitamente por el mecanismo por defecto (véase la cláusula 21) o explícitamente mediante una solicitud directa dentro de una instrucción **alt** (véase 20.1.6). La invocación de un altstep no produce otro estado puntual; para evaluar las alternativas principales de un altstep se utiliza el estado puntual efectivo de la instrucción **alt** en la que se ha invocado el altstep.

NOTA – Naturalmente, sí se producirá otra descripción de estado puntual dentro de un altstep si se ha especificado e introducido otra instrucción **alt** dentro de una alternativa principal seleccionada.

Para invocar implícitamente un altstep mediante el mecanismo por defecto hay que activar el altstep como opción por defecto mediante una instrucción **activate** antes de la invocación.

EJEMPLO 1:

```
 :
var default MyDefVarTwo := activate(MySecondAltStep());     // Activación de un altstep
                                                            // como opción por defecto
 :
```

Una solicitud explícita de un altstep en una instrucción **alt** puede interpretarse como una alternativa de solicitud de función.

EJEMPLO 2:

```
 :
alt {
    [] PCO3.receive {
           …
         }
    [] AnotherAltStep();    // solicitud explícita del altstep AnotherAltStep como
                            // alternativa de una instrucción alt
    [] MyTimer.timeout {}
}
```

Cuando se solicita explícitamente un altstep dentro de una instrucción **alt**, la siguiente alternativa a comprobar será la primera alternativa del **altstep**. Los procesos de comprobación y ejecución son los mismos para alternativas de un **altstep** y alternativas de una instrucción **alt** (véase 20.1), excepto que no se produce otra descripción de estado puntual para el **altstep**. Si el altstep termina con resultado negativo (es decir, se han comprobado todas las alternativas principales del **altstep** y no hay ramales concordantes), se procede a evaluar la siguiente alternativa o se invoca el mecanismo de opciones por defecto (si la solicitud explícita es la última alternativa de la instrucción **alt**). Si termina con resultado positivo, se terminará el componente de prueba, es decir, el **altstep** termina con una instrucción **stop**, se producirá un nuevo estado puntual y otra evaluación de la instrucción **alt**, es decir, el altstep termina con **repeat** (véase 20.2) o se continuará inmediatamente después de la instrucción **alt**, es decir, la alternativa principal seleccionada del altstep termina sin un **repeat** o **stop** explícito.

También es posible solicitar un **altstep** mediante una instrucción autónoma en una descripción de comportamiento de TTCN-3. En este caso, la solicitud de **altstep** puede interpretarse como una expresión simplificada de una instrucción **alt** con una sola alternativa que describe la solicitud explícita del **altstep**.

EJEMPLO 3:

```
// La instrucción
AnotherAltStep();    // Supóngase que AnotherAltStep es un altstep definido correctamente

//es una expresión simplificada de

alt {
    [] AnotherAltStep();
}
```

16.3 Funciones y altsteps para distintos tipos component

Véase 6.7.3.

17 Casos de prueba

17.0 Consideraciones generales

Los casos de prueba son una clase de función especial. La instrucción **execute** en la parte de control del módulo permite lanzar los casos de prueba (véase 27.1). El resultado de un caso de prueba ejecutado es siempre un valor de tipo **verdicttype**. Un caso de prueba no puede contener sino un MTC del tipo indicado por referencia en el encabezamiento de la definición de este caso de prueba. El comportamiento definido en el cuerpo del caso de prueba es el comportamiento del MTC.

Al invocar un caso de prueba se crea el MTC, se crean ejemplares de los puertos del MTC y la interfaz del sistema de prueba, y se activa en el MTC el comportamiento especificado en la definición del caso de prueba. Todas estas acciones serán ejecutadas implícitamente, es decir, sin operaciones **create** y **start** explícitas.

El encabezamiento del caso de prueba se ha dividido en dos partes con el fin de proporcionar la información necesaria para que estas operaciones implícitas puedan ocurrir:

a) la parte de interfaz (obligatoria): se indica mediante la palabra clave **runs on**, contiene una referencia al tipo de componente impuesto 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): se indica mediante la palabra clave **system** y contiene una referencia al tipo de componente que define los puertos necesarios para la interfaz del sistema de prueba. La parte de sistema de prueba se omitirá únicamente cuando sólo se crea un ejemplar del MTC durante la ejecución de la prueba. En este caso, el tipo del 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 la interfaz de sistema
                            // de prueba para el MTC
{
    :     // Este comportamiento se ejecuta en el MTC cuando se invoca el caso de prueba
}

// o, un caso de prueba donde sólo se define un ejemplar del MTC
testcase MyTestCaseTwo() runs on MyMtcType2
{
    :     // Este comportamiento se ejecuta en el MTC cuando se invoca el caso de prueba
}
```

17.1 Parametrización de casos de prueba

Los casos de prueba se pueden parametrizar. Es necesario aplicar las reglas definidas en 5.2 para las listas de parámetros formales.

18 Instrucciones de programa y operaciones

Los elementos de programa fundamentales de casos de prueba, funciones, altsteps y la parte de control de los módulos TTCN-3 son expresiones, las instrucciones de programa (asignaciones, construcciones de bucle, etc.), las instrucciones de comportamiento tales como comportamiento secuencial, comportamiento alternativo, entrelazado, comportamiento por defecto, etc. y las operaciones tales como **send**, **receive**, **create**, etc.

Hay instrucciones simples (que no incluyen otras instrucciones de programa) e instrucciones compuestas (que pueden incluir otras instrucciones, bloques de instrucciones y declaraciones).

Las instrucciones se ejecutarán siguiendo el orden de introducción: conforme al proceso secuencial que se ilustra en la figura 8.

Figura 8/Z.140 – Ilustración del proceso secuencial

Todas las instrucciones de la secuencia estarán separadas por el delimitador ";".

EJEMPLO:

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

En las instrucciones compuestas puede haber un bloque vacío de instrucciones y declaraciones, es decir, {}, por ejemplo un ramal de una instrucción **alt**, que significa que no se realiza ninguna acción.

Cuadro 11/Z.140 – Las instrucciones y las operaciones de TTCN-3

Instrucción	Palabra clave o símbolo asociados	Se puede utilizar en control de módulo	Se puede utilizar en funciones, casos de prueba y altsteps	No se puede utilizar en funciones desde plantillas, guardias booleanas o desde la inicialización de definiciones locales alstep
Expresiones	`(…)`	Sí	Sí	Sí
Instrucciones de programas básicas				
Asignaciones	`:=`	Sí	Sí	Sí (véase la nota 3)
Registros	`log`	Sí	Sí	Sí
Etiquetas y Goto	`label / goto`	Sí	Sí	Sí
Condiciones (If-else)	`if (…) {…} else {…}`	Sí	Sí	Sí
Bucle "For"	`for (…) {…}`	Sí	Sí	Sí
Bucle "While"	`while (…) {…}`	Sí	Sí	Sí
Bucle "Do while"	`do {…} while (…)`	Sí	Sí	Sí
Detener ejecución	`stop`	Sí	Sí	
Seleccionar caso	`select case (…) { case (…) {…} case else {…}}`	Sí	Sí	Sí
Instrucciones de programa para comportamiento				
Comportamiento alternativo	`alt {…}`	Sí (véase la nota 1)	Sí	
Reevaluación de comportamiento alternativo	`repeat`	Sí (véase la nota 1)	Sí	
Comportamiento entrelazado	`interleave {…}`	Sí (véase la nota 1)	Sí	
Devolver control	`return`		Sí (véase la nota 4)	Sí
Instrucciones de tratamiento por defecto				
Activar una opción por defecto	`activate`	Sí (véase la nota 1)	Sí	
Desactivar una opción por defecto	`deactivate`	Sí (véase la nota 1)	Sí	
Operaciones de configuración				
Crear componente de prueba paralelo	`create`		Sí	
Conectar puerto de componente a puerto de componente	`connect`		Sí	
Desconectar dos puertos de componente	`disconnect`		Sí	
Relacionar puerto con interfaz de prueba	`map`		Sí	
Anular relación de puerto con interfaz del sistema de prueba	`unmap`		Sí	
Obtener valor de referencia del componente de MTC	`mtc`		Sí	Sí
Obtener valor de referencia del componente de interfaz del sistema de prueba	`system`		Sí	Sí
Obtener valor de referencia del componente propio	`self`		Sí	Sí
Comenzar ejecución del proceso del componente de prueba	`start`		Sí	
Detener ejecución del proceso del componente de prueba	`stop`		Sí	
Suprimir un componente de prueba del sistema	`kill`		Sí	
Comprobar terminación de un proceso PTC	`running`		Sí	

Cuadro 11/Z.140 – Las instrucciones y las operaciones de TTCN-3

Instrucción	Palabra clave o símbolo asociados	Se puede utilizar en control de módulo	Se puede utilizar en funciones, casos de prueba y altsteps	No se puede utilizar en funciones desde plantillas, guardias booleanas o desde la inicialización de definiciones locales alstep
Verificar si en el sistema de prueba existe un PTC	`alive`		Sí	
Esperar terminación de un proceso PTC	`done`		Sí	
Esperar que el PTC deje de existir	`killed`		Sí	
Operaciones de comunicación				
Enviar mensaje	`send`		Sí	
Invocar solicitud de procedimiento	`call`		Sí	
Responder a solicitud de procedimiento de entidad distante	`reply`		Sí	
Plantear excepción (a una solicitud aceptada)	`raise`		Sí	
Recibir mensaje	`receive`		Sí	
Aceptar mensaje	`trigger`		Sí	
Aceptar solicitud de procedimiento de entidad distante	`getcall`		Sí	
Tratar respuesta de una solicitud anterior	`getreply`		Sí	
Reconocer excepción (de entidad llamada)	`catch`		Sí	
Comprobar mensaje/solicitud recibidos (actuales)	`check`		Sí	
Cola para liberar puertos	`clear`		Sí	
Liberar cola y habilitar transmisión y recepción en un puerto	`start`		Sí	
Inhabilitar las operaciones de transmisión y recepción que corresponden a un puerto	`stop`		Sí	
Inhabilitar las operaciones de transmisión y recepción que corresponden a nuevos mensajes/solicitudes	`halt`		Sí	
Operaciones de temporización				
Activar temporizador	`start`	Sí	Sí	
Desactivar temporizador	`stop`	Sí	Sí	
Leer tiempo transcurrido	`read`	Sí	Sí	
Comprobar si temporizador en curso	`running`	Sí	Sí	
Expiración de temporizador	`timeout`	Sí	Sí	
Operaciones de veredicto				
Fijar veredicto local	`setverdict`		Sí	
Obtener veredicto local	`getverdict`		Sí	Sí
Acciones externas				
Estimulación externa de una acción (SUT)	`action`	Sí	Sí	
Ejecución de casos de prueba				
Ejecutar caso de prueba	`execute`	Sí	Sí (véase la nota 2)	

NOTA 1 – Sólo se puede utilizar para controlar operaciones de temporizador.

NOTA 2 – Sólo se puede utilizar en funciones y alternativas (altsteps) que se utilizan en control de módulo.

NOTA 3 – Se prohíbe el cambio de las variables del componente.

NOTA 4 – Puede emplearse en funciones y altsteps (alternativas) pero no en los casos de prueba.

19 Expresiones e instrucciones de programa básicas

19.0 Consideraciones generales

Las expresiones e instrucciones de programa básicas se pueden utilizar en la parte de control de un módulo, y también en funciones, altsteps y casos de prueba TTCN-3.

Cuadro 12a/Z.140 – Instrucciones de programa básicas de TTCN-3

Instrucciones de programa básicas	
Instrucciones	**Palabra clave o símbolo asociados**
Asignaciones	**:=**
Registro	**log**
Etiqueta y Goto	**label / goto**
Condición (If-else)	**if (…) { … } else { … }**
Bucle "For"	**for (…) { … }**
Bucle "While"	**while (…) { … }**
Bucle "Do while"	**do { … } while (…)**
Detener ejecución	**stop**
Seleccionar caso	**select case (…) { case (…) {…} case else {…} }**

19.1 Expresiones

19.1.0 Consideraciones generales

La notación 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 sólo pueden utilizar funciones que devuelven valores. El resultado de una expresión ha de ser un valor de un tipo específico, y los operadores utilizados han de ser compatibles con el tipo de los operandos.

EJEMPLO:

```
(x + y – increment(z))*3;
```

19.1.1 Expresiones booleanas

Una expresión **boolean** sólo contendrá valores **boolean** y/o operadores **boolean** y/o operadores relacionales; deberá traducirse en un valor **booleano true** o **false**.

EJEMPLO:

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

19.2 Asignaciones

El símbolo "**:=**" permite asignar valores a las variables. Es condición que el lado derecho de la asignación se traduzca en un valor compatible con el tipo del lado izquierdo al ejecutar la asignación. El efecto de una asignación es vincular la variable al valor de la expresión. En la expresión no podrá haber variables no vinculadas. Todas las asignaciones se producen en el orden en que aparecen, es decir, procesamiento de izquierda a derecha.

EJEMPLO:

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

19.3 La instrucción Log

La instrucción **log** permite escribir uno o varios elementos de registro (log) en algún dispositivo de registro cronológico asociado al control de la prueba o al componente de prueba en el que se utiliza esta instrucción. Los elementos que han de registrarse se identificarán mediante una lista separada por comas en el argumento de la instrucción log. Puede tratarse de elementos de lenguaje particulares que se especifican en el cuadro 12b o de expresiones compuestas de dichos elementos.

Se recomienda enérgicamente que la ejecución de la instrucción log no tenga repercusión alguna en el comportamiento de la prueba. En particular, las funciones empleadas en una instrucción log no deberían modificar, explícita o implícitamente, los valores de las variables del componente, ni el estado del puerto o el temporizador, y no deberían cambiar el valor de ninguno de sus parámetros **inout** o **out**.

EJEMPLO:

```
var integer myVar:= 1;
log("Line 248 in PTC_A: ", myVar, " (actual value of myVar)");
// La cadena "Line 248 en PTC_A: 1 (actual value of myVar)" se escribe en un dispositivo
// de registro del sistema de prueba
```

NOTA 1 – Las funciones empleadas en las instrucciones log no deberían utilizar directa o indirectamente instrucciones distintas de **if…else**, **for**, **while**, **do…while**, **label**, **goto**, **return**, **mtc**, **system**, **self**, **running** (PTC o temporizador), **read** y **getverdict**.

NOTA 2 – La definición de capacidades complejas de registro y rastreo, que pueden depender de las herramientas, está fuera del ámbito de la presente Recomendación.

Cuadro 12b/Z.140 – Elementos de lenguaje de TTCN-3 que pueden ser registrados

Los que se emplean en una instrucción log	Lo que se registra	Comentarios
identificador de parámetro de módulo	valor efectivo	
valor literal	valor	Incluye además texto libre.
identificador de constante de datos	valor efectivo	
identificador de constante externas	valor efectivo	
ejemplar de plantilla	valores de plantilla o campo efectivos y símbolos de concordancia	
identificador de variable de tipo datos	valor efectivo o "NO INICIALIZADO"	Véanse las notas 3 y 4.
identificador de **self**, **mtc**, **system** o de variable de tipo componente	valor efectivo y, si está asignado, el nombre del ejemplar del componente o valor "NO INICIALIZADO"	En cuanto a los valores efectivos de registro, véanse las notas 2 a 4. El registro de los estados del componente efectivo se realizará conforme a la nota 5.
operación running (componente o temporizador)	valor devuelto	**true** o **false**. En caso de matrices de componentes o temporizadores, se ha de incluir la especificación de los elementos de la matriz.
operación alive (componente)	valor devuelto	**true** o **false**. En caso de matrices, se ha de incluir la especificación de los elementos de la matriz.
ejemplar de puerto	estado efectivo	El registro de los estados del puerto se realizará conforme a la nota 6.
identificador de variable de tipo por defecto	estado efectivo o 'NO INICIALIZADO'	El registro de los estados por defecto se realizará conforme a la nota 7. Véanse además las notas 2 a 4.
nombre de temporizador	estado efectivo	El registro de los estados del temporizador se realizará conforme a la nota 8.
operación read	valor devuelto	Véase 24.3.
funciones predefinidas	valor devuelto	Véase el anexo C.
ejemplar de función	valor devuelto	Sólo se permiten funciones con cláusula return.
ejemplar de función externa	valor devuelto	Sólo se permiten funciones externas con cláusula return.

Cuadro 12b/Z.140 – Elementos de lenguaje de TTCN-3 que pueden ser registrados

Los que se emplean en una instrucción log	Lo que se registra	Comentarios
identificador de parámetros formales	Véase la columna de comentarios	El registro de los parámetros efectivos ha de basarse en las reglas especificadas para los elementos de lenguaje que sustituyen. En caso de parámetros de valor ha de registrarse el valor del parámetro efectivo, en caso de parámetros de tipo plantilla han de registrarse los valores de plantilla o campo efectivos y los símbolos de concordancia, en caso de parámetros de tipo componente ha de registrarse la referencia del componente efectiva, etc. En el caso de los parámetros de temporizador, se permite también el uso de la operación read y en el caso de los parámetros de tipo y temporizador de componente se permite también el uso de la operación running.
NOTA 1 – El valor efectivo/plantilla efectiva representa el valor/plantilla en el momento de ejecución de la instrucción log. NOTA 2 – El tipo de valor registrado depende de la herramienta. NOTA 3 – En caso de identificadores de matrices sin especificación de los elementos de la matriz se han de registrar los valores efectivos, y en caso de referencias de componentes se han de registrar los nombres de todos los elementos de la matriz. NOTA 4 – La cadena "NO INICIALIZADA" se ha de registrar sólo si el elemento no está vinculado (no inicializado). NOTA 5 – Los estados de componente que pueden ser registrados son : Inactive, Running, Stopped y Killed (véase el anexo F para obtener detalles pormenorizados). NOTA 6 – Los estados de puerto que pueden ser registrados son: Started y Stopped (véase el anexo F para obtener detalles pormenorizados). NOTA 7 – Los estados por defecto que pueden ser registrados son: Activated y Deactivated. NOTA 8 – Los estados de temporizador que pueden ser registrados son: Inactive, Running y Expired (véase el anexo F para obtener detalles pormenorizados).		

19.4 La instrucción Label

La instrucción **label** permite especificar etiquetas en casos de prueba, funciones, alternativas (altsteps) y la parte de control de un módulo. Las instrucciones **label** se pueden utilizar libremente como otras instrucciones de programa de comportamiento en TTCN-3 de acuerdo con las reglas de sintaxis definidas en el anexo A. Puede aparecer antes o después de una instrucción TTCN-3, pero no puede ser la primera instrucción de una alternativa o una alternativa principal en una instrucción **alt**, una instrucción **interleave** o un altstep. Es condición que las etiquetas precedidas de la palabra clave **label** sean únicas entre todas las etiquetas definidas en el mismo caso de prueba, función, altstep o parte de control.

EJEMPLO:

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

// En el siguiente fragment de código TTCN-3 se definen las etiquetas L1, L2 y L3
 :
label L1;                                       // Define la etiqueta L1
alt{
[]  PCO1.receive(MySig1)
  { label L2;                                   // Define la etiqueta L2
    PCO1.send(MySig2);
    PCO1.receive(MySig3)
    }
[]  PCO2.receive(MySig4)
  {       PCO2.send(MySig5);
          PCO2.send(MySig6);
          label L3;                             // Define la etiqueta L3
          PCO2.receive(MySig7);
    }
}
 :
```

19.5 La instrucción Goto

La instrucción **goto** se puede utilizar en funciones, casos de prueba, alternativas (altsteps) y en la parte de control de un módulo TTCN-3. La instrucción **goto** ejecuta un salto a una etiqueta (**label**).

La instrucción **goto** permite saltar libremente (adelante o atrás) dentro de una secuencia de instrucciones, saltar fuera de una instrucción compuesta (por ejemplo un bucle **while**) y saltar varios niveles para salir de las instrucciones compuestas jerarquizadas (por ejemplo alternativas jerarquizadas). Ahora bien, la utilización de la instrucción **goto** está sometida a las siguientes reglas:

a) No es posible saltar para salir de funciones, casos de prueba, altsteps o la parte de control de un módulo TTCN-3, ni para entrar en ellos.

b) No es posible saltar para entrar en una secuencia de instrucciones definida en una instrucción compuesta (instrucción **alt**, bucle **while**, bucle **for**, instrucción **if-else**, bucle **do-while** y la instrucción **interleave**).

c) No es posible utilizar la instrucción **goto** dentro de una instrucción **interleave**.

EJEMPLO:

```
// El siguiente fragmento de código TTCN-3 incluye
 :
label L1;                                  // … definición de la etiqueta L1,
MyVar := 2 * MyVar;
if (MyVar < 2000) { goto L1; }             // … salto atrás a L1,
MyVar2 := Myfunction(MyVar);
if (MyVar2 > MyVar) { goto L2; }           // … salto adelante a L2,
PCO1.send(MyVar);
PCO1.receive -> value MyVar2;
label L2;                                  // … definición de la etiqueta L2,
PCO2.send(integer: 21);
alt {
  [] PCO1.receive { }
  [] PCO2.receive(integer: 67) {
         label L3;                         // … definición de la etiqueta L3,
         PCO2.send(MyVar);
         alt {
           []  PCO1.receive { }
           []  PCO2.receive(integer: 90) {
               PCO2.send(integer: 33);
               PCO2.receive(integer: 13);
                  goto L4;                 // … salto adelante para salir de dos instrucciones
                                           // alt jerarquizadas,
              }
           []  PCO2.receive(MyError) {
               goto L3;                    // … salto atrás para salir de la instrucción
                                           // alt actual,
              }
           []  any port.receive {
               goto L2;                    // … salto atrás para salir de dos instrucciones
                                           // alt jerarquizadas,
              }
    }
    }
  []   any port.receive {
       goto L2;                            // … y un largo salto atrás para salir de
                                           // una instrucción alt.
     }
}
label L4;
 :
```

19.6 La instrucción If-else

La instrucción **if-else** (instrucción condicional) se utiliza para indicar una ramificación en el flujo de control debido a expresiones **boolean**. Esquema de la instrucción condicional:

```
if (expression1)
   statementblock1
else
   statementblock2
```

donde statementblock$_x$, hace referencia a un bloque de enunciados.

EJEMPLO:

```
if (date == "1.1.2005"){ 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 La instrucción For

La instrucción **for** define un bucle de contador. El valor de la variable índice aumentará, disminuirá o será manipulado para alcanzar un criterio de terminación después de un determinado número de bucles de ejecución.

La instrucción **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 utiliza para manipular la variable índice.

EJEMPLO 1:

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

La expresión **boolean** especifica el criterio de terminación del bucle. Se comprueba al principio de cada nueva iteración de bucle. Si el resultado es **true**, la ejecución continúa con el bloque de instrucciones en la instrucción **for**, y si es **false** la ejecución continúa con la instrucción que sigue inmediatamente al bucle **for**.

Es posible declarar la variable índice de un bucle **for** antes de utilizarla en la instrucción **for,** o declararla e inicializarla en el encabezamiento de la instrucción **for**. En el segundo caso, el ámbito de la variable índice está limitado al cuerpo del bucle, es decir, sólo es visible dentro del cuerpo del bucle.

EJEMPLO 2:

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

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

19.8 La instrucción While

Las instrucciones **while** se ejecutarán mientras se mantenga la condición del bucle, que es necesario comprobar al principio de cada nueva iteración. Si la condición del bucle no se mantiene, el sistema sale y continúa la ejecución con la instrucción inmediatamente siguiente.

EJEMPLO:

```
while (j<10){ … }
```

19.9 La instrucción Do-while

Comparando con el bucle **while**, la única diferencia del bucle **do-while** es que la condición del bucle se comprueba al *final* de cada iteración. Por tanto, cuando se utiliza un bucle **do-while** el comportamiento será ejecutado por lo menos una vez antes de comprobar la condición del bucle por primera vez.

EJEMPLO:

```
do { … } while (j<10);
```

19.10 La instrucción Stop

La instrucción **stop** termina la ejecución de diferentes maneras según el contexto. Si se utiliza en la parte de control de un módulo o en una función utilizada por la parte de control de un módulo, termina la ejecución de la parte de control del módulo. Si se utiliza en un caso de prueba, un altstep o una función que se ejecutan en un componente de prueba, termina este componente.

EJEMPLO:

```
module MyModule {
     :    // Definiciones de módulo
    testcase MyTestCase() runs on MyMTCType system MySystemType{
          var MyPTCType ptc:= MyPTCType.create;      // PTC creation
          ptc.start(MyFunction());                   // start PTC execution
           :          // test case behaviour continued
          stop        // stops the MTC, all PTCs and the whole test case
    }
    function MyFunction() runs on MyPTCType {

           :
          stop        // detiene sólo al PTC, el caso de prueba continúa
    }
    control {
           :    // ejecución de la prueba
          stop  // detiene la campaña de pruebas
    } // termina el control
} // termina el módulo
```

NOTA – La semántica de una instrucción **stop** que termina un componente de prueba es idéntica a la operación **self.stop** para detener un componente (véase 22.6).

19.11 La instrucción Select Case

La instrucción **select case** representa una alternativa al uso de las instrucciones **if** .. **else** cuando se efectúa una comparación entre un valor y uno o varios otros valores. La instrucción contiene una parte de encabezamiento y cero o más ramificaciones. Nunca se ejecuta más de una de las ramificaciones. La instrucción **select case** se presenta de manera esquemática como sigue:

```
select (expression)
   {
    case (templateInstance1a, templateInstance1b,…)
          statementblock1
    case (templateInstance2a, templateInstance2b,…)
          statementblock2
    case else
          statementblock3
   }
```

Donde templateInstance se refiere a una plantilla definida o en línea y **statementblock**$_x$ se refiere a un bloque de instrucciones.

NOTA – La vista esquemática es equivalente a la siguiente en la que se emplean instrucciones if-else:

```
if (match(expression, templateInstance1a or match(expression, templateInstance1b
or …)
    statementblock1
else if (match(expression, templateInstance2a or match(expression, templateInstance2b or …)
    statementblock2
else
    statementblock3
```

En la parte de encabezamiento de la instrucción **select case** se incluirá una expresión. Cada ramificación comienza con la palabra clave **case** seguida por una lista de templateInstance (una ramificación list, que puede contener también un solo elemento) o con la palabra clave **else** (una ramificación else) y un bloque de instrucciones.

Todos los templateInstance en todas las ramificaciones list habrán de ser de un tipo compatible con el tipo de la expresión en el encabezamiento. Se selecciona una ramificación list y se ejecuta el bloque de instrucciones de la ramificación seleccionada, sólo si alguno de los templateInstance coincide con el valor de la expresión en el encabezamiento de la instrucción. Al ejecutar el bloque de instrucciones de la ramificación seleccionada (es decir, sin saltar mediante una instrucción go to), la ejecución continúa con la instrucción que sigue a la instrucción del caso seleccionado.

El bloque de instrucciones de una ramificación else se ejecuta siempre si no se ha seleccionado otra ramificación que preceda textualmente a la ramificación else.

Las ramificaciones se evalúan en su orden textual. Si ninguno de los templateInstance coincide con el valor de la expresión en el encabezamiento y la instrucción no contiene ninguna ramificación else, la ejecución continúa sin tener en cuenta ninguna de las ramificaciones **`select case`**.

EJEMPLO:

```
select (MyModulePar) // where MyModulePar is of charstring type
{
    case ("firstValue")
          {
           log ("The first branch is selected");
          }
    case (MyCharVar, MyCharConst)
          {
           log ("The second branch is selected");
          }
    case else
          {
           log ("The value of the module parameter MyModulePar is selected");
          }
}
```

20 Instrucciones de programa relativas al comportamiento

20.0 Consideraciones generales

Las instrucciones de programa relativas al comportamiento se pueden utilizar en casos de prueba, funciones y altsteps, excepto:

a) la instrucción **`return`** que sólo se empleará en funciones; y

b) las instrucciones **`alt`**, **`interleave`** y **`repeat`** que también se pueden utilizar en control de módulo.

Las instrucciones de programa relativas al comportamiento especifican el comportamiento dinámico de los componentes de prueba en los puertos de comunicación. El comportamiento de prueba se puede expresar secuencialmente o como un conjunto de alternativas, o combinando estos dos métodos. Un operador de entrelazado permite especificar secuencias entrelazadas o alternativas.

Cuadro 13/Z.140 – Las instrucciones de programa relativas al comportamiento de TTCN-3

Instrucciones de programa relativas al comportamiento	
Instrucción	**Palabra clave o símbolo asociado**
Comportamiento alternativo	**`alt { … }`**
Reevaluación de instrucciones alt	**`repeat`**
Comportamiento entrelazado	**`interleave { … }`**
Devolver control	**`return`**

20.1 Comportamiento alternativo

20.1.0 Consideraciones generales

Una forma más compleja de comportamiento es la expresión de secuencias de instrucciones como conjuntos de posibles alternativas para formar una arborescencia de trayectos de ejecución, como se ilustra en la figura 9.

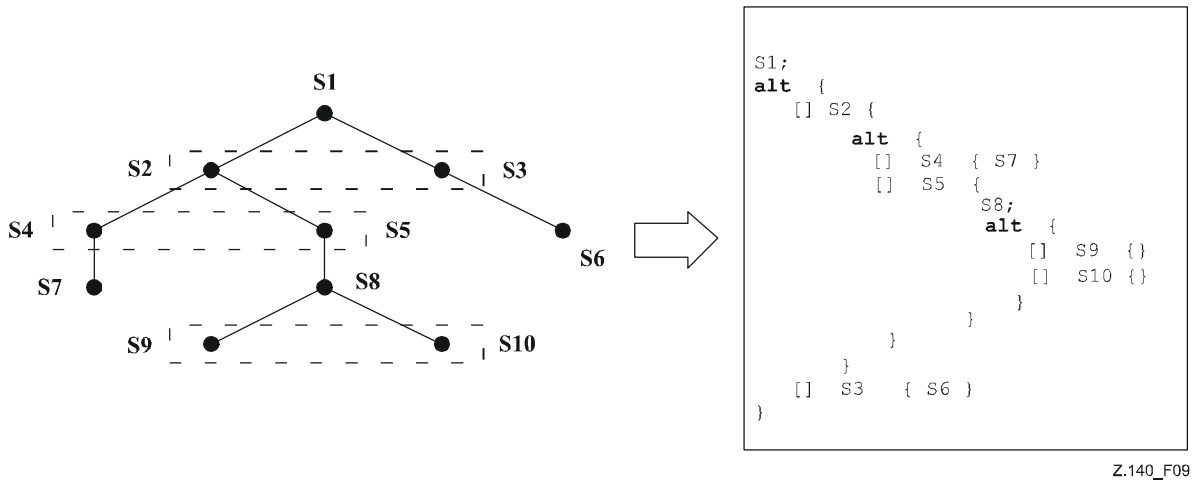


Figura 9/Z.140 – Ilustración de comportamiento alternativo

La instrucción **alt** indica ramificación de comportamiento de prueba debido a la recepción y el tratamiento de eventos de comunicación y/o de temporizador, y/o debido a la terminación de componentes de prueba paralelos. Por tanto, se relaciona con la utilización de las operaciones TTCN-3, **receive**, **trigger**, **getcall**, **getreply**, **catch**, **check**, **timeout**, **done** y **killed**. La instrucción **alt** indica un conjunto de posibles eventos y se trata de establecer si hay concordancia con un determinado estado puntual (véase 20.1.1).

EJEMPLO:

```
// Use of nested alternative statements
 :
alt {
  [] L1.receive(DL_REL_CO:*) {
       setverdict(pass);
       TAC.stop;
       TNOAC.start;
       alt {
         [] L1.receive(t_DL_EST_IN) {
                TNOAC.stop;
                setverdict(pass);
            }
         [] TNOAC.timeout {
             L1.send(t_DEL_EST_RQ);
             TAC.start;
             alt {
                 [] L1.receive(DL_EST_CO:*) {
                       TAC.stop;
                       setverdict(pass)
                   }
                  [] TAC.timeout {
                       setverdict(inconc);
                     }
                  [] L1.receive {
                       setverdict(inconc)
                     }
              }
          }
         [] L1.receive {
                setverdict(inconc)
            }
         }
     }
  [] TAC.timeout {
       setverdict(inconc)
    }
  [] L1.receive {
       setverdict(inconc)
    }
}
 :
```

20.1.1 Ejecución de comportamiento alternativo

Una instrucción **alt** provoca una descripción de estado puntual (*snapshot*). El estado puntual es un estado parcial de un componente de prueba que incluye toda la información necesaria para evaluar las condiciones booleanas colocadas como guardas de ramales alternativos, todos los componentes de prueba detenidos pertinentes, todos los eventos de expiración de temporizador pertinentes y los principales mensajes, solicitudes, respuestas y excepciones en las colas de puertos de entrada pertinentes. Son pertinentes los componentes de prueba, los temporizadores y los puertos referenciados como mínimo en una alternativa en la instrucción **alt**, o en una alternativa principal de un altstep invocado como alternativa en la instrucción **alt**, o activado como opción por defecto. Véase una descripción detallada de la semántica de estados puntuales en la semántica operacional de la notación TTCN-3 (Rec. UIT-T Z.143 [3]).

NOTA 1 – Un estado puntual no es más que un medio conceptual de describir el comportamiento de la instrucción **alt**. En la Rec. UIT-T Z.143 [3] se indican precisamente los algoritmos para la manipulación de estados puntuales.

NOTA 2 – En la semántica de la notación TTCN-3 se supone que un estado puntual es instantáneo, sin duración. Esta descripción puede necesitar cierto tiempo en una implementación real y podría presentarse una situación de competencia. No entra en el ámbito de esta Recomendación describir el tratamiento de estas situaciones.

Los ramales de alternativas de la instrucción **alt** y las alternativas principales de los altsteps invocados y altsteps que están activados por defecto serán tratados en el orden en que aparecen. Si hay varias opciones por defecto activadas, el orden inverso de su activación determina el orden de evaluación de las alternativas principales en estas opciones. Se utiliza el mecanismo por defecto descrito en la cláusula 21 para pasar a los ramales de alternativas en las opciones por defecto activadas.

Hay ramales con expresiones booleanas de guarda y ramales else (ramales de alternativas que empiezan por [**else**]) .

Un ramal else siempre se elige y se ejecuta al alcanzar su posición (véase 20.1.3).

Los ramales que pueden tener expresiones booleanas de guarda invocan un altstep (*altstep-branch*) o tienen en primera posición una operación **done** (*done-branch*), una operación **killed** *(killed-branch*), una operación **timeout** (*timeout-branch*) o una operación de recepción (*receiving-branch*), esto es, **receive**, **trigger**, **getcall**, **getrepy**, **catch** o una operación **check**. La guarda booleana será evaluada basándose en el estado puntual. Se considera que se ha satisfecho la condición de la guarda booleana cuando no se ha definido ninguna guarda y cuando el resultado de la guarda es **true**. A continuación se describe el tratamiento y la ejecución de los ramales.

La condición para elegir un ramal *altstep-branch* es satisfacer la guarda booleana. Al seleccionar este ramal se invoca el altstep referenciado: invocación del altstep y continuación de la evaluación del estado puntual dentro del altstep.

La condición para elegir un ramal *done-branch* es satisfacer la guarda booleana y que el componente de prueba esté especificado en la lista de componentes detenidos del estado puntual. Al seleccionarlo se ejecuta el bloque de instrucciones que sigue a la operación **done**. Por sí misma la operación **done** no tiene ningún otro efecto.

La condición para elegir un ramal *killed-branch* es satisfacer la guarda booleana y que el componente de prueba esté especificado en la lista de componentes killed del estado puntual. Al seleccionarlo se ejecuta el bloque de instrucciones que sigue a la operación **killed**. Por sí misma la operación **killed** no tiene ningún otro efecto.

La condición para elegir un ramal *timeout-branch* es satisfacer la guarda booleana y que el evento de expiración de temporización esté especificado en la lista de expiración de temporizadores del estado puntual. Al seleccionarlo se ejecuta la operación **timeout** especificada (el evento de expiración de temporización se retira de la lista de temporización) y se ejecuta el bloque de instrucciones que sigue a la operación **timeout**.

La condición para elegir un ramal *receiving-branch* es satisfacer la guarda booleana y que uno de los mensajes, solicitudes, respuestas o excepciones del estado puntual cumpla con los criterios de concordancia. Al seleccionarlo se ejecuta la operación de recepción (el mensaje, la solicitud, la respuesta o la excepción que concuerdan se retira de la lista del puerto, posiblemente se asigna la información recibida a una variable y se ejecuta el bloque de instrucciones que sigue a la operación de recepción. Si se trata de la operación de validación (**trigger**) también se retira el primer mensaje de la cola cuando se satisface la guarda booleana, pero no se cumplen los criterios de concordancia. En este caso no se ejecuta el bloque de instrucciones de esa alternativa.

NOTA 3 – En la semántica de la TTCN-3 se describe la evaluación de un estado puntual como una serie de acciones indivisibles de un componente de prueba. En la semántica no se considera que la evaluación de un estado puntual no tenga duración. Durante la evaluación puede ocurrir que se detenga un componente de prueba, que expire un temporizador y que pasen a la cola de un puerto o al componente mensajes, solicitudes, respuestas o excepciones. Ahora bien, estos eventos no se consideran en la evaluación porque no modifican el estado puntual.

Si no fuera posible elegir y ejecutar ninguno de los ramales de alternativa de la instrucción **alt** ni las alternativas principales en los altsteps invocados y las opciones por defecto activas, se ejecutará nuevamente la instrucción **alt**: se registra un nuevo puntual y se repite la evaluación de los ramales de alternativa con este nuevo estado puntual. Este procedimiento repetitivo continuará hasta que se pueda elegir y ejecutar uno de los ramales de alternativa, o hasta que

se detenga el componente de prueba por intervención de otro componente o del sistema de prueba (por ejemplo, porque se ha detenido el MTC) o por un error dinámico.

El caso de prueba se detendrá e indicará un error dinámico si un componente de prueba queda totalmente bloqueado. Ocurre cuando no se puede elegir ninguna de las alternativas, no hay ningún componente de prueba pertinente activo, no hay ningún temporizador pertinente activo y en todos los puertos pertinentes hay como mínimo un mensaje, una solicitud, una respuesta o una excepción que no coincide.

NOTA 4 – El procedimiento repetitivo que consiste en registrar otro estado puntual y reevaluar todas las alternativas es sólo un medio conceptual de describir la semántica de la instrucción **alt**. No entra en el ámbito de esta Recomendación describir el algoritmo que implementa esta semántica.

20.1.2 Selección/deselección de una alternativa

Si es necesario, se puede habilitar/inhabilitar una alternativa colocando una expresión booleana entre los corchetes cuadrados '[ ]' de la alternativa.

Se señala que la evaluación de una expresión booleana en guarda de una alternativa puede tener efectos secundarios. Para evitar estos efectos secundarios que son una causa de incoherencia entre el estado puntual efectivo y el estado del componente, hay que observar las mismas restricciones que se han indicado para la inicialización de definiciones locales dentro de altsteps (véase 16.2.2.1).

Es preciso colocar los corchetes cuadrados de apertura y de cierre '[' ']' al comienzo de cada alternativa, incluso si están vacíos. Esto hace más legible el enunciado y es necesario para distinguir sintácticamente una alternativa de otra.

EJEMPLO:

```
// Utilización de alternativa con expresiones booleanas (o guarda)

 :
alt {
  [x>1]   L2.receive {                        // Guarda/expresión booleana
          setverdict(pass);
      }
  [x<=1]  L2.receive {                        // Guarda/expresión booleana
          setverdict(inconc);
      }
}
 :
```

20.1.3 Ramal Else (o bien) en alternativas

Es posible definir cualquier ramal de una instrucción **alt** como ramal else incluyendo la palabra clave **else** entre los corchetes de apertura y cierre al principio de la alternativa. Este ramal no podrá contener ninguna de las acciones permitidas en ramales que tienen una guarda de expresiones booleanas (solicitud de **altstep** o una operación **done**, **killed**, **timeout** o **receiving**). El bloque de instrucciones del ramal else se ejecutará cuando no ha sido posible ejecutar ninguna de las otras alternativas cuyo texto aparece antes del ramal else.

EJEMPLO:

```
// Utilización de alternativa con expresiones booleanas (o guarda) y el ramal else
 :
alt {
  [x>1]   L2.receive {
          setverdict(pass);
      }
  [x<=1]  L2.receive {
          setverdict(inconc);
      }
  [else]  {                              // ramal else
          MyErrorHandling();
          setverdict(fail);
          stop;
      }
}
 :
```

Se señala que al final de todas las alternativas se invoca invariablemente el mecanismo de opciones por defecto (véase la cláusula 21). Este mecanismo no será solicitado si se ha definido un ramal **else**: en ningún caso se utilizarán opciones por defecto activas.

NOTA 1 – También es posible utilizar **else** en alternativas (altsteps).

NOTA 2 – Dentro de un ramal **else** puede utilizarse una instrucción **repeat**.

NOTA 3 – En una instrucción alt o en una altstep (alternativa) puede definirse más de un ramal else, no obstante siempre se ejecuta sólo el primer ramal else.

20.1.4 En blanco

20.1.5 Reevaluación de instrucciones alt

Una instrucción **repeat** (véase 20.2) permite especificar la reevaluación de una instrucción **alt**.

EJEMPLO:

```
alt {
  [] PCO3.receive {
   count := count + 1;
   repeat                  // Utilización de repeat
     }
  [] T1.timeout { }
  [] any port.receive {
   setverdict(fail);
   stop;
     }
}
```

20.1.6 Invocación de altsteps como alternativas

La notación TTCN-3 permite invocar altsteps como alternativas en instrucciones **alt** (véase 16.2.3).

EJEMPLO:

```
 :
alt {
  [] PCO3.receive { }
  [] AnotherAltStep();     // solicitud explícita del altstep AnotherAltStep
                     // como alternativa de una instrucción alt
  [] MyTimer.timeout { }
}
 :
```

20.2 La instrucción Repeat

El efecto de la instrucción **repeat**, cuando se emplea en el bloque de instrucciones y declaraciones de alternativas de las instrucciones **alt**, es la reevaluación de la instrucción **alt**: se registra otro estado puntual y se hace una evaluación de las alternativas de la instrucción **alt** en el orden en que se especifican. El efecto de la instrucción **repeat**, cuando se emplea en bloques de instrucciones y declaraciones de las partes de tratamiento de respuesta y excepción de las solicitudes de procedimiento de bloqueo, es la reevaluación de la parte de tratamiento de respuesta y excepción de la solicitud (véase 23.3.1.5).

EJEMPLO 1:

```
// Utilización de repeat en una instrucción alt
alt {
  [] PCO3.receive {
   count := count + 1;
   repeat                        // utilización de repeat
     }
  [] T1.timeout { }
  [] any port.receive {
   setverdict(fail);
   stop;
     }
}
```

Si la instrucción **repeat** se emplea en una alternativa principal en una definición de altstep, hace que se registre un nuevo estado puntual y se vuelva a evaluar la instrucción **alt** desde la que se ha solicitado el altstep. El altstep lo puede solicitar de forma implícita el mecanismo de opciones por defecto (véase la cláusula 21) o se solicita explícitamente en la instrucción **alt** (véase 20.1.6).

EJEMPLO 2:

```
// Utilización de repeat en un altstep
altstep AnotherAltStep() runs on MyComponentType {
    [] PCO1.receive{
          setverdict(inconc);
          repeat                        // utilización de repeat
       }
    [] PCO2.receive {}
}
```

20.3 Comportamiento entrelazado

La instrucción **interleave** permite especificar la presentación y el tratamiento entrelazados de las instrucciones **done**, **killed**, **timeout**, **receive**, **trigger**, **getcall**, **catch** y **check**.

En estas instrucciones **interleave** no podrán incluirse las instrucciones de transferencia de control **for**, **while**, **do-while**, **goto**, **activate**, **deactivate**, **stop**, **repeat**, **return**, solicitudes directas de altsteps como alternativas ni solicitudes (directas o indirectas) de funciones definidas por el usuario, que incluyen operaciones de comunicación. Tampoco se permite colocar guardas de expresiones booleanas en los ramales de una instrucción **interleave** (es decir, los corchetes '**[ ]**' estarán siempre vacíos). Tampoco se podrán especificar ramales **else** en comportamiento entrelazado.

El comportamiento entrelazado siempre puede ser sustituido por un conjunto equivalente de alternativas jerarquizadas. Los procedimientos para esta sustitución y la semántica operacional del entrelazado se describen en la Rec. UIT-T Z.143 [3].

Regla para la evaluación de una instrucción de entrelazado:

a) Cada vez que se ejecuta una instrucción de recepción se ejecutan a continuación las instrucciones siguientes que no son de recepción, hasta que se alcance la siguiente instrucción de recepción o termine la secuencia entrelazada.

   NOTA – Las instrucciones de recepción son instrucciones de TTCN-3 que pueden producirse en conjuntos de alternativas: **receive**, **check**, **trigger**, **getcall**, **getreply**, **catch, done, killed** y **timeout**. Las instrucciones que no son de recepción son las demás instrucciones que no son de transferencia de control y que se pueden utilizar en una instrucción **interleave**.

b) Después se registra otro estado puntual para continuar la evaluación.

La semántica operacional del entrelazado se describe en la Rec. UIT-T Z.143 [3].

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);
    }
}
:

// es una expresión simplificada de
:
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 La instrucción Return

La instrucción **return** termina la ejecución de una función o altstep (alternativa) y devuelve el control al punto en el que se ha solicitado la función o altstep. Cuando se emplea en funciones, la instrucción **return** se puede asociar (es facultativo) a un valor de respuesta.

NOTA – El efecto de la instrucción **return**, cuando se emplea en altsteps, es el mismo que si se alcanza el extremo del bloque de instrucciones y declaraciones de la alternativa elegida; por ejemplo, cuando una instrucción **alt** solicita la altstep, la ejecución continúa con la primera instrucción que sigue a la instrucción **alt**.

EJEMPLO:

```
function MyFunction() return boolean {
     :
    if (date == "1.1.2005") {
          return false;     // la ejecución se detiene el 1.1.2000 y devuelve
                            // el valor booleano false
    }
     :
    return true;            // devuelve el valor true
}

function MyBehaviour() return verdicttype {
     :
    if (MyFunction()) {
          setverdict(pass); // utilización de MyFunction en una instrucción if
    }
    else {
          setverdict(inconc);
    }
     :
    return getverdict;      // devolver explícitamente el veredicto
}
```

21 Tratamiento por defecto

21.0 Consideraciones generales

La notación TTCN-3 permite activar altsteps (véase 16.2) como opciones por defecto. Estas opciones por defecto, es decir, altsteps activados se almacenan como una lista ordenada para cada componente de prueba. Se mencionan en el orden inverso de activación: la última opción por defecto activada es el primer elemento en la lista de opciones por defecto activas. Las operaciones TTCN-3 **activate** (véase 21.3) y **deactivate** (véase 21.4) se aplican a la lista de opciones por defecto. La primera coloca otra opción por defecto como el primer elemento en la lista, y la segunda retira una opción por defecto de la lista. Estas opciones de la lista se pueden identificar con una referencia por defecto generada como resultado de la correspondiente operación **activate**.

Cuadro 14/Z.140 – Instrucciones TTCN-3 para el tratamiento por defecto

Instrucciones para el tratamiento por defecto	
Instrucción	**Palabra clave o símbolo asociados**
Activar una opción por defecto	**activate**
Desactivar una opción por defecto	**deactivate**

21.1 El mecanismo de opciones por defecto

Se recurrirá al mecanismo de opciones por defecto al final de todas las instrucciones **alt** si el estado puntual presente ha impedido ejecutar todas las alternativas especificadas. Este mecanismo invoca el primer altstep de la lista de opciones por defecto, es decir, la última opción por defecto activada y espera el resultado que produce la terminación. La terminación puede ser satisfactoria o fallida. Fallida cuando no ha sido posible seleccionar ninguna de las alternativas principales del **altstep** (véase 16.2) que definen el comportamiento por defecto, y satisfactorio si se ha elegido y ejecutado una de las alternativas principales de la opción por defecto.

En caso de terminación fallida el mecanismo de opciones por defecto invoca la siguiente opción de la lista. Si era la última, el mecanismo de opciones por defecto retornará al lugar de la instrucción **alt** en el que fue invocado, es decir, al final de la instrucción **alt** y señalará el resultado fallido en la ejecución de una opción por defecto. También se señala este resultado cuando la lista de opciones por defecto está vacía.

Un resultado fallido en la ejecución de una opción por defecto puede dar origen a un nuevo estado puntual o producir un error dinámico si el componente de prueba está bloqueado (véase 20.1).

Si el resultado es satisfactorio, la opción por defecto puede detener el componente de prueba mediante una instrucción **stop** o el flujo de control principal del componente continuará inmediatamente después de la instrucción **alt** en la que se ha solicitado el mecanismo de opciones por defecto, o el componente de prueba registrará otro estado puntual y reevaluará la instrucción **alt**. Esta última acción se tiene que especificar mediante una instrucción **repeat** (véase 20.2). Si la alternativa principal elegida de la opción por defecto termina sin una instrucción **repeat** el flujo de control del componente de prueba continuará inmediatamente después de la instrucción **alt**.

NOTA – La notación TTCN-3 no restringe la implementación del mecanismo de opciones por defecto. Por ejemplo, puede implementarse como un proceso que será solicitado implícitamente al final de cada instrucción **alt** o como una hebra (*thread*) particular destinada únicamente al tratamiento de la opción por defecto. La única condición es que se soliciten las opciones en el orden inverso de activación cuando se invoca el mecanismo de opciones por defecto.

21.2 Referencias de opciones por defecto

Se trata de referencias únicas a las opciones por defecto activadas. Un componente de prueba genera esta referencia al activar un altstep como opción por defecto, es decir, la referencia a una opción por defecto es el resultado de una operación **activate** (véase 21.3).

Las referencias a opciones por defecto son del tipo especial predefinido **default**. Las variables del tipo **default** permiten manipular opciones por defecto activadas en componentes de prueba. Hay un valor especial **null** para indicar una referencia a una opción por defecto no definida, por ejemplo para inicializar variables de manipulación de estas referencias.

Las referencias a opciones por defecto se utilizan en las operaciones **deactivate** (véase 21.4) para identificar la opción por defecto a desactivar.

El sistema de prueba ha de resolver externamente la representación de datos efectiva del tipo **default**. Siendo así, es posible especificar casos de prueba abstractos independientemente de cualquier entorno real de ejecución TTCN-3. Dicho de otra forma, en la notación TTCN-3 no hay restricciones relativas a la manipulación y la identificación de opciones por defecto para la implementación de un sistema de prueba.

EJEMPLO:

```
// Declaración de una variable para tratar opciones por defecto e inicialización
// con el valor null
var default MyDefaultVar := null;
 :
// Utilización de MyDefaultVar para almacenar una opción por defecto activada
MyDefaultVar := activate(MyDefAltStep());
// MyDefAltStep ha sido activada como opción por defecto
 :
```

```
// Utilización de MyDefaultVar para desactivar la opción por defecto MyDefAltStep
deactivate(MyDefaultVar);
 :
```

21.3 La operación activate

21.3.0 Consideraciones generales

La operación **activate** se utiliza para activar altsteps como opciones por defecto. Coloca el altstep referenciado como primer elemento en la lista de opciones por defecto y devuelve una referencia de opción por defecto. Esta referencia es un identificador único que se podrá utilizar en una operación **deactivate** para desactivar la opción por defecto.

El efecto de la operación **activate** es local para el componente de prueba en el que se solicita. Un componente de prueba no puede activar una opción por defecto en otro componente.

EJEMPLO 1:

```
 :
// Declaración de una variable para tratar opciones por defecto
var default MyDefaultVar := null;
 :
// Declaración de una referencia a una opción por defecto y activación de un altstep
// como opción por defecto
var default MyDefVarTwo := activate(MySecondAltStep());
 :
// Activación del altstep MyAltStep como opción por defecto
MyDefaultVar := activate(MyAltStep());
// MyAltStep activado como opción por defecto
 :
```

Puede solicitarse la operación **activate** sin necesidad de guardar la referencia de la opción por defecto devuelta. Esta forma es útil en casos de prueba que no exigen la desactivación explícita de la opción por defecto activada, es decir, dicha desactivación se realiza implícitamente en la terminación MTC.

EJEMPLO 2:

```
 :
// Activación de una altstep (alternativa) como una opción por defecto, sin asignación
// de una referencia de la opción por defecto
activate(MyCommonDefault());
```

21.3.1 Activación de altsteps parametrizados

Los parámetros efectivos de un altstep parametrizado (véase 16.2.1) que habría que activar como opción por defecto se han de señalar en la instrucción **activate** correspondiente. Por tanto, los parámetros efectivos se vinculan a la opción por defecto en el momento de activarlos (y no, por ejemplo, cuando los invoca el mecanismo de opciones por defecto). Todos los temporizadores en la lista de parámetros efectivos habrán de declararse como temporizadores locales de tipo componente (véase 8.5.1).

EJEMPLO:

```
altstep MyAltStep2    ( integer par_value1, MyType par_value2, MyPortType par_port,
                        timer par_timer )
{
 :
}

function MyFunc () runs on MyCompType

{ :
var default MyDefaultVar := null;

MyDefaultVar := activate(MyAltStep2(5, myVar, myCompPort, myCompTimer);
    // Activación de MyAltStep2 como opción por defecto con los parámetros efectivos 5 y
    // el valor de myVar. La modificación de myVar antes de que el mecanismo de opciones
    // por defecto solicite MyAltStep2 no modifica los parámetros efectivos de la solicitud.
 :
}
```

21.4 La operación deactivate

La operación **deactivate** se utiliza para desactivar opciones por defecto (altsteps activados anteriormente). La operación **deactivate** retira la opción referenciada de la lista de opciones por defecto.

El efecto de la operación **deactivate** es local para el componente de prueba en el que se solicita. Un componente de prueba no puede desactivar una opción por defecto en otro componente.

Una operación **deactivate** sin parámetros desactiva todas las opciones por defecto de un componente de prueba.

La solicitud de una operación **deactivate** con el valor especial **null** no produce ningún efecto. La solicitud de una operación **deactivate** con una referencia de opción por defecto no definida, por ejemplo una referencia caducada a una opción por defecto ya desactivada o una variable referencia de opción por defecto no inicializada provocará un error de funcionamiento.

EJEMPLO:

```
 :
var default MyDefaultVar := null;
var default MyDefVarTwo := activate(MySecondAltStep());
var default MyDefVarThree := activate(MyThirdAltStep());
 :
MyDefaultVar := activate(MyAltStep());
 :
deactivate(MyDefaultVar); // deactivates MyAltStep)
 :
deactivate;     // desactiva las otras opciones por defecto, en este caso MySecondAltStep
                // y MyThirdAltStep
 :
```

22 Operaciones de configuración

22.0 Consideraciones generales

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

Cuadro 15/Z.140 – Las operaciones de configuración de TTCN-3

Operación	Explicación	Ejemplos de sintaxis
Operaciones de conexión		
connect	Conecta el puerto de un componente de prueba con el puerto de otro componente de prueba	`connect(ptc1:p1, ptc2:p2);`
disconnect	Desconecta dos o más puertos conectados	`disconnect(ptc1:p1, ptc2:p2);`
map	Relaciona el puerto de un componente de prueba con el puerto de la interfaz del sistema de prueba	`map(ptc1:q, system:sutPort1);`
unmap	Anula la relación entre dos o más puertos relacionados	`unmap(ptc1:q, system:sutPort1);`
Operaciones de componente de prueba		

create	Crea un componente de prueba normal o activo (alive); la distinción entre componentes de prueba normales y activos se realiza durante la creación (el MTC funciona como un componente de prueba normal)	Componentes de prueba inactivos (non-alive):
		`var PTCType c := PTCType.create;`
		Componentes de prueba activos (alive):
		`var PTCType c := PTCType.create alive;`

start	El comienzo del comportamiento de prueba de un componente de prueba o el comienzo de un comportamiento, no tiene repercusión alguna en el estado de las variables de componente, los temporizadores o los puertos	`c.start(PTCBehaviour());`
stop	Detiene el comportamiento de prueba de un componente de prueba	`c.stop;`
kill	Provoca que un componente de prueba deje de existir	`c.kill;`
alive	Devuelve true cuando se crea el componente de prueba y está preparado para ejecutar un comportamiento o ya lo está ejecutando; en los demás casos devuelve false	`if (c.alive) …`
running	Devuelve true mientras el componente de prueba esté ejecutando un comportamiento; en los demás casos devuelve false	`if (c.running) …`
done	Comprueba si ya concluyó la función running en un componente de prueba	`c.done;`
killed	Comprueba si un componente de prueba ha dejado de existir	`c.killed { … }`

Cuadro 15/Z.140 – Las operaciones de configuración de TTCN-3

Operación	Explicación	Ejemplos de sintaxis
Operaciones de referencia		
mtc	Hace llegar la referencia al MTC	**connect**(**mtc**:p, ptc:p);
system	Hace llegar la referencia a la interfaz del sistema de prueba	**map**(c:p, **system**:sutPort);
self	Hace llegar la referencia al componente de prueba que ejecuta esta operación	**self.stop;**

22.1 La operación Create (crear)

El MTC es el único componente de prueba creado automáticamente al comenzar un caso de prueba. Habrá que crear explícitamente los demás componentes de prueba (PTC) durante la ejecución mediante operaciones **create**, que definen igualmente todos sus puertos, con colas de entrada vacías, y con todas las constantes, variables y temporizadores. Los puertos definidos con un tipo **in** o **inout** quedarán en estado de escucha preparados para recibir tráfico por la conexión.

Al crear un componente de forma explícita o implícita se restablecen todas sus variables y todos sus temporizadores al valor inicial (en su caso), y todas sus constantes se restablecen a los valores asignados.

Hay dos tipos de PTC: uno que puede ejecutar una función behaviour (comportamiento) una sola vez y otro que se mantiene activo tras la terminación de dicha función y que, por consecuencia, puede volver a utilizarse para ejecutar otra función. El segundo se crea mediante la palabra clave **alive** adicional. Un PTC de tipo alive debe ser destruido explícitamente mediante la operación **kill** (véase 22.9), mientras que uno de tipo non-alive se destruye implícitamente tras la conclusión de la función comportamiento. La terminación de un caso de prueba (el MTC) termina todos los PTC que existen, si los hubiere.

Como todos los componentes y puertos de prueba se destruyen implícitamente al terminar un caso de prueba, cada caso creará completamente su configuración requerida de componentes y conexiones cuando se invoca.

La operación **create** devolverá la referencia de componente única del nuevo ejemplar creado. La referencia única al componente se almacena generalmente en una variable (véase 8.7) y se puede utilizar para conectar ejemplares y para fines de comunicación, por ejemplo envío o recepción.

Facultativamente, puede asociarse un nombre con el componente recién creado. El nombre será un valor **charstring** y cuando se asigne tomará el lugar del argumento de la función **create**. Durante la creación, el sistema de prueba asociará automáticamente los nombres 'MTC' al MTC y 'SYSTEM' a la interfaz del sistema de prueba. No es necesario que los nombres de los componentes asociados sean únicos.

NOTA – El nombre del componente se emplea sólo para fines de registro (véase 19.3) y no para hacer referencia al componente (para este efecto se empleará la referencia del componente) y no tiene repercusión alguna sobre la concordancia.

EJEMPLO:

```
// Este ejemplo declara variables de tipo MyComponentType, que se emplean para guardar
// las referencias de componentes recien creados de tipo MyComponentType que son el
// resultado de las operaciones create. Se asigna un nombre asociado a algunos de los
// componentes creados.
 :
var MyComponentType MyNewComponent;
var MyComponentType MyNewestComponent;
var MyComponentType MyAliveComponent;
var MyComponentType MyAnotherAliveComponent;
 :
MyNewComponent := MyComponentType.create;
MyNewestComponent := MyComponentType.create("Newest");
MyAliveComponent := MyComponentType.create alive;
MyAnotherAliveComponent := MyComponentType.create("Another Alive") alive;
 :
```

La posibilidad de crear componentes en cualquier punto en una definición de comportamiento es una garantía de flexibilidad con respecto a configuraciones dinámicas (cualquier componente puede crear cualquier otro PTC). Las reglas de ámbito que determinan la visibilidad de referencias de componentes son las mismas reglas de las variables; para hacer referencia a componentes fuera del ámbito de creación propio es necesario transferir la referencia del componente como un parámetro o como un campo en un mensaje.

22.2 Las operaciones Connect y Map (conectar y establecer relación)

22.2.0 Consideraciones generales

Los puertos de un componente de prueba se pueden conectar a otros componentes o a los puertos de la interfaz del sistema de prueba. Para las conexiones entre dos componentes de prueba se utilizará la operación **connect**. Para conectar un componente de prueba a la interfaz del sistema de prueba, se utilizará la operación **map**. La operación **connect** conecta directamente un puerto a otro, con el lado **in** conectado al lado **out** y viceversa. La operación **map** puede considerarse simplemente como una traducción de nombres que define la forma de referenciar los trenes de comunicación.

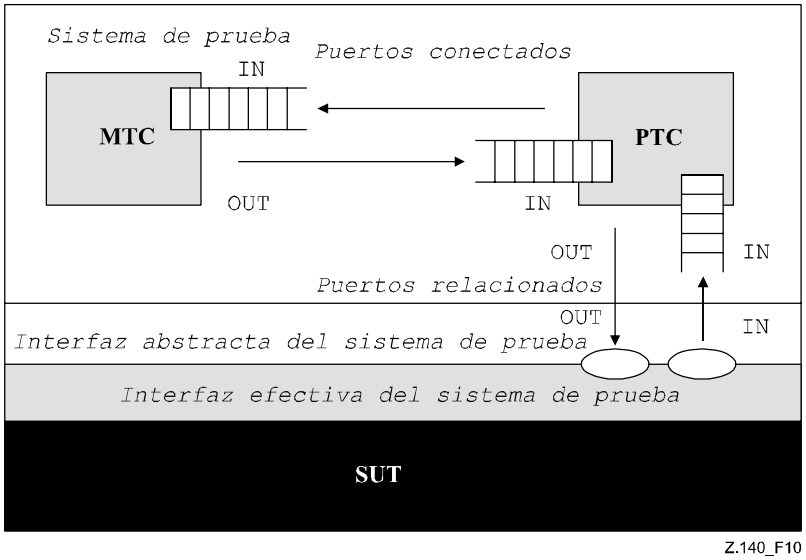


Figura 10/Z.140 – Ilustración de las operaciones connect y map

Los puertos que se han de conectar se identifican mediante las referencias de los componentes a conectar y los nombres de los puertos a conectar, tanto en la operación **connect** como en la operación **map**.

La operación **mtc** identifica el MTC, la operación **system** identifica la interfaz del sistema de prueba y la operación **self** identifica el componente de prueba de donde se solicitó la operación **self** (véase 22.4). Todas estas operaciones se pueden utilizar para identificar y conectar puertos.

Es posible solicitar las operaciones **connect** y **map** desde cualquier definición de comportamiento, excepto la parte de control de un módulo. Ahora bien, es condición que antes de solicitar una de estas operaciones se hayan creado los componentes que se han de conectar y se conozcan sus referencias de componente así como los nombres de los puertos pertinentes.

Las dos operaciones **map** y **connect** permiten la conexión de un puerto a más de un puerto. No se permite la conexión a un puerto que está en relación con la interfaz (map), ni establecer una relación con la interfaz para 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 MyNewPTC;
 :
MyNewPTC := MyComponentType.create;
 :
 :
connect(MyNewPTC:Port1, mtc:Port3);
map(MyNewPTC:Port2, system:PCO1);
 :
 :
// En este ejemplo se crea un nuevo componente de tipo MyComponentType y su referencia se
// almacena en la variable MyNewPTC. Después, en la operación connect, se conecta Port1
// de este nuevo componente con Port3 del MTC. Mediante la operación map se conecta Port2
// del nuevo componente con el puerto PCO1 de la interfaz del sistema de prueba
```

22.2.1 Conexiones y relaciones coherentes

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

Suponiendo que:

a) los puertos PORT1 y PORT2 son los puertos que se han de conectar;

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 ⊆ inlist-PORT2 y outlist-PORT2 ⊆ inlist-PORT1.

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

- outlist-PORT1 ⊆ outlist-PORT2 *e* inlist-PORT2 ⊆ inlist-PORT1.

No está permitido realizar las operaciones en ningún otro caso.

Como la notación TTCN-3 permite configuraciones y direcciones dinámicas, no es posible hacer todas estas comprobaciones de coherencia estáticamente al hacer la compilación. Las comprobaciones que no se puedan hacer en el momento de la compilación se harán en el momento de ejecución. Si el resultado es fallido se producirá un error de caso de prueba.

22.3 Las operaciones Disconnect y Unmap (desconectar y anular relación)

Las operaciones **disconnect** y **unmap** son las operaciones contrarias de **connect** y **map**. Desconectan puertos (previamente conectados) de componentes de prueba y anulan la relación entre puertos de componentes de prueba y puertos en la interfaz del sistema de prueba previamente relacionados.

Es posible solicitar las operaciones **disconnect** y **unmap** desde cualquier componente si se conocen las referencias de componentes y los nombres de los puertos pertinentes. Una operación **disconnect** y **unmap** sólo tiene efecto si se trata de anular una conexión o una relación creadas anteriormente.

EJEMPLO 1:

```
  :
  :
connect(MyNewComponent:Port1, mtc:Port3);
map(MyNewComponent:Port2, system:PCO1);
  :
  :
disconnect(MyNewComponent:Port1, mtc:Port3);   // desconectar la conexión hecha anteriormente
unmap(MyNewComponent:Port2, system:PCO1);      // anular la relación definida anteriormente
```

Para facilitar las operaciones **disconnect** y **unmap** relacionadas con todas las conexiones y concordancias de un componente o un puerto, pueden emplearse dichas operaciones con un solo argumento, el cual especifica que se ha de desconectar o anular la concordancia de un lado de las conexiones. La palabra clave **all port** puede ser útil para señalar todos los puertos de un componente.

EJEMPLO 2:

```
  :
disconnect(MyNewComponent:Port1);      // desconecta todas las conexiones del Port1,
                                       // que pertenece al componente MyNewComponent.
unmap(MyNewComponent:all port);        // anula la concordancia de todos los puertos
                                       // del componente MyNewComponent
  :
```

La utilización de una operación **disconnect** o **unmap** sin ningún parámetro representa una forma simplificada de utilizar la operación con el parámetro **self:all port**. Éste desconecta o anula la concordancia de todos los puertos del componente que solicita la operación.

EJEMPLO 3:

```
 :
disconnect;                              // es una forma simplificada de …
disconnect(self:all port);               // que desconecta todos los puertos del componente
                                         // que solicitó la operación
 :
unmap;                                   // es una forma simplificada de …
unmap(self:all port);                    // que anula la concordancia de todos los puertos
                                         // del componente que solicitó la operación
 :
```

La palabra clave **`all component`** ha de utilizarse únicamente en combinación con la palabra clave **`all port`**, es decir, **`all component:all port`**, y sólo podrá emplearla el MTC. Además, el argumento **`all component:all port`** debe utilizarse como el único argumento de una operación **`disconnect`** o **`unmap`** y permite liberar todas las conexiones y concordancias de la configuración de prueba.

EJEMPLO 4:

```
 :
 :
disconnect(all component:all port);      // el MTC desconecta todos los puertos de todos
                                         // los componentes en la configuración de prueba.
 :
 :
unmap(all component:all port);           // el MTC anula la concordancia de todos los puertos
                                         // de todos los componentes en la configuración de
                                         // prueba.
 :
```

22.4 Las operaciones MTC, System y Self

Hay tres operaciones (**`mtc`** y **`system`**) que devuelven la referencia del componente de prueba principal y de la interfaz del sistema de prueba respectivamente (véase 8.7). La operación **`self`** se puede utilizar para devolver la referencia del componente en el que se ha solicitado.

EJEMPLO:

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

Las operaciones de asignación, igualdad y desigualdad son las únicas permitidas para las referencias de componentes.

22.5 La operación Start (activar un componente de prueba)

Después de crear y conectar un PTC es necesario vincular el comportamiento al PTC y lanzar la ejecución de dicho comportamiento. Es el objeto de la operación **`start`** (ya que la creación de PTC no lanza la ejecución del comportamiento del componente). Se distingue entre **`create`** y **`start`** para poder efectuar operaciones de conexión antes de ejecutar efectivamente el componente de prueba.

La operación **`start`** vinculará el comportamiento requerido al componente de prueba. Este comportamiento se define mediante referencia a una función ya definida.

Un PTC de tipo alive puede ejecutar varias funciones de comportamiento en orden secuencial. Si se activa una segunda función de comportamiento en un PTC de tipo non-alive o una función en un PTC que aún está funcionando, se produce un error de caso de prueba. Si, tras la terminación de una función anterior, se activa una función en un PTC tipo alive, éste empleará los valores de variables, temporizadores, puertos y el veredicto local en el estado en que se encontraban tras la terminación de la función anterior. En particular, si en la función anterior se activó un temporizador, convendría habilitar la función subsiguiente a fin de tratar un posible evento de fin de temporización.

EJEMPLO:

```
function MyFirstBehaviour() runs on MyComponentType { … }
function MySecondBehaviour() runs on MyComponentType { … }
 :
var MyComponentType MyNewPTC;
var MyComponentType MyAlivePTC;
 :
MyNewPTC := MyComponentType.create;          // Creación de un nuevo componente de prueba
                                             // non-alive.
MyAlivePTC := MyComponentType.create alive;  // Creación de un nuevo componente de prueba
                                             // de tipo alive
 :
```

```
MyNewPTC.start(MyFirstBehaviour());          // Activación del componente non-alive.
MyNewPTC.done;                               // En espera de la terminación
MyNewPTC.start(MySecondBehaviour());         // Error de caso de prueba
 :
MyAlivePTC.start(MyFirstBehaviour());        // Activación del componente tipo alive
MyAlivePTC.done;                             // En espera de la terminación
MyAlivePTC.start(MySecondBehaviour());       // Activación de la siguiente función en
                                             // el mismo componente
 :
```

La función invocada en una operación **start** para lanzar un componente de prueba está sometida a las siguientes restricciones:

- Si esta función tiene parámetros, sólo serán parámetros **in**, es decir, parámetros por valor.
- Esta función tendrá una definición **runs on** que hace referencia a un tipo de componente que es compatible con el nuevo componente creado (véase 6.7.3).
- No se transferirán puertos ni temporizadores a esta función.

NOTA – Como los puertos **in** e **inout** quedan en estado de escucha al crear el componente, es posible que haya mensajes en espera de tratamiento en las colas de entrada de estos puertos al lanzar la ejecución.

22.6 La operación Stop (detener un comportamiento de prueba)

La instrucción **stop** permite a un componente de prueba detener la ejecución de su propio comportamiento de prueba de funcionamiento en curso o la ejecución del comportamiento de prueba que funciona en otro componente del sistema prueba. En el segundo caso es necesario utilizar la referencia para identificar el componente que se va a detener. Un componente puede detener su propio comportamiento mediante una simple instrucción **stop** (detener ejecución) (véase 19.10) o designándose a sí mismo en la operación **stop**, por ejemplo mediante la operación **self**.

EJEMPLO 1:

NOTA 1 – Las operaciones **create**, **start**, **running**, **done** y **killed** sólo se pueden utilizar para los PTC, pero la operación **stop** también se puede aplicar al MTC.

```
var MyComponentType MyComp := MyComponentType.create;  // Creación de un nuevo componente
                                                       // de prueba
MyComp.start(CompBehaviour());                         // Lanzamiento del nuevo componente
 :
if (date == "1.1.2005") {
   MyComp.stop;            // El componente "MyComp" se detiene
}

 :
if (a < b ) {
    :
   self.stop;        // El componente de prueba que está en ejecución detiene su propio
                     // comportamiento
}
 :
   stop              // El componente de prueba detiene su propio comportamiento
```

La forma explícita de terminar la ejecución del comportamiento de funcionamiento en curso es detener un componente de prueba. Un comportamiento de componente de prueba también se termina al completar su ejecución cuando se alcanza el final del caso de prueba o de la función que está activada en este componente o bien mediante una instrucción **return** explícita. Esta forma de terminación se conoce asimismo como detención implícita, que produce el mismo efecto que una detención explícita: el veredicto global se actualiza con el veredicto local del componente de prueba detenido (véase la cláusula 25).

Si el componente de prueba detenido es el MTC, se liberan los recursos de todos los PTC, los PTC se suprimen del sistema de prueba y el caso de prueba se termina (véase 27.2).

Si se detiene un componente de prueba de tipo non-alive (implícita o explícitamente) se provoca su destrucción y la liberación de todos los recursos asociados con el componente de prueba.

Si se detiene un componente de tipo alive se detiene únicamente el comportamiento del funcionamiento actual pero el componente sigue existiendo y puede ejecutar un nuevo comportamiento (que se activa mediante la operación **start**). El componente permanece en un estado coherente tras detener su comportamiento.

NOTA 2 – Por ejemplo, si se detiene el comportamiento de un componente tipo alive durante la asignación de un nuevo valor a una variable vinculada, ésta permanece vinculada tras haber detenido el componente (con el valor antiguo o con el nuevo). De modo similar, si se detiene el componente durante la reactivación de un temporizador activo, éste permanece en estado activo tras la terminación del comportamiento.

En la cláusula 25 se precisan las reglas para terminar casos de prueba y calcular el veredicto de prueba final.

Sólo el MTC puede utilizar la palabra clave **all** para detener todos los PTC activos, excepto el propio MTC.

NOTA 3 – Un PTC puede detener la ejecución del caso de prueba deteniendo el MTC.

EJEMPLO 2:

```
 :
all component.stop          // El MTC detiene todos los PTC del caso de prueba
                            // pero no su propia ejecución.
 :
```

NOTA 4 – No entra en el ámbito de esta Recomendación definir precisamente el mecanismo para detener los PTC.

22.7 La operación Running (activo)

El comportamiento de un componente de prueba puede utilizar la operación **running** para indagar si ha terminado la ejecución del comportamiento de otro componente. La operación **running** sólo se puede utilizar para los PTC, y devuelve **true** a los PTC que han sido activados pero que aún no han sido terminados o detenidos. En otros casos devuelve **false**. Como la operación **running** se considera una expresión **boolean**, devuelve un valor **boolean** para indicar si ha terminado el componente de prueba especificado (o todos los componentes). A diferencia de la operación **done**, la operación **running** se puede utilizar libremente en expresiones **boolean**.

Cuando se utiliza la palabra clave **all**, la operación **running** devolverá el valor **true** si todos los PTC que se activaron y no han sido detenidos explícitamente por otro componente están ejecutando su comportamiento. En otros casos devuelve el valor **false**.

Cuando se utiliza la palabra clave **any**, la operación **running** devolverá el valor **true** si hay al menos un PTC que está ejecutando su comportamiento. En otros casos devuelve el valor **false**.

EJEMPLO:

```
if (PTC1.running)                       // Utilización de running en una instrucción if
{
    // Especificación de alguna acción!
}

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

22.8 La operación Done (terminado)

El comportamiento de un componente de prueba puede utilizar la operación **done** para indagar si ha terminado la ejecución del comportamiento de otro componente. La operación **done** sólo se puede utilizar para los PTC.

La operación **done** se utilizará como una operación de recepción o una operación **timeout**. Esto significa que no se utilizará en una expresión **boolean**, pero se puede emplear para determinar una alternativa en una instrucción **alt** o como instrucción autónoma en una descripción de comportamiento. En el segundo caso, se considera que la operación **done** es una expresión simplificada de una instrucción **alt** con una sola alternativa, es decir, tiene semántica bloqueante y por eso permite esperar pasivamente la terminación de componentes de prueba.

Cuando se aplica la operación **done** a un PTC, habrá concordancia sólo si su comportamiento ha sido detenido (implícita o explícitamente) o si el PTC ha sido eliminado. En otros casos la concordancia no es satisfactoria.

Cuando se utiliza la palabra clave **all** con la operación **done**, habrá concordancia si no hay ningún PTC que esté ejecutando su comportamiento, y también cuando no se ha creado ningún PTC.

Cuando se utiliza la palabra clave **any** con la operación **done**, habrá concordancia si al menos el comportamiento de un PTC ha sido detenido o eliminado. En los otros casos la concordancia no es satisfactoria.

NOTA – Si se detiene el comportamiento de un componente tipo non-alive, la acción también se traduce como su supresión del sistema de prueba, mientras que si se detiene un componente tipo alive la acción permite que éste permanezca activo (alive) en el sistema de prueba. En ambos casos la operación **done** concuerda.

EJEMPLO:

```
// Utilización de done en alternativas
:
alt {
    []    MyPTC.done {
          setverdict(pass)
          }

    []    any port.receive {
          repeat

          }
}
:
var MyComp c := MyComp.create alive;
c.start(MyPTCBehaviour());
:
c.done;
    // concuerda en cuanto se detiene la función MyPTCBehaviour (o la función/altstep
    // solicitada por ella)
c.done;
    // concuerda también con el final de of MyPTCBehaviour (o la función/altstep
    // solicitada por ella)
if(c.running) {c.done}
    // en este punto la operación done concuerda sólo con el final del siguiente
    // comportamiento

// esta operación done en una instrucción autónoma:
 :
all component.done;
 :

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

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

22.9 La operación Kill (componente de prueba eliminar)

Cuando se aplica la operación **kill** a un componente de prueba, se detiene la ejecución de su comportamiento activo (si lo hubiere), se liberan todos los recursos asociados con el mismo (incluidas todas las conexiones de puertos del componente eliminado) y se suprime el componente del sistema de prueba. La operación **kill** puede aplicarse al propio componente de prueba efectivo mediante una sola instrucción **kill** o designándose a sí misma mediante las operaciones **self** y kill. Esta última también puede aplicarse a otro componente de prueba. El componente que ha de ser eliminado será designado mediante su referencia de componente. Si la operación **kill** se aplica al MTC, por ejemplo, **mtc.kill** , se termina el caso de prueba.

EJEMPLO 1:

```
var PTCType MyAliveComp := PTCType.create alive;   // Crea un componente de prueba tipo
                                                   // alive
MyAliveComp.start(MyFirstBehaviour());             // Se activa el nuevo componente
MyAliveComp.done;                                  // En espera de la  terminación
MyAliveComp.start(MySecondBehavior());             // Activa el componente por segunda vez
MyAliveComp.done;                                  // En espera de terminación
MyAliveComp.kill;                                  // Libera sus recursos
```

El MTC puede emplear la palabra clave **all** únicamente para detener y eliminar todos los PTC activos salvo el propio MTC.

EJEMPLO 2:

```
all component.kill;        // El MTC detiene en primer lugar todos los PTC (de tipo alive
                           // y normal) del caso de prueba y libera sus recursos.
```

22.10 La operación Alive (activo)

Se trata de una operación booleana que comprueba si se ha creado un componente de prueba y si éste está preparado para ejecutar una función de comportamiento o si ya la está ejecutando. Cuando se aplica la operación **alive** a un componente de prueba normal, devuelve true si el componente está inactivo o ejecutando una función y false en los demás casos. Cuando se aplica a un componente de prueba de tipo alive, devuelve true si el componente está inactivo, activo o detenido. Devuelve false si el componente ha sido eliminado.

La operación **alive** puede emplearse de forma similar a la operación **running** sólo en los PTCS (véase 22.7). En particular, cuando se combina con la palabra clave **all**, devuelve true si todos los PTC (tipo alive o normal) están activos.

Si la operación **alive** se combina con la palabra clave **any**, devuelve true si al menos un PTC está activo.

EJEMPLO:

```
 :
PTC1.done;                                  // En espera de la terminación del componente
if (PTC1.alive) {                           // Si el componente aun está activo …
    PTC1.start(AnotherFunction());          // … ejecutar otra función en el mismo.
}
```

22.11 La operación Killed (eliminado)

Esta operación permite determinar si un componente de prueba distinto está activo o ha sido suprimido del sistema de prueba.

La operación **killed** será utilizada de la misma forma que las operaciones de recepción. Lo anterior significa que no será utilizada en las expresiones **booleanas**, pero podrá ser aprovechada para determinar una alternativa en una instrucción **alt** o como una instrucción autónoma en una descripción de comportamiento. En el segundo caso, la operación **done** se considera una forma simplificada de una instrucción **alt** con una sola alternativa, es decir, dispone de bloqueo, semántica, y por consiguiente, también de la capacidad para esperar de forma pasiva la terminación de los componentes de prueba.

NOTA – Cuando se comprueban componentes de prueba normales, una operación killed concuerda si detiene (implícita o explícitamente) la ejecución del comportamiento del componente o si ha sido **eliminada** (**killed**) explícitamente, es decir, resulta equivalente a la operación done (véase 22.8). No obstante, cuando se comprueban componentes de prueba de tipo alive, la operación **killed** concuerda únicamente si el componente ha sido eliminado mediante la operación **kill**. En los otros casos la operación **killed** no tiene éxito.

La operación **killed** sólo puede ser utilizada para PTC.

Cuando la palabra clave **all** se emplea con la operación **killed**, ésta concuerda si todos los PTC del caso de prueba han dejado de existir. También concuerda si no se ha creado ningún PTC.

Cuando la palabra clave **any** se emplea con la operación **killed**, ésta concuerda si al menos un PTC ha dejado de existir. En los otros casos la concordancia no es satisfactoria.

EJEMPLO:

```
var MyPTCType ptc := MyPTCType.create alive; // crea un componente de prueba tipo alive
timer T(10.0);                               // crea un temporizador
T.start;                                     // activa el temporizador
ptc.start(MyTestBehavior());                 // arranca la ejecución de una función en
                                             // el PTC
alt {
[] ptc.killed {                              // si el PTC fue eliminado durante la
                                             // ejecución …
   T.stop;                                   // … detener el temporizador y …
   setverdict(inconc);                       // … poner el veredicto a 'no concluyente'
   }
[] ptc.done {                                // si el PTC terminó normalmente …
   T.stop;                                   // … detener el temporizador y …
   ptc.start(AnotherFunction());             // … activar otra función en el PTC
   }
[] T.timeout {                               // si se alcanza el fin de temporización
                                             // antes de que se detenga el PTC
   ptc.kill;                                 // … eliminar el PTC y …
   setverdict(fail);                         // … poner el veredicto a 'fallo'
   }
}
```

22.12 Utilización de matrices de componentes

Las operaciones **create**, **connect**, **start**, **stop** y **kill** no funcionan directamente en matrices de componentes. Hay que proporcionar un elemento específico de la matriz como parámetro para estas operaciones y utilizar una matriz de referencias de componentes. El elemento de la matriz pertinente se asigna al resultado de la operación **create**.

EJEMPLO:

```
// Ejemplo de modelización de operaciones crear, conectar y operar para matrices de
// componentes, utilizando 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 MyPTCType1    MyPtc [11];
     :
    for (i:= 0; i<=10; i:=i+1)
    {
          MyPtc [i] := MyPTCType1.create;
          connect(self:PtcCoordination, MyPtc [i]:MtcCoordination);
          MyPtc [i].start(MyPtcBehaviour());
    }
     :
}
```

22.13 Utilización de las palabras clave any y all con componentes

Las palabras clave **any** y **all** se pueden utilizar con operaciones de configuración como se indica en el cuadro 16.

Cuadro 16/Z.140 – Utilización de any y all con componentes

Operación	Palabra clave permitida		Ejemplo	Comentarios
	any (véase la nota)	**all** (véase la nota)		
create				
start				
running	Sí, pero sólo desde el MTC	Sí, pero sólo desde el MTC	**any component.running;**	¿Hay algún PTC ejecutando el comportamiento de prueba?
			all component.running;	¿Todos los PTC están ejecutando el comportamiento de prueba?
alive	Sí, pero sólo desde el MTC	Sí, pero sólo desde el MTC	**any component.alive;**	¿Hay algún PTC activo?
			all component.alive;	¿Todos los PTC están activos?
done	Sí, pero sólo desde el MTC	Sí, pero sólo desde el MTC	**any component.done;**	¿Hay algún PTC que haya completado la ejecución?
			all component.done;	¿Todos los PTC han completado la ejecución?
killed	Sí, pero sólo desde el MTC	Sí, pero sólo desde el MTC	**any component.killed;**	¿Hay algún PTC que haya dejado de existir?
			all component.killed;	¿Todos los PTC han dejado de existir?
stop		Sí, pero sólo desde el MTC	**all component.stop;**	Detener el comportamiento de todos los PTC.
kill		Sí, pero sólo desde el MTC	**all component.kill;**	Eliminar todos los PTC: dejan de existir
NOTA – **any** y **all** se refieren únicamente a los PTC: el MTC no se tiene en cuenta.				

23 Operaciones de comunicación

23.0 Consideraciones generales

La notación TTCN-3 soporta comunicaciones por *mensajes* y comunicaciones por *procedimientos*. También permite examinar el elemento de cabeza de la fila de un puerto de entrada y controlar el acceso a los puertos mediante *operaciones de control*.

Cuadro 17/Z.140 – Las operaciones de comunicación de TTCN-3

Operaciones de comunicación			
Operación de comunicación	**Palabra clave**	**Se puede utilizar en puertos basados en mensajes**	**Se puede utilizar en puertos basados en procedimientos**
Comunicación por mensajes			
Enviar mensaje	**`send`**	Sí	
Recibir mensaje	**`receive`**	Sí	
Aceptar mensaje	**`trigger`**	Sí	
Comunicación por procedimientos			
Invocar solicitud de procedimiento	**`call`**		Sí
Aceptar solicitud de procedimiento de entidad distante	**`getcall`**		Sí
Responder a solicitud de procedimiento de entidad distante	**`reply`**		Sí
Plantear excepción (a una solicitud aceptada)	**`raise`**		Sí
Tratar respuesta de una solicitud anterior	**`getreply`**		Sí
Reconocer excepción (de entidad llamada)	**`catch`**		Sí
Examinar elemento de cabeza en las colas de puertos de entrada			
Comprobar mensaje/solicitud/excepción/respuesta recibidos	**`check`**	Sí	Sí
Operaciones de control			
Liberar la cola de puertos	**`clear`**	Sí	Sí
Liberar cola y habilitar la transmisión y recepción en un puerto	**`start`**	Sí	Sí
Inhabilitar las operaciones de transmisión y recepción que corresponden a un puerto	**`stop`**	Sí	Sí
Inhabilitar las operaciones de transmisión y recepción que corresponden a nuevos mensajes/solicitudes	**`halt`**	Sí	Sí

23.1 Formato general de las operaciones de comunicación

23.1.0 Consideraciones generales

Las operaciones **`send`** y **`call`** y otras similares se utilizan para intercambiar información entre componentes de prueba, o entre el SUT y los componentes de prueba. Para explicar el formato general dividimos estas operaciones en dos grupos:

a) un componente envía un mensaje (operación **`send`**), solicita un procedimiento (operación **`call`**), responde a una solicitud aceptada (operación **`reply`**) o plantea una excepción (operación **`raise`**). Estas acciones se denominan colectivamente *operaciones de emisión*;

b) un componente recibe un mensaje (operación **`receive`**), acepta un mensaje (operación **`trigger`**), acepta una solicitud de procedimiento (operación **`getcall`**), recibe una respuesta para un procedimiento solicitado anteriormente (operación **`getreply`**) o reconoce una excepción (operación **`catch`**). Estas acciones se denominan colectivamente *operaciones de recepción*.

23.1.1 Formato general de las operaciones de emisión

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

La parte emisión:

- especifica en qué puerto se ha de producir la operación especificada;
- define la solicitud de mensaje o procedimiento que se ha de transmitir;
- proporciona una parte de la dirección (facultativa) que identifica de manera única a uno o varios interlocutores a los que se ha de enviar un mensaje, solicitud, respuesta o excepción.

El nombre de puerto, el nombre de operación y el valor estarán presentes en todas las operaciones de emisión. La parte de la dirección (indicado por la palabra clave **to**) es facultativa y sólo es necesario especificarla para las conexiones uno a varios, donde:

- se utiliza una comunicación unidifusión y se identifica explícitamente una entidad receptora;
- se utiliza una comunicación multidifusión y ha de identificarse explícitamente un conjunto de entidades receptoras;
- se utiliza una comunicación de tipo radiodifusión y han de designarse todas las entidades conectadas al puerto especificado.

NOTA – Los términos comunicación "unidifusión", "multidifusión" y "radiodifusión" se utilizan en relación con la comunicación de un puerto. Esto significa que sólo es posible designar uno, varios o todos los componentes conectados al puerto especificado. Estos términos también pueden emplearse para los puertos relacionados, en cuyo caso, se puede llegar a una, varias o todas las entidades dentro del SUT a través del puerto relacionado especificado.

EJEMPLO 1:

Parte emisión			**Parte de tratamiento de respuesta y excepciones (facultativa)**
Puerto y operación	**Parte con el valor**	**Parte de la dirección (facultativa)**	
`MyP1.`**`send`**	`(MyVariable + YourVariable - 2)`	**`to`** `MyPartner;`	

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

En la parte de tratamiento de respuesta y excepciones de la operación call se utilizan las operaciones **`getreply`** y **`catch`** para proporcionar la funcionalidad requerida.

EJEMPLO 2:

Parte emisión			**Parte de tratamiento de respuesta y excepciones (facultativa)**
Puerto y operación	**Parte con el valor**	**Parte de la dirección (facultativa)**	

`MyP1.`**`call`**	`(MyProc:{MyVar1})`		`{`
			`  [] MyP1.`**`getreply`**`(MyProc:{MyVar2}) {}`
			`  [] MyP1.`**`catch`**`(MyProc, ExceptionOne) {}`
			`}`

23.1.2 Formato general de las operaciones de recepción

Una operación de recepción consiste en una parte *recepción* y una parte *asignación* (facultativa).

La parte recepción:

a) especifica en qué puerto se ha de ejecutar la operación;

b) define una parte de concordancia que especifica la entrada aceptable o concordante con la instrucción;

c) proporciona una expresión de dirección (facultativa) que identifica de manera única al interlocutor (en el caso de conexiones de uno a muchos).

Hay que incluir el nombre de puerto, el nombre de operación y el valor de todas las operaciones de recepción. La identificación del interlocutor (indicado por la palabra clave **`from`**) es facultativa y sólo es necesaria para identificar explícitamente la entidad receptora en los casos de conexiones de uno a muchos.

La parte de asignación de una operación receptora es facultativa. En el caso de puertos basados en mensajes se utiliza cuando se requiere almacenar mensajes recibidos. En el caso de puertos basados en procedimientos se utiliza para almacenar los parámetros **in** e **inout** de una solicitud aceptada, para almacenar el valor devuelto o para almacenar excepciones. En la parte de asignación es necesaria una tipificación rigurosa; por ejemplo, la variable que se utiliza para almacenar un mensaje tiene que ser del mismo tipo que el mensaje entrante.

La parte de asignación también se puede utilizar para asignar la dirección **sender** de un mensaje, una excepción, una **reply** o una **call** a una variable. Es útil en el caso de conexiones de uno a muchos, entre otras cosas porque se puede recibir el mismo mensaje o solicitud de diferentes componentes, pero siempre hay que devolver el mensaje, la respuesta o la excepción al componente emisor original.

EJEMPLO:

Parte recepción				**Parte asignación (facultativa)**		
Puerto y operación	**Parte de concordancia**	**Expresión de dirección (facultativa)**		**Asignación de valor (facultativa)**	**Asignación de valor de parámetro (facultativa)**	**Asignación de valor de emisor (facultativa)**
MyP1.**getreply**	(AProc:{?} **value** 5)		->		**param** (V1)	**sender** APeer

Parte recepción				**Parte asignación (facultativa)**		
Puerto y operación	**Parte de concordancia**	**Expresión de dirección (facultativa)**		**Asignación de valor (facultativa)**	**Asignación de valor de parámetro (facultativa)**	**Asignación de valor de emisor (facultativa)**
MyP2.**receive**	(MyTemplate(5,7))	**from** APeer	->	**value** MyVar		

23.2 Comunicación por mensajes

23.2.0 Consideraciones generales

La comunicación por mensajes consiste en un intercambio asíncrono de mensajes. La comunicación por mensajes no es bloqueante en la operación **send**, como se ilustra en la figura 11, el tratamiento en el EMISOR continúa inmediatamente después de la operación **send**. El RECEPTOR se bloquea en la operación **receive** hasta que se procese el mensaje recibido.

La notación TTCN-3 completa la operación **receive** con una operación **trigger** que filtra, conforme a determinados criterios de concordancia, un tren de mensajes recibidos en un determinado puerto de entrada. Los mensajes en cabeza de la cola que no satisfacen los criterios de concordancia serán simplemente retirados del puerto.

Figura 11/Z.140 – El proceso asíncrono de las operaciones send y receive

23.2.1 La operación Send (enviar)

23.2.1.0 Consideraciones generales

La operación **send** se utiliza para colocar un mensaje en un puerto de salida. Se puede especificar por referencia a una plantilla definida, pero también se puede definir como una plantilla en línea. Si el mensaje se define en línea, se utilizará la parte del tipo facultativo en caso de ambigüedad con respecto al tipo del mensaje que se envía.

La operación **send** sólo se utilizará en puertos por mensajes (o mixtos) y es condición que el tipo de la plantilla que se envía esté en la lista de tipos salientes, en la definición del tipo del puerto.

EJEMPLO:

```
MyPort.send(MyTemplate(5,MyVar));      // Envía la plantilla MyTemplate con los parámetros
                                       // efectivos 5 y MyVar a través de MyPort.

MyPort.send(5);                        // Envía el valor entero 5 (que representa
                                       // una plantilla en línea)
```

23.2.1.1 Transmisiones de tipo unidifusión, multidifusión o radiodifusión

La notación TTCN-3 soporta comunicación de tipo unidifusión, multidifusión o radiodifusión. La determinación del mecanismo de comunicación que ha de utilizarse puede llevarse a cabo mediante la cláusula **to** facultativa en la operación **send**. La cláusula **to** puede omitirse en las conexiones uno a uno que emplean la comunicación unidifusión y la estructura del sistema de prueba puede determinar de manera inequívoca el receptor del mensaje. Las conexiones uno a muchos han de incluir una cláusula **to**.

Si la cláusula **to** designa un solo interlocutor, se especifica una comunicación unidifusión. Si la cláusula **to** incluye una lista de interlocutores, se utiliza la comunicación multidifusión. Si la cláusula **to** incluye la palabra clave **all component** se define la comunicación de tipo radiodifusión.

EJEMPLO:

```
MyPort.send(charstring:"My string") to MyPartner;
                                       // Envía la cadena "My string" a un componente
                                       // que tiene una referencia almacenada en la
                                       // variable MyPartner
MyPCO.send(MyVariable + YourVariable – 2) to MyPartner;
                                       // Envía el resultado de la expresión aritmética
                                       // a MyPartner.
MyPCO2.send(MyTemplate) to (MyPeerOne, MyPeerTwo);
                                       / Especifica una comunicación multidifusión,
                                       // donde el valor MyTemplate se envía a las
                                       // dos referencias de componente almacenadas
                                       // en las variables MyPeerOne y MyPeerTwo.

MyPCO3.send(MyTemplate) to all component;
                                       //Comunicación de tipo radiodifusión: el valor
                                       // Mytemplate se envía a todos los componentes
                                       // que pueden designarse por este puerto.
                                       // Si MyPCO3 es un puerto relacionado, los
                                       // componentes pueden residir dentro del SUT.
```

23.2.2 La operación Receive (recibir)

23.2.2.0 Consideraciones generales

La operación **receive** se utiliza para recibir un mensaje de una cola de puerto de mensajes entrantes. Se puede especificar por referencia a una plantilla definida, pero también se puede definir como una plantilla en línea. Si el mensaje se define en línea, la parte de tipo facultativo ha de estar presente en caso de que el tipo del mensaje que se recibe sea ambiguo. La operación **receive** sólo se utilizará en puertos por mensajes (o mixtos) y es condición que el tipo del valor que se recibe esté en la lista de tipos entrantes, en la definición del tipo del puerto.

La operación **receive** retira el mensaje en cabeza de la cola de puerto entrante asociada solamente si ese mensaje satisface todos los criterios de concordancia asociados a la operación **receive**. No se producirá ninguna vinculación de los valores entrantes con los términos de la expresión o con la plantilla.

Si no hay concordancia no se retirará el primer mensaje de la cola del puerto. Entonces, si la operación **receive** se utiliza como alternativa de una instrucción **alt** y el resultado es fallido, la ejecución del caso de prueba continuará con la siguiente alternativa de la instrucción **alt**.

Los criterios de concordancia dependen del tipo y el valor del mensaje que se ha de recibir. Este tipo y este valor se determinan mediante el argumento de la operación **receive**, es decir, se pueden deducir de la plantilla definida o pueden especificarse en línea. El campo de tipo facultativo en los criterios de concordancia de la operación **receive** se utilizará para evitar toda ambigüedad del tipo del valor que se recibe.

NOTA 1 – Los atributos de codificación también intervienen en la concordancia, implícitamente, impidiendo que el decodificador produzca un valor abstracto a partir de un mensaje recibido que no ha sido codificado en la forma especificada por los atributos.

Cuando se realiza una comunicación uno a muchos se puede restringir la operación **receive** a un determinado interlocutor, utilizando la palabra código **from**.

EJEMPLO 1:

```
MyPort.receive(MyTemplate(5, MyVar));  // Concordancia con un mensaje que satisface las
                                       // condiciones de la plantilla MyTemplate en el
                                       // el puerto MyPort.

MyPort.receive(A<B);                   // Concordancia con un valor booleano que depende
                                       // de resultado de A<B

MyPort.receive(integer:MyVar);         // Concordancia de un entero con el valor de MyVar en
                                       // el puerto MyPort
MyPort.receive(MyVar);                 // Alternativa del ejemplo anterior

MyPort.receive(charstring:"Hello")from MyPeer;     // Concordancia con la cadena de
                                                   // caracteres "Hello" de MyPeer
```

Si la concordancia es satisfactoria, el valor que se retira de la cola del puerto se puede almacenar en una variable. Se indica mediante el símbolo '->' y la palabra clave **value**.

También es posible extraer y almacenar la referencia del componente o la dirección del emisor de un mensaje: se indica mediante la palabra clave **sender**.

NOTA 2 – Cuando el mensaje se recibe por un puerto conectado, en la siguiente palabra clave **sender** se almacena únicamente la referencia del componente, pero el sistema de prueba guardará internamente también el nombre del componente, si lo hubiere (que se ha de utilizar para efectos de registro).

EJEMPLO 2:

```
MyPort.receive(MyType:?) -> value MyVar; // El valor del mensaje recibido se asigna a MyVar.

MyPort.receive(A<B) -> sender MyPeer     // La dirección del emisor se asigna a MyPeer

MyPort.receive(MyTemplate:{5, MyVarOne}) -> value MyVarTwo sender MyPeer;
// El valor del mensaje recibido se almacena en MyVarTwo y la dirección del emisor
// se almacena en MyPeer.
```

23.2.2.1 Recepción de cualquier mensaje

Si la operación **receive** no tiene una lista de argumentos para los criterios de concordancia de tipo y valor del mensaje que se ha de recibir, el mensaje en cabeza de la cola de puerto entrante (en su caso) será retirado si se cumplen los demás criterios de concordancia.

Los mensajes recibidos como resultado de una operación *ReceiveAnyMessage* no se asignarán a una variable.

EJEMPLO:

```
MyPort.receive;                        // Retira el primer valor de MyPort.

MyPort.receive from MyPartner;         // Retira el primer mensaje de MyPort si el emisor
                                       // es MyPeer

MyPort.receive -> sender MySenderVar;  // Retira el primer mensaje de MyPort y asigna
                                       // la dirección del emisor MySenderVar
```

23.2.2.2 Recepción en cualquier puerto

Para recibir un mensaje en cualquier puerto se utilizarán las palabras clave **any** y **port**.

EJEMPLO:

```
any port.receive(MyMessage);
```

23.2.3 La operación Trigger (aceptar mensaje)

23.2.3.0 Consideraciones generales

La operación **trigger** retira el mensaje de cabeza de la cola del puerto de entrada asociado. Si este mensaje satisface los criterios de concordancia, la operación **trigger** se desarrolla como una operación **receive**. Cuando no hay concordancia se retira simplemente el mensaje de la cola. La operación **trigger** sólo se utilizará en puertos por mensajes (o mixtos) y es condición que el tipo del valor que se recibe esté en la lista de tipos entrantes, en la definición del tipo del puerto.

NOTA – La nota 1 a 22.2.2.0 vale igualmente para la operación **trigger**.

La operación **trigger** se puede utilizar como una instrucción autónoma en una descripción de comportamiento. Entonces se considera como una expresión simplificada de una instrucción **alt** con una sola alternativa, es decir, tiene

semántica bloqueante y, por tanto, permite esperar el siguiente mensaje que concuerda con la plantilla o el valor especificados en la cola.

EJEMPLO 1:

```
MyPort.trigger(MyType:?);
// Especifica que la operación validará la recepción del primer mensaje observado
// del tipo MyType con un valor arbitrario en el puerto MyPort.
```

En la operación **trigger** hay que proporcionar el nombre del puerto, los criterios de concordancia de tipo y valor, una restricción **from** facultativa (selección del interlocutor) y una asignación (facultativa) del mensaje concordante y del componente de emisor a variables.

EJEMPLO 2:

```
MyPort.trigger(MyType:?) from MyPartner;
// Valida la recepción del primer mensaje de tipo MyType en el puerto MyPort,
// recibido de MyPartner.

MyPort.trigger(MyType:?) from MyPartner -> value MyRecMessage;
// Muy similar al ejemplo anterior, pero además el mensaje que satisface todos los
// criterios de concordancia se almacena en la variable MyRecMessage.

MyPort.trigger(MyType:?) -> sender MyPartner;
// Muy similar al ejemplo anterior, pero además se extrae la referencia del componente
// de emisor y se almacena en la variable MyPartner.

MyPort.trigger(integer:?) -> value MyVar sender MyPartner;
// Valida la recepción de un valor entero arbitrario, que también se almacenará
// en la variable MyVar. La referencia del componente de emisor se almacenará en
// la variable MyPartner.
```

23.2.3.1 Aceptar cualquier mensaje

Una operación **trigger** sin lista de argumentos valida la recepción de cualquier mensaje; tiene pues el mismo significado que una operación receive para cualquier mensaje. Los mensajes recibidos como resultado de esta operación *TriggerOnAnyMessage* no se asignarán a una variable.

EJEMPLO:

```
MyPort.trigger;

MyPort.trigger from MyPartner;

MyPort.trigger -> sender MySenderVar;
```

23.2.3.2 Aceptar en cualquier puerto

Para validar la recepción de un mensaje en cualquier puerto se utilizarán las palabras clave **any** y **port**.

EJEMPLO:

```
any port.trigger
```

23.3 Comunicación por procedimientos

23.3.0 Consideraciones generales

El principio de esta forma de comunicación es solicitar procedimientos en entidades distantes. La notación TTCN-3 soporta la comunicación por procedimientos *bloqueante* y *no bloqueante*. La primera bloquea en el lado llamante y el lado llamado, y la segunda sólo bloquea en el lado llamado. Es necesario especificar firmas de procedimientos para esta comunicación no bloqueante, conforme a las reglas de la cláusula 13.

En la figura 12 se ha representado el esquema de la comunicación por procedimientos bloqueante. La parte LLAMANTE utiliza la operación **call** para solicitar un procedimiento distante en la parte LLAMADA. Ésta acepta la solicitud realizando una operación **getcall** y reacciona utilizando sea una operación **reply** para contestar la llamada, o planteando una excepción (operación **raise**). La parte LLAMANTE utiliza las operaciones **getreply** o **catch** para tratar la respuesta o la excepción. Las líneas discontinuas de la figura 12 representan el bloqueo de la parte LLAMANTE y la parte LLAMADA.

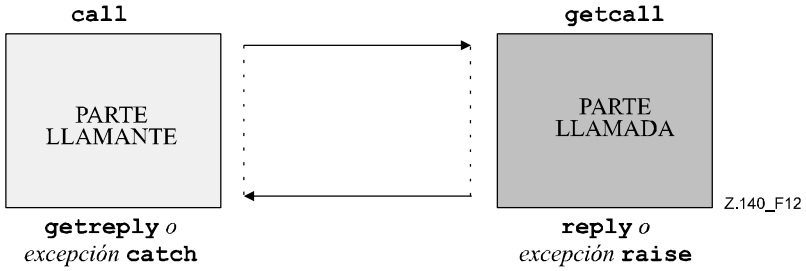


Figura 12/Z.140 – Ilustración de la comunicación por procedimientos bloqueante

En la figura 13 se ha representado el esquema de la comunicación por procedimientos no bloqueante. La parte LLAMANTE utiliza la operación **call** para solicitar un procedimiento distante en la parte LLAMADA y continúa su ejecución, es decir, no espera una respuesta ni una excepción. La parte LLAMADA acepta la solicitud realizando una operación **getcall** y ejecuta el procedimiento solicitado. Si la ejecución no es satisfactoria, puede plantear una excepción para informar a la parte LLAMANTE. La parte LLAMANTE utiliza la operación **catch** en una instrucción **alt** para tratar la excepción. Las líneas discontinuas de la figura 13 representan el bloqueo de la parte LLAMADA hasta que termina el tratamiento de la solicitud y el posible planteamiento de una excepción.

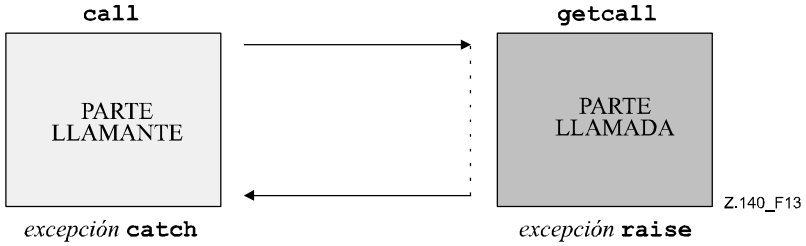


Figura 13/Z.140 – Ilustración de la comunicación por procedimientos no bloqueante

23.3.1 La operación Call (solicitud de procedimiento)

23.3.1.0 Consideraciones generales

La operación **call** se utiliza para especificar que un componente de prueba solicita un procedimiento en el SUT o en otro componente de prueba. Sólo se utilizará en puertos de comunicación por procedimientos (o mixtos). En la definición de tipo del puerto en que se realiza esta operación hay que incluir el nombre del procedimiento, en la lista **out** o **inout**, es decir, hay que autorizar la solicitud del procedimiento en este puerto.

La información a transmitir en la parte de emisión de la operación **call** es una firma que podrá definirse como plantilla de firma o en línea. Todos los parámetros **in** e **inout** de la firma han de tener un valor específico, lo que significa que no está permitido utilizar, por ejemplo, el mecanismo de concordancia *AnyValue*.

Los argumentos de la firma de la operación **call** no se utilizan para obtener nombres de variables para parámetros **out** e **inout**. El valor de respuesta del procedimiento y los valores de los parámetros **out** e **inout** se asignarán a las variables explícitamente en la parte de tratamiento de respuesta y excepciones de la operación **call**, mediante las operaciones **getreply** y **catch**. Este proceso permite utilizar plantillas de firmas en las operaciones **call** como se utilizan plantillas para los tipos.

EJEMPLO 1:

```
// Dado …
signature MyProc (out integer MyPar1, inout boolean MyPar2);
 :
// solicitud de MyProc
MyPort.call(MyProc:{ -, MyVar2}) {     // plantilla de firma en línea para la solicitud
                                       // de MyProc
[] MyPort.getreply(MyProc:{?, ?}) { }
}

// … y otra solicitud de MyProc
MyPort.call(MyProcTemplate) {    // con la plantilla de firma para la solicitud de MyProc
    [] MyPort.getreply(MyProc:{?, ?}) { }
}
```

Si es una comunicación de uno a muchos hay que especificar de forma única el interlocutor mediante la palabra clave **to**.

EJEMPLO 2:

```
MyPort.call(MyProcTemplate) to MyPeer {      // solicitud de MyProc en MyPeer
[] MyPort.getreply(MyProc:{?, ?}) { }
}
```

23.3.1.1 Tratamiento de respuestas y excepciones a la operación Call

Cuando se trata de comunicación por procedimientos no bloqueante (véase 23.3.1.4) el tratamiento de las excepciones a las operaciones **call** se lleva a cabo mediante operaciones **catch** (véase 23.3.6) que se emplean como alternativas en las instrucciones **alt**.

Si se utiliza la opción **nowait** (véase 23.3.1.2), las operaciones **getreply** (véase 23.3.4) y **catch** (véase 23.3.6) se utilizan como alternativas en instrucciones **alt** para el tratamiento de respuestas y excepciones a operaciones **call**.

Cuando se trata de comunicación por procedimientos bloqueante, la parte de respuestas y excepciones de la operación call permite realizar este tratamiento, mediante las operaciones **getreply** (véase 23.3.4) y **catch** (véase 23.3.6).

La parte de tratamiento de respuestas y excepciones de la operación **call** es similar al cuerpo de una instrucción **alt**. Define una serie de alternativas que describen las respuestas y excepciones posibles a la solicitud. Para seleccionar las alternativas sólo se utilizarán operaciones **getreply** y **catch** para el procedimiento solicitado. Las operaciones **getreply** y **catch** tratarán únicamente las respuestas y excepciones formuladas por el procedimiento solicitado. No se autoriza la utilización de ramales **else** ni la invocación de **altsteps**.

Si es necesario se puede habilitar/inhabilitar la alternativa utilizando una expresión **boolean** entre los corchetes '[ ]'.

La parte de tratamiento de respuesta y excepciones de la operación call se ejecuta como una instrucción **alt** sin ninguna opción por defecto activa. Siendo así, hay un estado puntual correspondiente que incluye toda la información necesaria para evaluar las guardas booleanas (facultativas), que puede incluir el primer elemento (en su caso) del puerto en el que se solicita el procedimiento y que puede incluir una excepción de temporización generada por el temporizador (facultativo) que supervisa la solicitud (véase 23.3.1.2).

La evaluación de las guardas de expresiones booleanas de alternativas en la parte de tratamiento de respuesta y excepciones puede tener efectos secundarios. Para evitar efectos secundarios imprevistos se aplicarán las mismas reglas de las guardas booleanas en instrucciones **alt** (véase 20.1.1).

EJEMPLO:

```
// Dado
signature MyProc3 (out integer MyPar1, inout boolean MyPar2) return MyResultType
    exception (ExceptionTypeOne, ExceptionTypeTwo);
 :

// Solicitud de MyProc3
MyPort.call(MyProc3:{ -, true }) to MyPartner {

  [] MyPort.getreply(MyProc3:{?, ?}) -> value MyResult param (MyPar1Var,MyPar2Var) { }

  [] MyPort.catch(MyProc3, MyExceptionOne) {
   setverdict(fail);
   stop;
   }
  [] MyPort.catch(MyProc3, ExceptionTypeTwo : ?) {
   setverdict(inconc);
   }
  [MyCondition] MyPort.catch(MyProc3, MyExceptionThree) { }
}
```

23.3.1.2 Tratamiento de excepciones de temporización a la operación Call

La operación **call** puede incluir una condición de temporización (es facultativo): es un valor o una constante explícitos de tipo **float** que determina un periodo contado desde el principio de la operación **call**, transcurrido el cual el sistema de prueba producirá una excepción **timeout**. Si no hay ningún valor de temporización en la operación **call** no se producirán excepciones **timeout**.

EJEMPLO 1:

```
MyPort.call(MyProc:{5,MyVar}, 20E-3) {

  [] MyPort.getreply(MyProc:{?, ?}) { }

  [] MyPort.catch(timeout) {                // excepción de temporización a 20 ms
   setverdict(fail);
   stop;
     }
}
```

La utilización de la palabra clave **nowait** en lugar de un valor excepción de temporización en una operación **call** permite solicitar un procedimiento y continuar sin esperar una respuesta, una excepción planteada por el procedimiento o una excepción de temporización.

EJEMPLO 2:

```
MyPort.call(MyProc:{5, MyVar}, nowait);      // El componente de prueba llamante continúa la
                                             // ejecución sin esperar a que termine MyProc
```

Si se utiliza la palabra clave **nowait** habrá que utilizar las operaciones **getreply** o **catch** en una instrucción **alt** ulterior para retirar de la cola una posible respuesta o excepción del procedimiento solicitado.

23.3.1.3 Solicitud de procedimientos bloqueantes sin devolución de valor, sin parámetros out, parámetros inout y excepciones

Puede definirse un procedimiento bloqueante sin valores de respuesta ni parámetros **out** e **inout** y que no plantea excepciones. La operación call para estos procedimientos también ha de tener una parte de tratamiento de respuesta y excepciones para tratar el bloqueo uniformemente.

EJEMPLO:

```
// Dado …
signature MyBlockingProc (in integer MyPar1, in boolean MyPar2);
 :
// una solicitud de MyBlockingProc
MyPort.call(MyBlockingProc:{ 7, false }) {
  [] MyPort.getreply( MyBlockingProc:{ -, - } ) { }
}
```

23.3.1.4 Solicitud de procedimientos no bloqueantes

Los procedimientos no bloqueantes no tienen parámetros **out** e **inout**, y no tienen ningún valor de respuesta; su carácter no bloqueante se indica en la correspondiente definición de firma mediante la palabra código **noblock**.

La operación **call** para un procedimiento no bloqueante no tendrá una parte de tratamiento de respuesta y excepciones, no planteará excepciones de temporización y no utilizará la palabra código **nowait**.

Hay que retirar de la cola las posibles excepciones planteadas por procedimientos no bloqueantes, utilizando operaciones **catch** en las instrucciones **alt o interleave** ulteriores.

23.3.1.5 Solicitudes de procedimientos de tipo unidifusión, multidifusión y radiodifusión

Como en el caso de la operación **send**, la notación TTCN-3 también soporta solicitudes de procedimientos de tipo unidifusión, multidifusión y radiodifusión. El método es el mismo que se describe en 23.2.1.1, es decir, el argumento de la cláusula **to** de una operación **call** representa, para las solicitudes de tipo unidifusión, la dirección de una entidad receptora (o puede omitirse en las conexiones uno a uno), para las solicitudes de tipo multidifusión, una lista de direcciones de un conjunto de receptores y para las solicitudes de tipo radiodifusión, la palabra clave **all component**. En las conexiones uno a uno puede omitirse la cláusula **to** puesto que la estructura del sistema identifica de manera única a la entidad receptora.

El tratamiento de las respuestas y excepciones de una operación **call** unidifusión bloqueante o no bloqueante se explicó en 23.3.1.1 a 23.3.1.4. Una operación **call** multidifusión o radiodifusión puede dar lugar a varias respuestas y excepciones de distintos interlocutores.

En la operación **call** multidifusión o radiodifusión de un procedimiento no bloqueante, todas las excepciones que pueden ser formuladas por los distintos interlocutores pueden ser tratadas en instrucciones **catch**, **alt** o **interleave** subsiguientes.

La operación **call** multidifusión o radiodifusión de un procedimiento bloqueante tiene dos opciones. En la parte de tratamiento de respuestas y excepciones de la operación **call** se tramita sólo una respuesta o excepción y a continuación las respuestas y excepciones adicionales pueden tratarse en instrucciones **alt** o **interleave** subsiguientes. Opcionalmente, se tratan varias respuestas y excepciones utilizando instrucciones repeat en uno o más de los bloques de instrucciones y declaraciones de la parte de tratamiento de respuestas y excepciones de la operación solicitada: la ejecución de una instrucción repeat origina que se evalúe nuevamente el cuerpo de la solicitud.

NOTA – En la segunda opción, el usuario ha de tratar las repeticiones.

EJEMPLO 1:

```
var boolean first:= true;
MyPort.call(MyProc:{5,MyVar}, 20E-3) to (MyPeerOne, MyPeerTwo)
{   // Solicitud multidifusión de MyProc
    // Trata la respuesta de MyPeerOne
    [first] MyPort.getreply(MyProc:{?, ?}) from MyPeerOne {
          if (first) { first := false; repeat; }
          :
    }
    // Trata la respuesta de MyPeerTwo
    [first] MyPort.getreply(MyProc:{?, ?}) from MyPeerTwo {
          if (first) { first := false; repeat; }
          :
    }
    [] MyPort.catch(timeout) {          // excepción de fin de temporización tras 20ms
          setverdict(fail);
          stop;
    }
}

alt {
  [] MyPort.getreply(MyProc:{?, ?}) {  // Trata las otras respuestas a la solicitud
                                       // de tipo radiodifusión
          repeat
  }
}
```

En la operación **call** multidifusión o radiodifusión de un procedimiento bloqueante en la que se emplea la palabra clave **nowait**, todas las respuestas y excepciones han de tratarse en instrucciones **alt** o **interleave** subsiguientes.

EJEMPLO 2:

```
MyPort.call(MyProc:{5,MyVar}) to (MyPeer1, MyPeer2) nowait;
// Solicitud multidifusión de MyProc

interleave {
  [] MyPort.getreply(MyProc:{?, ?}) from MyPeer1 { }     // Trata la respuesta de MyPeer1
  [] MyPort.getreply(MyProc:{?, ?}) from MyPeer2 { }     // Trata la respuesta de MyPeer2
}
```

23.3.2 La operación Getcall (aceptar solicitud)

23.3.2.0 Consideraciones generales

La operación **getcall** se utiliza para especificar que un componente de prueba acepta una solicitud del SUT o de otro componente de prueba. La operación **getcall** se utilizará solamente en puertos basados en procedimientos (o mixtos) y la condición para aceptar el procedimiento es que la firma de la solicitud correspondiente esté incluida en la lista de procedimientos entrantes permitidos de la definición del tipo de puerto.

La operación **getcall** retira la primera solicitud de la cola del puerto entrante solamente si se cumplen los criterios de concordancia asociados con la operación **getcall** (la firma de la solicitud procesada y el interlocutor). Los criterios de concordancia para la firma se pueden especificar en línea y también se pueden obtener de una plantilla de firma.

Es posible restringir una operación **getcall** a un determinado interlocutor en el caso de conexiones de uno a muchos y esta restricción se indicará mediante la palabra clave **from**.

EJEMPLO 1:

```
MyPort.getcall(MyProc; MyProcTemplate(5, MyVar));  // acepta una solicitud de MyProc
                                                   // en MyPort

MyPort.getcall(MyProc:{5, MyVar}) from MyPeer;     // acepta una solicitud de MyProc
                                                   // en MyPort de MyPeer
```

El argumento de firma de la operación **getcall** no se ha de utilizar para transferir nombres de variables 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.

La parte de asignación (facultativa) de la operación **getcall** comprende la asignación de valores de parámetros **in** e **inout** a variables y la extracción de la dirección del componente llamante. La parte de asignación de valor no debe emplearse con la operación **getcall**. La palabra clave **param** se utiliza para obtener los valores de parámetros de una llamada.

La palabra clave **sender** se utiliza si es necesario extraer la dirección del emisor (por ejemplo, para direccionar una **respuesta** o excepción a la parte llamante en una configuración de uno a muchos).

EJEMPLO 2:

```
MyPort.getcall(MyProc:{?, ?}) from MyPartner -> param (MyPar1Var, MyPar2Var);
// Los valores de parámetros in o inout de MyProc se asignan a MyPar1Var y MyPar2Var.

MyPort.getcall(MyProc:{5, MyVar}) -> sender MySenderVar;
// Acepta una llamada de MyProc en MyPort con los parámetros in o inout 5 y MyVar.
// Se extrae la dirección de la parte llamante y se almacena en MySenderVar.

// Los siguientes ejemplos de getcall ilustran las posibilidades de utilización de
// atributos de concordancia y la omisión de partes facultativas que no siempre son
// importantes para especificar la prueba.

MyPort.getcall(MyProc:{5, MyVar}) -> param(MyVar1, MyVar2) sender MySenderVar;

MyPort.getcall(MyProc:{5, ?}) -> param(MyVar1, MyVar2);

MyPort.getcall(MyProc:{?, MyVar}) -> param( - , MyVar2);
// El valor del primer parámetro inout no es importante o no se utiliza

// Los siguientes ejemplos ilustran las posibilidades para asignar valores de parámetros
// in e inout a variables. Se supone que ésta es la firma del procedimiento que se va a
// solicitar:

signature MyProc2(in integer A, integer B, integer C, out integer D, inout integer E);

MyPort.getcall(MyProc2:{?, ?, 3, - , ?}) -> param (MyVarA, MyVarB, - , -, MyVarE);
// Los parámetros A, B y E se asignan a las variables MyVarA, MyVarB,
// y MyVarE. No es necesario tener en cuenta el parámetro out D.

MyPort.getcall(MyProc2:{?, ?, 3, -, ?}) -> param (MyVarA:= A, MyVarB:= B, MyVarE:= E);
// Notación alternativa para la asignación de valores de los parámetros in e inout
// a variables. Los nombres en la lista de asignaciones son los que se utilizan en
// la firma de MyProc2

MyPort.getcall(MyProc2:{1, 2, 3, -, *}) -> param (MyVarE:= E);
// El valor del parámetro inout es el único necesario para proseguir la ejecución
// del caso de prueba
```

23.3.2.1 Aceptar cualquier solicitud

Una operación **getcall** sin lista de argumentos para los criterios de concordancia de firmas retira la primera solicitud de la cola de puerto entrante (en su caso) si se cumplen los demás criterios de concordancia. Los parámetros de las solicitudes aceptadas como resultado de esta operación *AcceptAnyCall* no se asignarán a una variable.

EJEMPLO:

```
MyPort.getcall;                              // Retira la primera solicitud de MyPort

MyPort.getcall from MyPartner;               // Retira una solicitud de MyPartner
                                             // del puerto MyPort

MyPort.getcall -> sender MySenderVar;        // Retira una solicitud de MyPort y extrae
                                             // la dirección de la parte llamante
```

23.3.2.2 Aceptar una solicitud en cualquier puerto

Para aceptar una solicitud en cualquier puerto se utiliza la palabra clave **any** con la operación **getcall**.

EJEMPLO:

```
any port.getcall(MyProc)
```

23.3.3 La operación Reply (respuesta)

La operación **reply** se utiliza para responder a una solicitud aceptada conforme a la firma del procedimiento. La operación **reply** sólo podrá utilizarse en un puerto por procedimientos (o mixto). En la definición del tipo del puerto se ha de incluir el nombre del procedimiento al que pertenece la operación **reply**.

NOTA – No siempre es posible comprobar estáticamente la relación entre una solicitud aceptada y una operación **reply**. En el contexto de pruebas se autoriza la especificación de una operación **reply** sin una operación **getcall** asociada.

La parte del valor de la operación **reply** consiste en una referencia de firma con una lista de parámetros efectivos asociados y el valor de respuesta (facultativo). La firma se puede definir en forma de plantilla o en línea. Todos los parámetros **out** e **inout** de la firma tendrán un valor específico, lo que significa que no se permite la utilización de mecanismos de concordancia del tipo de *AnyValue*.

Las respuestas a una o varias operaciones **call** podrán ser enviadas a una, varias o todas las entidades par conectadas al puerto designado. Esto puede especificarse de la misma forma que en 23.2.1.1. Esto significa que el argumento de la cláusula **to** de una operación **reply** representa, para las respuestas unidifusión, la dirección de una entidad receptora, para las respuestas multidifusión, una lista de direcciones de un conjunto de receptores y para las respuestas radiodifusión, la palabra clave **all component**.

Para las conexiones uno a uno, puede omitirse la cláusula **to** puesto que la estructura del sistema identifica inequívocamente a la entidad receptora.

Si es necesario devolver un valor a la parte llamante, hay que indicarlo explícitamente mediante la palabra clave **value**.

EJEMPLO:

```
MyPort.reply(MyProc2:{ - ,5});              // Responde a una solicitud aceptada
                                            // de MyProc2.

MyPort.reply(MyProc2:{ - ,5}) to MyPeer;    // Responde a una solicitud aceptada de MyProc2
                                            // recibida de MyPeer

MyPort.reply(MyProc2:{ - ,5}) to (MyPeer1, MyPeer2);       // Respuesta multidifusión a
                                                           // MyPeer1 y MyPeer2

MyPort.reply(MyProc2:{ - ,5}) to all component;    // Respuesta radiodifusión a todas las
                                                   // entidades conectadas a MyPort

MyPort.reply(MyProc3:{5,MyVar} value 20);          // Responde a una solicitud aceptada
                                                   // de MyProc3.
```

23.3.4 La operación Getreply (tratar respuesta)

23.3.4.0 Consideraciones generales

La operación **getreply** se utiliza para tratar respuestas de un procedimiento solicitado anteriormente. Sólo se utilizará en un puerto por procedimientos (o mixtos). La definición de tipo de puerto ha de incluir el nombre del procedimiento al que pertenece la operación **getreply**.

La operación **getreply** retira la primera respuesta de la cola del puerto entrante solamente si se cumplen los criterios de concordancia asociados con la operación **getreply**. Estos criterios de concordancia están asociados con la firma del procedimiento procesado y el interlocutor. Los criterios de concordancia para la firma se pueden especificar en línea y también se pueden obtener de una plantilla de firma.

Es posible especificar la concordancia con un valor de respuesta recibido, utilizando la palabra clave **value**.

Es posible restringir una operación **getreply** a un determinado interlocutor en el caso de conexiones de uno a muchos, utilizando la palabra clave **from**.

EJEMPLO 1:

```
MyPort.getreply(MyProc:{5, ?} value 20);          // Acepta una respuesta de MyProc con
                                                  // dos parámetros out o inout y el
                                                  // valor de respuesta 20

MyPort.getreply(MyProc2:{ - , 5}) from MyPeer;    // Acepta una respuesta de de MyProc2
                                                  // recibida de MyPeer
```

El argumento de firma de la operación **getreply** no se ha de utilizar para transferir nombres de variables para parámetros **out** e **inout**. La asignación de valores de parámetros **out** e **inout** a variables se efectuará en la parte de asignación de esta operación **getreply**. Esto permite utilizar plantillas de firma en operaciones **getreply** de la misma manera que se utilizan plantillas para tipos.

La parte de asignación (facultativa) de la operación **getreply** comprende la asignación de valores de parámetros **out** e **inout** a variables y la extracción de la dirección del emisor de la respuesta. La palabra clave **value** se utiliza para extraer valores de respuesta, y la palabra clave **param** para extraer los valores de parámetros de una respuesta. La palabra clave **sender** se utiliza si es necesario extraer la dirección del emisor.

EJEMPLO 2:

```
MyPort.getreply(MyProc1:{?, ?} value ?) -> value MyRetValue param(MyPar1,MyPar2);
// El valor de respuesta se asigna a la variable MyRetValue y los valores de los dos
// parámetros out o inout se asignan a las variables MyPar1 y MyPar2.

MyPort.getreply(MyProc1:{?, ?} value ?) -> value MyRetValue param( - , MyPar2)
sender MySender;
// No se tiene en cuenta el valor del primer parámetro para la ejecución del resto
// de la prueba; se extrae la dirección del componente emisor y se almacena en
// la variable MySender.

// Los siguientes ejemplos ilustran algunas posibilidades para asignar valores
// de parámetros out e inout a variables. Supóngase que ésta es la firma del
// procedimiento solicitado

signature MyProc2(in integer A, integer B, integer C, out integer D, inout integer E);

MyPort.getreply(ATemplate) -> param( - , - , - , MyVarOut1, MyVarInout1);

MyPort.getreply(ATemplate) -> param(MyVarOut1:=D, MyVarOut2:=E);

MyPort.getreply(MyProc2:{ - , - , - , 3, ?}) - param(MyVarInout1:=E);
```

23.3.4.1 Obtener cualquier respuesta (GetAnyReply)

Una operación **getreply** sin lista de argumentos para los criterios de concordancia de firmas retira el mensaje respuesta en cabeza de la cola de puerto entrante (en su caso) siempre que se cumplan los demás criterios de concordancia. Los parámetros o los valores de respuesta aceptados como resultado de la operación *GetAnyReply* no se asignarán a una variable. Si se utiliza *GetAnyReply* en la parte de tratamiento de respuestas y excepciones de una operación **call**, ha de tratar únicamente respuestas del procedimiento invocado por la operación **call**.

EJEMPLO:

```
MyPort.getreply;                        // Retira la primera respuesta de MyPort

MyPort.getreply from MyPeer;            // Retira de MyPort la primera respuesta recibida
                                        // de MyPeer

MyPort.getreply -> sender MySenderVar; // Retira la primera respuesta de MyPort y extrae
                                        // la dirección de la entidad emisora
```

23.3.4.2 Obtener una respuesta en cualquier puerto

Para obtener una respuesta en cualquier puerto se utilizarán las palabras clave **any port**.

EJEMPLO:

```
any port.getreply(Myproc)
```

23.3.5 La operación Raise (plantear excepción)

La operación **raise** se utiliza para plantear una excepción. Sólo se plantearán excepciones en puertos por procedimientos (o mixtos). Una excepción es una reacción a una solicitud de procedimiento aceptada que produce un resultado excepcional. Hay que especificar el tipo de excepción en la firma del procedimiento solicitado. En la definición del tipo del puerto, en la lista de solicitudes de procedimientos aceptadas, deberá aparecer el nombre del procedimiento al que pertenece la excepción.

NOTA – No siempre es posible comprobar estáticamente la relación entre una solicitud aceptada y una operación **raise**. En el contexto de pruebas se autoriza la especificación de una operación **raise** sin una operación **getcall** asociada.

La parte del valor de la operación **raise** consiste en una referencia de firma seguida por el valor de la excepción.

Las excepciones se especifican como tipos. Por tanto, el valor de excepción se puede deducir de una plantilla, pero también puede ser el valor resultante de una expresión (naturalmente, puede ser un valor explícito). Se utilizará el campo de tipo facultativo en la especificación de valor de la operación **raise** cuando sea necesario para evitar toda ambigüedad en cuanto al tipo de valor que se envía.

Las excepciones a una o varias operaciones **call** pueden ser enviadas a una, varias o todas las entidades par conectadas al puerto designado. Esto puede especificarse conforme a 23.2.1.1. Significa que el argumento de la cláusula **to** de una operación **raise** representa, para las excepciones unidifusión, la dirección de una entidad receptora, para las excepciones multidifusión, una lista de direcciones de un conjunto de receptores y para las excepciones de tipo radiodifusión las palabras clave **all component**.

Para las conexiones uno a uno, puede omitirse la cláusula **to** puesto que la estructura del sistema identifica inequívocamente a la entidad receptora.

EJEMPLO:

```
  MyPort.raise(MySignature, MyVariable + YourVariable – 2);
// Plantea una excepción con un valor que es el resultado de la expresión aritmética en MyPort

  MyPort.raise(MyProc, integer:5});
  // Plantea una excepción con el valor entero 5 para MyProc

  MyPort.raise(MySignature, "My string") to MyPartner;
  // Plantea una excepción con el valor "My string" en MyPort para Mysignature y la envía
  // a MyPartner

  MyPort.raise(MySignature, "My string") to (MyPartnerOne, MyPartnerTwo);
  // Plantea una excepción con el valor "My string" en MyPort y la envía a MyPartnerOne
  // y MyPartnerTwo (es decir, comunicación multidifusión)

  MyPort.raise(MySignature, "My string") to all component;
  // Plantea una excepción con el valor "My string" en MyPort para MySignature y la envía
  // a todas las entidades conectadas a MyPort (es decir, comunicación multidifusión)
```

23.3.6 La operación Catch (reconocer excepción)

23.3.6.0 Consideraciones generales

La operación **catch** se utiliza para reconocer excepciones planteadas por un componente de prueba o el SUT como reacción a una solicitud de procedimiento. La operación **catch** sólo se utilizará en puertos por procedimientos (o mixtos). El tipo de la excepción reconocida se especificará en la firma del procedimiento que ha planteado la excepción. Dado que las excepciones se especifican como tipos, pueden tratarse como mensajes. Por ejemplo, es posible utilizar plantillas para diferenciar distintos valores del mismo tipo de excepción.

La operación **catch** retira la primera excepción de la lista del puerto entrante asociada si satisface todos los criterios de concordancia asociados a la operación **catch**. Los valores entrantes no se podrán vincular a los términos de la expresión ni a la plantilla. La asignación de los valores de excepción a las variables ha de llevarse a cabo en la parte de asignación de la operación **catch**.

Es posible restringir una operación **catch** a un determinado interlocutor en el caso de conexiones de uno a muchos, utilizando la palabra clave **from**.

EJEMPLO 1:

```
MyPort.catch(MyProc, integer: MyVar);  // Reconoce una excepción entero del valor MyVar
                                       // planteada por MyProc en el puerto MyPort

MyPort.catch(MyProc, MyVar);                 // Es una alternativa al ejemplo anterior

MyPort.catch(MyProc, A<B);                   // Reconoce una excepción booleana

MyPort.catch(MyProc, MyType:{5, MyVar});     // Definición de plantilla de un valor de
                                             // excepción en línea

MyPort.catch(MyProc, charstring:"Hello")from MyPeer;     // Reconoce la excepción "Hello"
                                                         // de MyPeer
```

La parte de asignación (facultativa) de la operación **catch** consiste en la asignación del valor de excepción y la extracción de la dirección del componente llamante. Se utiliza la palabra clave **value** para extraer el valor de una excepción, y la palabra clave **sender** si es necesario extraer la dirección del emisor.

EJEMPLO 2:

```
MyPort.catch(MyProc, MyType:?) from MyPartner -> value MyVar;
// Reconoce una excepción de MyPartner y asigna su valor a MyVar.

MyPort.catch(MyProc, MyTemplate(5)) -> value MyVarTwo sender MyPeer;
// Reconoce una excepción, asigna su valor a MyVarTwo y extrae la dirección del emisor.
```

La operación **catch** puede incluirse en la parte de tratamiento de respuestas y excepciones de una operación **call** y también se puede utilizar para determinar una alternativa en una instrucción **alt**. Si la operación **catch** se utiliza en la parte de aceptación de una operación **call**, es redundante la información relativa al nombre de puerto y la referencia de firma para indicar el procedimiento que planteó la excepción, porque esta información se deduce de la operación **call**. Sin embargo, esta información se repetirá para que el proceso sea más legible (por ejemplo, en el caso de instrucciones **call** complejas).

23.3.6.1 La excepción Timeout (expiración de temporizador)

Se trata de una excepción **timeout** especial asociada a la expiración de un temporizador y que puede ser reconocida mediante la operación **catch**. La excepción **timeout** es una salida de emergencia cuando un procedimiento solicitado no responde ni plantea una excepción dentro de un plazo predeterminado (véase 23.3.1.2).

EJEMPLO:

```
MyPort.call(MyProc:{5,MyVar}, 20E-3) {
  [] MyPort.getreply(MyProc:{?, ?}) { }
  [] MyPort.catch(timeout) {           // excepción timeout después de 20 ms
        setverdict(fail);
        stop;
     }
}
```

El reconocimiento de excepciones **timeout** se restringirá a la parte de tratamiento de excepciones de una solicitud. En la operación **catch** que trata una excepción **timeout** no puede haber otros criterios de concordancia (incluida una parte **from**) ni una parte de asignación.

23.3.6.2 Reconocer cualquier excepción

Una operación **catch** sin lista de argumentos permite reconocer cualquier excepción válida. En el caso más general no se utiliza la palabra clave **from**. Los valores de excepción aceptados por *CatchAnyException* no han de ser asignados a una variable. Si se utiliza *CatchAnyException* en la parte de tratamiento de respuestas y excepciones de una operación **call**, ésta ha de tratar únicamente las excepciones planteadas por el procedimiento invocado por la operación **call**. *CatchAnyException* también reconocerá la excepción **timeout**.

EJEMPLO:

```
MyPort.catch;

MyPort.catch from MyPartner;

MyPort.catch -> sender MySenderVar;
```

23.3.6.3 Reconocer una excepción en cualquier puerto

Para reconocer (**catch**) una excepción en cualquier puerto se utiliza la palabra clave **any**.

EJEMPLO:

```
any port.catch;
```

23.4 La operación Check (comprobar)

23.4.0 Consideraciones generales

Se trata de una operación genérica que permite acceder en lectura al primer elemento de las colas de puertos *entrantes* por mensajes y por procedimientos, sin retirar dicho elemento de la cola. La operación **check** tiene que tratar valores de un cierto tipo en los puertos por mensajes y distinguir entre solicitudes pendientes de aceptación, excepciones que hay que reconocer y respuestas originadas por anteriores solicitudes en los puertos por procedimientos.

La operación **check** utiliza las operaciones de recepción **receive**, **getcall**, **getreply** y **catch,** así como sus partes de concordancia y de asignación, para definir la condición que se ha de comprobar y extraer el valor o valores de sus parámetros, si es necesario.

Se comprobará el elemento de *cabeza* 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 toma una copia del elemento de cabeza y se ejecuta en esta copia la operación de recepción especificada en la operación **check**. La operación **check** fracasa si la operación de recepción termina con resultado fallido, es decir, si no se cumplen los criterios de concordancia. En este caso, se descarta la *copia* del elemento de cabeza de la cola y la ejecución de la prueba continúa normalmente, es decir, se evalúa la instrucción o la alternativa que siguen a esta operación de comprobación. La operación **check** es fructuosa si la operación de recepción termina con resultado satisfactorio.

Una utilización errónea de la operación **check**, por ejemplo, para comprobar una excepción en un puerto por mensajes 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 o durante la compilación.

EJEMPLO:

```
MyPort1.check(receive(5)); // Comprobar si hay mensaje de entero de valor 5.

MyPort2.check(getcall(MyProc:{5, MyVar}) from MyPartner);
// Comprobar si hay solicitud de MyProc en el puerto MyPort2 de MyPartner

MyPort2.check(getreply(MyProc:{5, MyVar} value 20));
// Comprobar si hay respuesta del procedimiento MyProc en MyPort2 con valor de respuesta 20
// y los valores de los dos parámetros out e inout son 5 y el valor de MyVar.

MyPort2.check(catch(MyProc, MyTemplate(5, MyVar)));

MyPort2.check(getreply(MyProc1:{?, MyVar} value *) -> value MyReturnValue param(MyPar1));

MyPort.check(getcall(MyProc:{5, MyVar}) from MyPartner -> param (MyPar1Var, MyPar2Var));

MyPort.check(getcall(MyProc:{5, MyVar}) -> sender MySenderVar);
```

23.4.1 Comprobar cualquier operación (Check any)

Una operación **check** sin lista de argumentos permite comprobar si hay algo en espera de procesamiento en una cola de puerto entrante. La operación **check** any 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 simplificada con una cláusula **sender**. Para un puerto mixto, la operación **check** any comprueba las colas de entrada por mensaje y por procedimiento. Si ambas colas de entrada de esta operación concuerdan, se otorgará prioridad a la información relacionada con la cola por procedimiento, es decir, se devuelve como resultado de la operación **check** any. Por ejemplo, si las colas de entrada por mensaje y por procedimiento de un puerto mixto no están vacías y la información del emisor debería recuperarse mediante la operación **check** any, debería devolverse el emisor de la solicitud, respuesta o excepción en la cola de entrada por procedimiento.

NOTA – La información relacionada con la cola de entrada por mensaje de un puerto mixto puede recuperarse fácilmente gracias a la operación **check** combinada con una operación **receive** any, por ejemplo,

`MyPort.check(receive) -> sender Mysender.`

EJEMPLO:

```
MyPort.check;

MyPort.check(from MyPartner);

MyPort.check(-> sender MySenderVar);
```

23.4.2 Comprobar en cualquier puerto

Las palabras clave **any port** permiten comprobar en cualquier puerto. En el caso de una operación **check** en cualquier puerto sin argumento, las colas de entrada de los puertos mixtos han de comprobarse como se especifica en 23.4.1.

EJEMPLO:

```
any port.check;
```

23.5 Control de puertos de comunicación

23.5.0 Consideraciones generales

Operaciones de TTCN-3 para controlar puertos por mensajes, puertos por procedimientos y puertos mixtos:

- **clear**: retirar el contenido de la cola del puerto entrante;
- **start**: retirar el contenido de la cola del puerto entrante y habilitar las operaciones de transmisión y recepción en el puerto;
- **stop**: inhabilitar las operaciones de transmisión y prohibir las de recepción que coinciden en el puerto;
- **halt**: inhabilitar las operaciones de transmisión en el puerto inmediatamente y prohibir las de recepción para que coincidan con los nuevos mensajes/solicitudes/respuestas/excepciones que acceden a la cola del puerto tras la ejecución de la operación halt. Las entradas que ya están en la cola pueden ser procesadas.

23.5.1 La operación Clear (liberar puerto)

La operación **clear** limpia el contenido de la cola *entrante* del puerto denominado. Si la cola del puerto ya está vacía, esta operación no tendrá ninguna acción.

EJEMPLO:

```
MyPort.clear;  // libera el puerto MyPort
```

23.5.2 La operación Start (activar puerto)

Si la definición de un puerto incluye operaciones de recepción como **receive**, **getcall** etc., la operación **start** limpia la cola entrante del puerto denominado y activa la espera de tráfico por el puerto. Si el puerto se ha definido para operaciones de emisión, también se podrán ejecutar en este puerto operaciones como **send**, **call**, **raise**, etc.

EJEMPLO:

```
MyPort.start;  // activar MyPort
```

La opción por defecto será activar implícitamente todos los puertos de un componente al crearlo. El efecto de la operación activar puerto será reactivar puertos que no han sido desactivados, retirando todos los mensajes en espera en la cola entrante.

23.5.3 La operación Stop (desactivar puerto)

Si la definición de un puerto incluye operaciones de recepción como **receive** y **getcall**, la operación **stop** pone fin a la espera de tráfico en el puerto denominado. Si el puerto se ha definido para operaciones de emisión, la operación **stop** invalida la ejecución de operaciones tales como **send**, **call**, **raise**, etc.

EJEMPLO 1:

```
MyPort.stop;   // desactivar MyPort
```

NOTA – Al cesar el estado de espera de tráfico en el puerto se han de realizar completamente todas las operaciones de recepción definidas antes de la operación stop, antes de suspender el funcionamiento del puerto.

EJEMPLO 2:

```
MyPort.receive (MyTemplate1) -> value RecPDU;
                                        // decodificación del valor recibido, comparación
                                        // con MyTemplate1 y almacenamiento del valor
                                        // concordante en la variable RecPDU

MyPort.stop;                            // No se ejecuta ninguna operación de recepción
                                        // definida después de ejecutar la operación stop
                                        // (excepto si el puerto se reactiva con una
                                        // operación start ulterior)

MyPort.receive (MyTemplate2);           // Esta operación no concuerda y se bloqueará
                                        // (suponiendo que no está activada ninguna
                                        // opción por defecto)
```

23.5.4 Operación parar puerto (halt port)

Si un puerto acepta operaciones de recepción como **receive**, **trigger** y **getcall**, a partir de la ejecución de la operación **halt** en ese puerto, se prohíbe que las operaciones de recepción traten los elementos de solicitud que acceden a la cola del puerto. Los elementos de solicitud de mensajes y procedimientos que ya estaban en la cola antes de ejecutar la operación **halt** pueden ser tratados con las operaciones de recepción. Si el puerto acepta operaciones de

transmisión, la operación **halt** port prohíbe inmediatamente que se ejecuten operaciones de transmisión tales como **send**, **call**, **raise**, etc. Las operaciones halt subsiguientes no tendrán efecto alguno sobre el estado del puerto o de su cola.

NOTA 1 – Cuando se ejecuta la operación **halt** port se coloca virtualmente un marcador tras la recepción de la última entrada en la cola. Las entradas previas al marcador pueden tratarse normalmente. Una vez procesadas todas las entradas previas al marcador, el estado del puerto es equivalente al estado stopped (detenido).

NOTA 2 – Si se ejecuta una operación **stop** port en un puerto parado antes de que hayan sido procesadas todas las entradas previas al marcador, se prohíben inmediatamente las operaciones de recepción ulteriores (es decir, el marcador pasa virtualmente a la cabeza de la cola).

NOTA 3 – Si se ejecuta una operación **start** port en un puerto parado, se liberan todas las entradas en la cola, independientemente de que hayan llegado antes o después de ejecutar la operación **halt** port. Además se suprime el marcador.

NOTA 4 – Si se ejecuta una operación **clear** en un puerto parado, se liberan todas las entradas en la cola, independientemente de que hayan llegado antes o después de ejecutar la operación **halt** port. Además se coloca virtualmente el marcador en la cabeza de la cola.

EJEMPLO:

```
MyPort.halt;                        // A partir de este momento no se permiten transmisiones
                                    // en Myport; el tratamiento de mensajes en la cola aún
                                    // es posible

MyPort.receive (MyTemplate1);       // Si había un mensaje en la cola antes de la operación
                                    // halt y concuerda con MyTemplate1, será tratado;
                                    // en los otros casos la operación de recepción se bloquea.
```

23.6 Utilización de las palabras clave any y all con puertos

Las palabras claves **any** y **all** se pueden utilizar con operaciones de configuración y comunicación conforme al cuadro 18.

Cuadro 18/Z.140 – Any y all con puertos

Operación	Permitido		Ejemplo
	any	**all**	
receive, trigger, getcall, getreply, catch, check)	sí		any port.receive
connect/map			
start, stop, clear, halt		sí	all port.start

24 Operaciones de temporización

24.0 Consideraciones generales

La notación TTCN-3 soporta varias operaciones de temporización que se pueden utilizar en casos de prueba, funciones, altsteps y el control de módulo.

Se supone que todas las unidades de ámbito de TTCN-3 en las que se han declarado temporizadores mantienen su propia *lista de los temporizadores en curso* y de *temporizadores que han expirado*. Las listas de temporizadores forman parte de los estados puntuales registrados cuando se ejecuta un caso de prueba. Las listas se actualizan cuando se activa, se desactiva o expira un temporizador en la unidad de ámbito, y cuando se ejecuta la operación **timeout**.

NOTA 1 – Las listas de temporizadores en curso y temporizadores que han expirado sólo son conceptuales y no restringen la implementación de temporizadores. Se pueden utilizar igualmente otras estructuras de datos, por ejemplo conjuntos sin ninguna restricción de acceso a eventos de temporización, sea por el orden en que se registran los eventos de expiración de temporización o de otra clase.

NOTA 2 – Se supone que para cada componente de prueba hay listas especiales de temporizadores en curso y temporizadores que han expirado, que realizan el tratamiento de los eventos de activación/desactivación y expiración de los temporizadores declarados en la definición correspondiente de tipo de componente.

Al expirar un temporizador (conceptualmente ocurre inmediatamente antes del tratamiento instantáneo de una serie de alternativas de eventos), se registra un evento de expiración en la lista correspondiente propia de la unidad de ámbito en la que se ha declarado el temporizador. Este temporizador queda inmediatamente inactivo. En un momento dado sólo puede haber un registro para cada temporizador en la lista de la unidad de ámbito para los temporizadores que han expirado.

Todos los temporizadores en curso serán cancelados automáticamente al desactivar el componente de forma explícita o implícita.

Cuadro 19/Z.140 – Operaciones de temporización de TTCN-3

Operaciones de temporización	
Instrucción	**Palabra clave o símbolo asociados**
Activar temporizador	**start**
Desactivar temporizador	**stop**
Leer tiempo transcurrido	**read**
Comprobar si el temporizador está en curso	**running**
Evento expiración de temporización	**timeout**

24.1 La operación Start (activar temporizador)

La operación **start** se utiliza para provocar la activación un temporizador. Se utilizarán valores de tipo **float** no negativos (es decir, 0,0 o superiores). Al activar un temporizador se registra su nombre en la lista de temporizadores en curso (de la unidad de ámbito).

EJEMPLO:

```
MyTimer1.start;            // MyTimer1 se activa con duración por defecto
MyTimer2.start(20E-3);     // MyTimer2 se activa con duración de 20 ms

// También se pueden activar en un bucle los elementos de matrices de temporización,
// por ejemplo el temporizador t_Mytimer [5];
var float v_timerValues [5];

para (var integer i := 0; i<=4; i:=i+1)
  { v_timerValues [i] := 1.0 }

para (var integer i := 0; i<=4; i:=i+1)
  {t_Mytimer [i].start ( v_timerValues [i])}
```

Se utilizará el parámetro de valor de temporizador facultativo si no se ha indicado ninguna duración por defecto, o para 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 actual del temporizador. Las ulteriores operaciones **start** para este temporizador que no especifiquen una duración utilizarán la duración por defecto.

Un temporizador activado con el valor de temporización 0,0 expira inmediatamente. Se producirá un error de funcionamiento si se activa un temporizador con un valor de temporización negativo, por ejemplo un valor que resulta de una expresión o sin especificación de valor.

El reloj del temporizador funciona a partir del valor de coma flotante cero (0,0) hasta un máximo indicado por el parámetro de duración.

Es posible aplicar la operación **start** a un temporizador en curso: será desactivado y reactivado nuevamente. Hay que retirar el registro de este temporizador de la lista de temporizadores que han expirado (en su caso).

24.2 La operación Stop (desactivar temporizador)

La operación **stop** se utiliza para detener un temporizador en curso y suprimirlo de la lista de correspondiente. El temporizador pasa al estado inactivo y se establece un valor decimal de tiempo transcurrido cero (0,0).

La desactivación de un temporizador inactivo es una operación válida, pero no produce ningún efecto. La desactivación de un temporizador que ha expirado provoca su inclusión en la lista de temporizadores que han de suprimirse. Se puede utilizar la palabra clave **all** para detener todos los temporizadores visibles en la unidad de ámbito en la que se ha solicitado la operación **stop**.

EJEMPLO:

```
MyTimer1.stop;        // detiene MyTimer1
all timer.stop;       // detiene todos los temporizadores en curso
```

24.3 La operación Read (leer temporizador)

La operación **read** se utiliza para extraer el tiempo que ha transcurrido desde la activación del temporizador especificado El valor devuelto será de tipo **float**.

EJEMPLO:

```
var float Myvar;
MyVar := MyTimer1.read;     // asigna a MyVar el tiempo transcurrido desde
                            // que se activó MyTimer1
```

La aplicación de la operación **read** a un temporizador inactivo, es decir, que no figura en la lista de temporizadores en curso devolverá el valor float cero (0,0).

24.4 La operación Running (temporizador en curso)

La operación **running** se utiliza para comprobar si un temporizador figura en la lista de temporizadores en curso de una determinada unidad de ámbito (es decir, que se ha activado y que no ha expirado ni se ha desactivado). La operación devuelve el valor **true** si el temporizador figura en la lista, y **false** si no figura.

EJEMPLO:

```
if (MyTimer1.running) { … }
```

24.5 La operación Timeout (expiración de temporizador)

La operación **timeout** permite comprobar si ha expirado un temporizador, o todos los temporizadores, en una unidad de ámbito de un componente de prueba o control de módulo en que se ha solicitado esta operación timeout.

Cuando se ejecuta una operación **timeout** con indicación del nombre de un temporizador se examinan las listas de temporizadores que han expirado, conforme a las reglas de ámbito de la notación TTCN-3. Si un evento de expiración concuerda con el nombre del temporizador, se retira de la lista y la operación **timeout** termina satisfactoriamente. No se utilizará la operación **timeout** en una expresión **booleana**, pero sí puede utilizarse para determinar una alternativa en una instrucción **alt** o como instrucción autónoma en una descripción de comportamiento. En el último caso se considera que la operación **timeout** es una expresión simplificada de una instrucción **alt** con una sola alternativa, esto es, tiene semántica bloqueante y por ello permite esperar pasivamente la expiración de temporizadores.

EJEMPLO 1:

```
MyTimer1.timeout;     // comprueba si ha expirado el temporizador MyTimer1 activado antes
```

En la operación **timeout** se puede utilizar la palabra clave **any** (en vez de un temporizador denominado explícitamente): el resultado será satisfactorio si la lista de temporizadores que han expirado no está vacía.

EJEMPLO 2:

```
any timer.timeout;    // comprueba si ha expirado uno de los temporizadores activados antes
```

24.6 Utilización de las palabras clave any y all con temporizadores

Las palabras clave **any** y **all** se pueden utilizar con operaciones de temporizador como se indica en el cuadro 20.

Cuadro 20/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**

25 Operaciones de veredicto de prueba

25.0 Consideraciones generales

Las operaciones de veredicto permiten fijar y extraer veredictos utilizando, respectivamente, las operaciones **setverdict** y **getverdict**. Estas operaciones sólo se utilizarán en casos de prueba, altsteps y funciones.

Cuadro 21/Z.140 – Las operaciones de veredicto de prueba de TTCN-3

Operaciones de veredicto de prueba	
Instrucción	**Palabra clave o símbolo asociado**
Fijar veredicto local	**setverdict**
Obtener veredicto local	**getverdict**

Cada componente de prueba de la configuración activa mantendrá su propio veredicto local. Es un objeto creado para cada componente de prueba al definir esa ejemplificación, que se utiliza para hacer un seguimiento del veredicto individual en cada componente de prueba (es decir, en el MTC y en cada uno de los PTC).

25.1 Veredicto de caso de prueba

Hay además un veredicto de caso de prueba global ejemplificado y tratado por el sistema de prueba que se actualiza cuando cada componente de prueba (es decir el MTC y cada uno de los PTC) termina la ejecución. Este veredicto no se puede tratar utilizando las operaciones **getverdict** y **setverdict**. El caso de prueba devuelve el valor de este veredicto al terminar la ejecución. El veredicto devuelto se pierde si no se guarda explícitamente en la parte de control (por ejemplo, asignado a una variable).

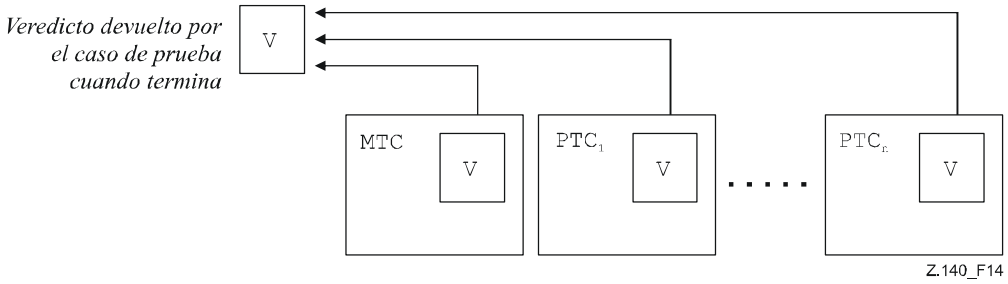


Figura 14/Z.140 – Ilustración de la relación entre veredictos

NOTA – La notación TTCN-3 no especifica los mecanismos efectivos que actualizan los veredictos local y de casos de prueba, porque estos mecanismos dependen de la implementación.

25.2 Valores de veredicto y reglas de reemplazo

25.2.0 Consideraciones generales

El veredicto puede tener cinco valores diferentes: **pass** (satisfactorio), **fail** (fallido), **inconc** (no concluyente), **none** (ninguno) y **error**. Son los valores diferenciados de **verdicttype** (véase 6.1).

NOTA – **inconc** significa un veredicto no concluyente.

La operación **setverdict** sólo se utilizará con los valores **pass**, **fail**, **inconc** y **none**.

EJEMPLO 1:

```
setverdict(pass);
setverdict(inconc);
```

El valor del veredicto local se puede extraer mediante la operación **getverdict**.

EJEMPLO 2:

```
MyResult := getverdict;          // Donde MyResult es una variable de tipo verdicttype
```

Cuando un componente de prueba se ejemplifica, se crea el objeto de su veredicto local y se le atribuye el valor **none**.

El efecto de una modificación del valor del veredicto local (utilización de la operación **setverdict**) ha de ajustarse a las reglas de reemplazo del cuadro 22. El veredicto del caso de prueba se actualiza implícitamente al terminar un componente de prueba. El efecto de esta operación implícita también ha de ajustarse a las reglas de reemplazo del cuadro 22.

Cuadro 22/Z.140 – Reglas de reemplazo del veredicto

Valor actual de veredicto	Nuevo valor de asignación de veredicto			
	pass	**inconc**	**fail**	**none**
none	**pass**	**inconc**	**fail**	**none**
pass	**pass**	**inconc**	**fail**	**pass**
inconc	**inconc**	**inconc**	**fail**	**inconc**
fail	**fail**	**fail**	**fail**	**fail**

EJEMPLO 3:

```
 :
setverdict(pass);          // el veredicto local se pone a satisfactorio
 :
setverdict(fail);          // hasta que se ejecuta esta línea, que provoca el reemplazo del
 :                         // valor del veredicto local por fallido. Cuando el ptc termina,
 :                         // el veredicto de caso de prueba se pone a fallido
```

25.2.1 Veredicto de error

El veredicto **error** es especial porque lo establece el sistema de prueba para indicar que se ha producido un error de caso de prueba (es decir, durante la ejecución). No lo establecerá la operación **setverdict** ni lo devolverá la operación **getverdict**. 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**.

26 Acciones externas

Es posible que en determinadas situaciones de prueba no existan o se desconozcan *a priori* algunas interfaces con el SUT (por ejemplo, la interfaz de gestión), en casos en los que es necesario estimular el SUT para realizar determinadas acciones (por ejemplo, enviar un mensaje al sistema de prueba). En otros casos el personal de ejecución tiene que realizar determinadas acciones (por ejemplo, modificar las condiciones de entorno de las pruebas: temperatura, tensión de alimentación, etc.).

Estas acciones se pueden describir como una expresión de cadena: se permite la utilización de cadenas literales, variables y parámetros de tipo cadena, etc., y cualquier concatenación de los mismos.

EJEMPLO:

```
var charstring myString:= " now."
action("Send MyTemplate on lower PCO" & myString);  // Descripción informal de la acción
                                                    // externa
```

No se especifica qué hay que hacer en el SUT o qué debe hacer el SUT para activar esta acción; sólo es una descripción informal de la acción necesaria.

Las acciones externas se pueden utilizar en casos de prueba, funciones, altsteps y control de módulo.

27 Parte de control de módulo

27.0 Consideraciones generales

La parte de definiciones del módulo define los casos de prueba, y la parte de control ejecuta estos casos. Todas las variables (en su caso) definidas en la parte de control de un módulo se transferirán al caso de prueba por parametrización si se van a utilizar en la definición de comportamiento de ese caso de prueba, es decir, la notación TTCN-3 no soporta ninguna clase de variables.

La configuración de prueba se restablece al activar cada caso de prueba. Todos los componentes y los puertos controlados por las operaciones **create**, **connect**, etc., en un caso de prueba anterior se suprimen al desactivar ese caso (no son 'visibles' para el nuevo caso de prueba).

27.1 Ejecución de casos de prueba

Para invocar un caso de prueba se utiliza una instrucción **execute**. Como resultado de la ejecución de un caso de prueba se devolverá un veredicto **none**, **pass**, **inconc**, **fail** o **error** que se puede asignar a una variable para ulterior procesamiento.

La instrucción **execute** permite la supervisión de un caso de prueba por medio de una temporización (es facultativo) (véase 27.5).

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'
```

27.2 Terminación de casos de prueba

Un caso de prueba finaliza cuando termina el MTC. Al terminar el MTC (de forma explícita o implícita) el sistema de prueba suprimirá todos los componentes de prueba paralelos en funcionamiento.

NOTA 1 – El mecanismo concreto para detener todos los PTC es específico de la herramienta y, por tanto, 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 distintos componentes de prueba, de acuerdo con las reglas definidas en la cláusula 25. El veredicto local efectivo de un componente de prueba será su veredicto local final cuando el componente de prueba termine por sí mismo, disponga su propia desactivación o sea desactivado por otro componente de prueba o por el sistema de prueba.

NOTA 2 – Para evitar que un retardo en la desactivación de los PTC provoque una situación de competencia en el cálculo de veredictos de prueba, es conveniente que el MTC compruebe si todos los PTC se han desactivado (por medio de la instrucción **done** o **killed**) antes de desactivarse a sí mismo.

27.3 Control de la ejecución de casos de prueba

Las instrucciones de programa definidas en los cuadros 11 y 12 se pueden utilizar en la parte de control de un módulo para especificar, por ejemplo, el orden de ejecución de los casos pruebas o el número de veces que se debería ejecutar un caso de prueba.

EJEMPLO:

```
module MyTestSuite () {
     :
    control {
            :
           // Hacer esta prueba 10 veces
           count:=0;
           while (count < 10)
           {     execute (MySimpleTestCase1());
           count := count+1;
           }
    }
}
```

Si no se utilizan instrucciones de programación, la opción por defecto es ejecutar los casos de prueba en el orden secuencial en el que aparecen en el control de módulo.

NOTA – Esto no excluye la posibilidad de revocar este orden por defecto en una herramienta. El usuario o la herramienta pueden seleccionar un orden de ejecución diferente.

Para controlar la ejecución de casos de prueba también pueden emplearse la selección de casos de prueba y su correspondiente anulación (véase 27.4).

27.4 Selección de casos de prueba

En la notación TTCN-3 hay diferentes formas de seleccionar casos de prueba y de anular su selección. Por ejemplo, se pueden utilizar expresiones booleanas para seleccionar los casos de prueba a ejecutar y deseleccionarlos. Esto incluye naturalmente la utilización de funciones que devuelven un valor **boolean**.

EJEMPLO 1:

```
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 reunirlos en una función y ejecutar esa función a partir del control de módulo.

EJEMPLO 2:

```
function MyTestCaseGroup1()
{ execute(MySimpleTestCase1());
  execute(MySimpleTestCase2());
  execute(MySimpleTestCase3());
}
function MyTestCaseGroup2()
{ execute(MySimpleTestCase4());
  execute(MySimpleTestCase5());
  execute(MySimpleTestCase6());
}
 :
control
{ if (MySelectionExpression1()) { MyTestCaseGroup1(); }
  if (MySelectionExpression2()) { MyTestCaseGroup2(); }
     :
}
```

Debido a que un caso de prueba devuelve un valor simple de tipo **verdicttype**, también es posible controlar el orden de ejecución de un caso de prueba en función del resultado del mismo. La utilización de **verdicttype** de TTCN-3 representa otra manera de seleccionar casos de prueba.

EJEMPLO 3:

```
if ( execute (MySimpleTestCase()) == pass )
{ execute (MyGoOnTestCase()) }
else
{ execute (MyErrorRecoveryTestCase()) };
```

27.5 Utilización de temporizadores en el control

Es posible utilizar un temporizador para controlar la ejecución de casos de prueba. Se puede hacer utilizando una temporización explícita en la instrucción **execute**. Si el caso de prueba no ha terminado al finalizar este plazo, el resultado de ejecución del caso será un veredicto de error y el sistema de prueba terminará el caso. Para la supervisión de los casos de prueba se utiliza un temporizador de sistema que no es necesario declarar o activar.

EJEMPLO 1:

```
MyReturnVal := execute (MyTestCase(), 7E-3);
// Devolverá un veredicto de error si MyTestCase no termina la ejecución dentro de 7 ms
```

También se pueden utilizar explícitamente operaciones de temporizador para controlar la ejecución de un caso de prueba.

EJEMPLO 2:

```
// Ejemplo de utilización de la operación running (temporizador en curso)
while (T1.running or x<10) // Siendo T1 un temporizador activado antes
{   execute(MyTestCase());
    x := x+1;
}

// Ejemplo de utilización de las operaciones start y timeout

timer T1 := 1,0;
 :
execute(MyTestCase1());
T1.start;
T1.timeout;     // Pausa antes de ejecutar el siguiente caso de prueba
execute(MyTestCase2());
```

28 Especificación de atributos

28.0 Consideraciones generales

Es posible asociar atributos a elementos de lenguaje de TTCN-3 por medio de la instrucción **with**. La sintaxis para el argumento de la instrucción **with** (los atributos efectivos) es una cadena de texto libre.

Hay cuatro clases de atributos:

a) **display**: permite la especificación de atributos de visualización relacionados con formatos de presentación específicos;

b) **encode**: permite referencias a determinadas reglas de codificación;

c) **variant**: permite referencias a determinadas variantes de codificación;

d) **extension**: permite la especificación de atributos definidos por el usuario.

28.1 Atributos de visualización (Display)

Todos los elementos de lenguaje de TTCN-3 pueden tener atributos **display** para especificar una determinada forma de presentación, por ejemplo en un formato tabular.

En la Rec. UIT-T Z.141 [1] se describen cadenas especiales de atributos de visualización para el formato de presentación tabular (conformidad).

En la Rec. UIT-T Z.142 [2] se describen cadenas especiales de atributos de visualización para el formato de presentación gráfica.

El usuario puede definir otros atributos **display**.

NOTA – Como los atributos definidos por el usuario no están normalizados, puede haber diferencias de interpretación de estos atributos entre las herramientas, y es posible que algunas herramientas no los soporten.

28.2 Codificación de valores

28.2.0 Consideraciones generales

Las reglas de codificación definen cómo se codifica y cómo se transmite por un **puerto** de comunicación un determinado valor, plantilla, etc., y cómo se han de codificar las señales recibidas. La notación TTCN-3 no tiene un mecanismo de codificación por defecto; las reglas de codificación o directrices de codificación se definen de alguna manera externa a TTCN-3.

En TTCN-3 se pueden especificar reglas de codificación generales o particulares utilizando los atributos **encode** y **variant**.

28.2.1 Atributos de codificación

El atributo **encode** permite asociar a las definiciones de TTCN-3 una determinada regla o directiva de codificación referenciada.

La manera en que se definen las reglas de codificación propiamente dichas (por ejemplo la prosa, las funciones, etc.) no entra en el ámbito de esta Recomendación. Si no hay referencias a ninguna regla, cada implementación determinará la codificación.

Los atributos de codificación se utilizarán de manera jerárquica en la mayoría de los casos. El nivel más alto es todo el módulo, el siguiente nivel es un grupo y el nivel más bajo es un determinado tipo o definición:

a) **`module`**: la codificación se aplica a todos los tipos definidos en el módulo, incluidos los tipos TTCN-3 (tipos integrados);

b) **`group`**: la codificación se aplica a un grupo de definiciones de tipos definidos por el usuario;

c) **`type o definition`**: la codificación se aplica a un solo tipo o una sola definición definidos por el usuario;

d) **`field`**: la codificación se aplica a un campo de un tipo **`record`** o **`set`** o de una **`plantilla`**;

EJEMPLO:

```
module MyTTCNmodule
{   :
    import from MySecondModule {
          type MyRecord
    }
    with { encode "MyRule 1" }    // Los ejemplares de MyRecord se codifican según
                                  // la regla MyRule 1

    :
    type charstring MyType;       // Se codifica normalmente aplicando la 'regla de
                                  // codificación global'
    :
    group MyRecords
    {     :
          type record MyPDU1
          {
                integer     field1,     // field1 se codifica según la regla 'Rule 3
                boolean     field2,     // field2 se codifica según la regla 'Rule 3'
                Mytype      field3      // field3 se codifica según la regla 'Rule 2'
          }
          with { encode (field1, field2) "Rule 3" }
          :
    }
    with { encode "Rule 2" }

}
with { encode "Global encoding rule" }
```

28.2.2 Atributos de variantes (Variant)

Los atributos **`variant`** se utilizarán para especificar una modificación particular del plan de codificación vigente, en lugar de reemplazarlo. Los atributos variant difieren de otros atributos, porque están estrechamente relacionados con los atributos de codificación. Por ese motivo, para los atributos variant se aplican reglas de revocación adicionales (véase 28.5.1).

EJEMPLO:

```
module MyTTCNmodule1
{   :
    type charstring MyType;       // Se codifica normalmente aplicando la 'regla de
                                  // codificación global'
    :
    group MyRecords
    {     :
          type record MyPDU1
          {
          integer     field1,     // field1 se codifica según la regla 'Rule 2' utilizando
                                  // la variante de codificación 'length form 3'
          Mytype      field3      // field3 se codifica según la regla 'Rule 2' utilizando
                                  // cualquier formato posible de longitud de codificación
          }
          with { variant (field1) "length form 3" }
          :
    }
    with { encode "Rule 2" }

}
with { encode "Global encoding rule" }
```

28.2.3 Cadenas especiales

Las siguientes cadenas son los atributos **variant** predefinidos (normalizados) para los tipos básicos simples (véase E.2.1):

a) "8 bit" y "unsigned 8 bit" cuando se aplican a tipos enteros y enumerados, significan que el valor entero o los números enteros asociados a las enumeraciones serán tratados como si estuvieran representados en 8 bits (tipo simple) en el sistema.

b) "16 bit" y "unsigned 16 bit" cuando se aplican a tipos enteros y enumerados, significan que el valor entero o los números enteros asociados a las enumeraciones serán tratados como si estuvieran representados en 16 bits (dos bytes) en el sistema.

c) "32 bit" y "unsigned 32 bit" cuando se aplican a tipos enteros y enumerados, significan que el valor entero o los números enteros asociados a las enumeraciones serán tratados como si estuvieran representados en 32 bits (cuatro bytes) en el sistema.

d) "64 bit" y "unsigned 64 bit" cuando se aplican a tipos enteros y enumerados, significan que el valor entero o los números enteros asociados a las enumeraciones serán tratados como si estuvieran representados en 64 bits (ocho bytes) en el sistema.

e) "IEEE754 float","IEEE754 double", "IEEE754 extended float" e "IEEE754 extended double" cuando se aplican a un tipo con coma decimal, significan que el valor se deberá codificar y decodificar conforme a la norma IEEE 754 (véase la bibliografía).

Las siguientes cadenas son los atributos **variant** predefinidos (normalizados) para **charstring** y **universal charstring** (véase E.2.2):

a) "UTF-8" cuando se aplica al tipo de cadena de caracteres universal, significa que cada carácter del valor se deberá codificar y decodificar conforme al formato de transformación UCS 8 (UTF-8) como se define en el anexo R de ISO/CEI 10646 [10].

b) "UCS-2" cuando se aplica al tipo de cadena de caracteres universal, significa que cada carácter del valor se deberá codificar y decodificar conforme al formato de representación codificada UCS-2 (véase 14.1 de ISO/CEI 10646 [10]).

c) "UTF-16" cuando se aplica al tipo de cadena de caracteres universal, significa que cada carácter del valor se deberá codificar y decodificar conforme al formato de transformación UCS 16 (UTF-16) como se define en el anexo Q de ISO/CEI 10646 [10].

d) "8 bit" cuando se aplica a tipos de cadena de caracteres o cadena de caracteres universal, significan que cada carácter del valor se deberá codificar y decodificar conforme al formato de representación codificada especificado en ISO/CEI 8859 (codificación de 8 bits).

La siguiente cadena es el atributo **variant** predefinido (normalizado) para los tipos estructurados (véase E.2.3):

a) "IDL:fixed FORMAL/01-12-01 v.2.6" cuando se aplica a un tipo record, significa que el valor será tratado como valor IDL de coma decimal fija (véase la bibliografía).

Estos atributos de variantes se pueden utilizar combinados con los atributos de codificación más generales especificados en un nivel más alto. Por ejemplo, si se especifica **universal charstring** con el atributo **variant** "UTF-8" en un módulo que tiene de por sí un atributo de codificación global "BER:1997" (véase 12.2/Z.146 [6]), cada uno de los caracteres de la cadena será codificado inicialmente conforme a las reglas UTF-8, y después este valor UTF-8 será codificado aplicando las reglas BER más globales.

28.2.4 Codificaciones no válidas

Es posible especificar reglas de codificaciones no válidas, pero en una fuente externa al módulo designada por referencia, como se hace referencia a reglas de codificación válidas.

28.3 Atributos de extensión

Todos los elementos de lenguaje de TTCN-3 pueden tener atributos de **extension** especificados por el usuario.

NOTA – Como los atributos definidos por el usuario no están normalizados, puede haber diferencias de interpretación de estos atributos entre las herramientas, y es posible que algunas herramientas no los soporten.

28.4 Ámbito de los atributos

Una instrucción **with** puede asociar atributos a un solo elemento de lenguaje. También es posible asociar atributos a varios elementos del lenguaje, por ejemplo creando una lista de campos de un tipo estructurado en una instrucción de atributo asociada a una sola definición de tipo, o asociando una instrucción **with** a la unidad de ámbito o a un **group** de elementos de lenguaje del contexto.

EJEMPLO:

```
// MyPDU1 se visualizará como PDU
type record MyPDU1 { … } with { display "PDU"}

// MyPDU2 se visualizará como PDU con el atributo de extensión MyRule particular
// de la aplicación
type record MyPDU2 { … }
with
{
    display "PDU";
    extension "MyRule"
}

// Esta definición de grupo …
group MyPDUs {
    type record MyPDU3 { … }
    type record MyPDU4 { … }
}
with {display "PDU"}        // Todos los tipos del grupo MyPDUs se visualizarán como PDU
                            // es idéntica a la siguiente
group MyPDUs {
    type record MyPDU3 { … } with { display "PDU"}
    type record MyPDU4 { … } with { display "PDU"}
}
```

28.5 Reglas de revocación de atributos

Una definición de atributo en una unidad de ámbito de nivel más particular reemplazará a la definición general de atributo de un nivel más general. Las reglas de revocación adicionales para los atributos variant se definen en 28.5.1.

EJEMPLO 1:

```
type record MyRecordA
{
  :
} with { encode "RuleA" }

// En esta expresión, MyRecordA se codifica según la regla RuleA y no según la regla RuleB
// de type record MyRecordB
{
  :
  field  MyRecordA
} with { encode "RuleB" }
```

Una instrucción **with** dentro del alcance de otra instrucción **with** reemplazará a esta última instrucción. La misma regla vale para la utilización de una instrucción **with** con grupos. Hay que tomar precauciones si el mecanismo de reemplazo se utiliza combinado con referencias a definiciones individuales. La regla general es asignar y reemplazar los atributos en el orden de aparición.

```
// Ejemplo de utilización del mecanismo de reemplazo con la instrucción with
group MyPDUs
{
    type record MyPDU1 { … }
    type record MyPDU2 { … }

    group MySpecialPDUs
    {
          type record MyPDU3 { … }
          type record MyPDU4 { … }
    }
    with {extension "MySpecialRule"}    // MyPDU3 y MyPDU4 tendrán atributo de extensión
                                        // MySpecialRule particular de la aplicación
}
```

```
with
{
    display "PDU";          // Todos los tipos del grupo MyPDU se visualizarán
    extension "MyRule";     // como PDU y tendrán el atributo de extensión MyRule
                            // (si no son reemplazados)
}

// es lo mismo que …
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" }
    }
}
```

Una directiva **override** en un ámbito más general reemplazará una definición de atributo de un ámbito más particular.

EJEMPLO 2:

```
type record MyRecordA
{
   :
} with { encode "RuleA" }

// En esta expresión se codifica MyRecordA aplicando la regla RuleB
type record MyRecordB
{
     :
  fieldA    MyRecordA
} with { encode override "RuleB" }
```

La directiva **override** impone un determinado atributo a todos los tipos incluidos de un ámbito más particular.

28.5.1 Reglas de revocación adicionales para atributos variant

El atributo **variant** siempre está relacionado con un atributo **encode**. Una variante de una codificación puede cambiar, pero una codificación no debe hacerlo sin reemplazar todos los atributos de la variante efectivos. Por ese motivo, se aplican las siguientes reglas de revocación para los atributos variant:

- un atributo **variant** reemplaza a un atributo **variant** efectivo conforme a las reglas que se definen en 28.5;
- un atributo **encoding**, que reemplaza a un atributo **encoding** efectivo conforme a las reglas que se definen en 28.5, también reemplaza a un atributo **variant** efectivo correspondiente, es decir, no se proporciona un nuevo atributo **variant**, sino que el atributo **variant** efectivo se desactiva;
- un atributo **encoding**, que cambia un atributo **encoding** efectivo de un elemento de lenguaje importado conforme a las reglas que se definen en 28.6, también cambia un atributo **variant** efectivo correspondiente, es decir, no se proporciona un nuevo atributo **variant**, sino que el atributo **variant** efectivo se desactiva.

EJEMPLO:

```
module MyVariantEncodingModule {
    :
    type charstring MyCharString;       // Se codifica normalmente conforme a Encoding 1"
    :
    group MyVariantsOne {
          :
          type record MyPDUone
          {
          integer           field1,     // field1 se codificará conforme a "Encoding 2"
                                        // únicamente. "Encoding 2" reemplaza a
                                        // "Encoding 1" y la variante 'Variant 1'
          Mytype            field3      // field3 se codificará conforme a "Encoding 1"
                                        // con la variante "Variant 1".
          }
          with { encoding (field1) "Encoding 2" }
          :
    }
```

```
    with { variant "Variant 1" }

    group MyVariantsTwo
    {     :
          type record MyPDUtwo
          {
          integer           field1,     // field1 se codificará conforme a Encoding 3"
                                        // mediante la codificación de la variante
                                        // 'Variant 3'
          Mytype            field3      // field3 se codificará conforme a "Encoding 3"
                                        // mediante la codificación de la variante
                                        // "Variant 2"
          }
          with { variant (field1) "Variant 3" }
          :
    }
    with { encode "Encoding 3"; variant 'Variant 2'}

}
with { encode "Encoding 1" }
```

28.6 Modificar atributos de elementos de lenguaje importados

Los elementos de lenguaje se importan generalmente con sus atributos y en algunos casos hay que modificarlos. Por ejemplo, si un tipo se visualiza como ASP en un módulo, es posible que otro módulo que lo importe tenga que visualizarlo como PDU. En estos casos se permite modificar los atributos en la instrucción **`import`**.

EJEMPLO:

```
import from MyModule {
    type MyType
}
with { display "ASP" }           // MyType se visualizará como ASP

import from MyModule {
    group MyGroup
}
with {
    display "PDU";          // La opción por defecto es visualizar todos los tipos como PDU
    extension "MyRule"
}
```

Anexo A

Forma de Backus-Nauer y semántica estática

A.1 Forma de Backus-Nauer para la notación TTCN-3

A.1.0 Consideraciones generales

En este anexo se define la sintaxis de TTCN-3 con la forma Backus-Nauer ampliada (en adelante denominada BNF).

A.1.1 Convenios para la descripción de la sintaxis

En el cuadro A.1 se describe la metanotación empleada para especificar la gramática BNF ampliada para TTCN-3:

Cuadro A.1/Z.140 – Metanotación sintáctica

::=	significado
abc xyz	abc seguido por xyz
|	alternativa
[abc]	0 ó 1 ejemplar de abc
{abc}	0 o más ejemplares de abc
{abc}+	1 o más ejemplares de abc
(...)	agrupación textual
Abc	el símbolo abc no terminal
"abc"	un símbolo abc terminal

A.1.2 Símbolos finalizadores de instrucciones

En general todas las construcciones de lenguaje TTCN-3 (es decir, definiciones, declaraciones, instrucciones y operaciones) se terminan con un punto y coma (;). El punto y coma es facultativo si la construcción de lenguaje termina con un corchete a la derecha (}) y cuando el siguiente símbolo es un corchete a la derecha (}), es decir, cuando la construcción de lenguaje es el último enunciado en un bloque de instrucciones, operaciones y declaraciones.

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). También se permite utilizar el símbolo de subrayado (_). Un identificador comenzará con una letra (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 se abrirán con el par de símbolos /* y se cerrarán con el par de símbolos */.

EJEMPLO 1:

```
/* Éste es un comentario de bloque
que ocupa dos líneas */
```

No podrá haber un comentario de bloque dentro de otro.

```
/* Este comentario /* no es */ válido */
```

Los comentarios de línea se abrirán con el par de símbolos // y se cerrarán con una <*newline*> (nueva línea).

EJEMPLO 2:

```
// Éste es un comentario de línea
// que ocupa dos líneas
```

Los comentarios de línea pueden seguir a las instrucciones de programa TTCN-3 pero no estarán incluidos en una instrucción.

EJEMPLO 3:

```
// El siguiente comentario no es válido
const // Se trata de MyConst integer MyConst := 1;

// Este comentario es válido
const integer MyConst := 1; // Se trata de MyConst
```

A.1.5 Símbolos terminales de TTCN-3

En los cuadros A.2 y A.3 se indican los símbolos terminales y las 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 de alternativa	**[ ]**
Símbolo "hasta" (en una gama)	**..**
Comentarios de línea y comentarios de bloque	**/* */ //**
Símbolo finalizador de línea/instrucción	**;**
Símbolos de operador aritmético	**+ / -**
Símbolo de operador de concatenación de cadenas	**&**
Símbolos de operador de equivalencia	**!= == >= <=**
Símbolos de delimitación 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 de número de coma flotante	**E**

También son palabras reservadas los identificadores de función predefinidos que se indican en el cuadro 10 y se describen en el anexo C.

Cuadro A.3/Z.140 – Lista de terminales de TTCN-3 que son palabras reservadas

action	fail	noblock	select
activate	false	none	self
address	float		send
alive	for	not	sender
all	from	not4b	set
alt	function	nowait	setverdict
altstep		null	signature
and	getverdict		start
and4b	getcall		stop
any	getreply	octetstring	subset
anytype	goto	of	superset
	group	omit	system
bitstring		on	
boolean	hexstring	optional	template
		or	testcase
case	if	or4b	timeout
call	ifpresent	out	timer
catch	import	override	to
char	in		trigger
charstring	inconc	param	true
check	infinity	pass	type
clear	inout	pattern	
complement	integer	port	union
component	interleave	procedure	universal
connect			unmap
const	kill	raise	
control	killed	read	value
create		receive	valueof
	label	record	var
deactivate	language		variant
default	length		verdicttype
disconnect	log		
display		rem	
do	map	repeat	while
done	match	reply	with
	message	return	
else	mixed	running	xor
encode	mod	runs	xor4b
enumerated	modifies		
error	module		
except	modulepar		
exception	mtc		
execute			
extends			
extension			
external			

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.

A.1.6 Programas ejecutables de BNF en la sintaxis TTCN-3

A.1.6.0 Módulo TTCN-3

```
1. TTCN3Module ::= TTCN3ModuleKeyword TTCN3ModuleId
                   "{"
                   [ModuleDefinitionsPart]
                   [ModuleControlPart]
                   "}"
                   [WithStatement] [SemiColon]
2. TTCN3ModuleKeyword ::= "module"
3. TTCN3ModuleId ::= ModuleId
4. ModuleId ::= GlobalModuleId [LanguageSpec]
/* SEMÁNTICA ESTÁTICA - LanguageSpec sólo puede omitirse si el módulo referenciado incluye la
notación TTCN-3 */
5. GlobalModuleId ::= ModuleIdentifier
6. ModuleIdentifier ::= Identifier
7. LanguageSpec ::= LanguageKeyword FreeText
8. LanguageKeyword ::= "language"
```

A.1.6.1 Parte de definiciones del módulo

A.1.6.1.0 Consideraciones generales

```
9. ModuleDefinitionsPart ::= ModuleDefinitionsList
10. ModuleDefinitionsList ::= {ModuleDefinition [SemiColon]}+
11. ModuleDefinition ::= (TypeDef |
                          ConstDef |
                          TemplateDef |
                          ModuleParDef |
                          FunctionDef |
                          SignatureDef |
                          TestcaseDef |
                          AltstepDef |
                          ImportDef |
                          GroupDef |
                          ExtFunctionDef |
                          ExtConstDef) [WithStatement]
```

A.1.6.1.1 Definiciones de tipos (Typedef)

```
12. TypeDef ::= TypeDefKeyword TypeDefBody
13. TypeDefBody ::= StructuredTypeDef | SubTypeDef
14. TypeDefKeyword ::= "type"
15. StructuredTypeDef ::= RecordDef |
                          UnionDef |
                          SetDef |
                          RecordOfDef |
                          SetOfDef |
                          EnumDef |
                          PortDef |
                          ComponentDef
16. RecordDef ::= RecordKeyword StructDefBody
17. RecordKeyword ::= "record"
18. StructDefBody ::= (StructTypeIdentifier [StructDefFormalParList] | AddressKeyword)
                      "{" [StructFieldDef {"," StructFieldDef}] "}"
19. StructTypeIdentifier ::= Identifier
20. StructDefFormalParList ::= "(" StructDefFormalPar {"," StructDefFormalPar} ")"
21. StructDefFormalPar ::=    FormalValuePar
/* SEMÁNTICA ESTÁTICA - FormalValuePar ha de traducirse en un parámetro in */
22. StructFieldDef ::= (Type | NestedTypeDef) StructFieldIdentifier [ArrayDef] [SubTypeSpec]
                       [OptionalKeyword]
23. NestedTypeDef ::= NestedRecordDef |
                      NestedUnionDef |
                      NestedSetDef |
                      NestedRecordOfDef |
                      NestedSetOfDef |
                      NestedEnumDef
24. NestedRecordDef ::= RecordKeyword "{" [StructFieldDef {"," StructFieldDef}] "}"
25. NestedUnionDef ::= UnionKeyword "{" UnionFieldDef {"," UnionFieldDef} "}"
26. NestedSetDef ::= SetKeyword "{" [StructFieldDef {"," StructFieldDef}] "}"
27. NestedRecordOfDef ::= RecordKeyword [StringLength] OfKeyword (Type | NestedTypeDef)
28. NestedSetOfDef ::= SetKeyword [StringLength] OfKeyword (Type | NestedTypeDef)
29. NestedEnumDef ::= EnumKeyword "{" EnumerationList "}"
30. StructFieldIdentifier ::= Identifier
31. OptionalKeyword ::= "optional"
32. UnionDef ::= UnionKeyword UnionDefBody
33. UnionKeyword ::= "union"
```

```
34. UnionDefBody ::= (StructTypeIdentifier [StructDefFormalParList] | AddressKeyword)
                     "{" UnionFieldDef {"," UnionFieldDef} "}"
35. UnionFieldDef ::= (Type | NestedTypeDef) StructFieldIdentifier [ArrayDef] [SubTypeSpec]
36. SetDef ::= SetKeyword StructDefBody
37. SetKeyword ::= "set"
38. RecordOfDef ::= RecordKeyword [StringLength] OfKeyword StructOfDefBody
39. OfKeyword ::= "of"
40. StructOfDefBody ::= (Type | NestedTypeDef) (StructTypeIdentifier | AddressKeyword) [SubTypeSpec]
41. SetOfDef ::= SetKeyword [StringLength] OfKeyword StructOfDefBody
42. EnumDef ::= EnumKeyword (EnumTypeIdentifier | AddressKeyword)
                "{" EnumerationList "}"
43. EnumKeyword ::= "enumerated"
44. EnumTypeIdentifier ::= Identifier
45. EnumerationList ::= Enumeration {"," Enumeration}
46. Enumeration ::= EnumerationIdentifier ["("[Minus] Number ")"]
47. EnumerationIdentifier ::= Identifier
48. SubTypeDef ::= Type (SubTypeIdentifier | AddressKeyword)  [ArrayDef] [SubTypeSpec]
49. SubTypeIdentifier ::= Identifier
50. SubTypeSpec ::= AllowedValues [StringLength] | StringLength
/* SEMÁNTICA ESTÁTICA - AllowedValues ha de ser del mismo tipo que el campo original */
51. AllowedValues ::= "(" (ValueOrRange {"," ValueOrRange}) | CharStringMatch ")"
52. ValueOrRange ::= RangeDef | ConstantExpression
/* SEMÁNTICA ESTÁTICA - El programa ejecutable de RangeDef sólo se utilizará con tipos basados en
enteros, cadena de caracteres, cadena de caracteres universal o coma flotante */
/* SEMÁNTICA ESTÁTICA - Al definir subtipos de cadena de caracteres o cadena de caracteres
universal, no se han de mezclar gamas y valores en la misma SubTypeSpec */
53. RangeDef ::= LowerBound ".." UpperBound
54. StringLength ::= LengthKeyword "(" SingleConstExpression [".." UpperBound] ")"
/*- SEMÁNTICA ESTÁTICA  StringLength sólo se utilizará con tipos de cadena o para limitar valores
set of o record of. SingleConstExpression y UpperBound deberán traducirse en valores enteros no
negativos (UpperBound puede ser infinito) */
55.     LengthKeyword ::= "length"
56. PortType ::= [GlobalModuleId Dot] PortTypeIdentifier
57. PortDef ::= PortKeyword PortDefBody
58. PortDefBody ::= PortTypeIdentifier PortDefAttribs
59. PortKeyword ::= "port"
60. PortTypeIdentifier ::= Identifier
61. PortDefAttribs ::= MessageAttribs | ProcedureAttribs | MixedAttribs
62. MessageAttribs ::= MessageKeyword
                       "{" {MessageList [SemiColon]}+ "}"
63. MessageList ::= Direction AllOrTypeList
64. Direction ::= InParKeyword | OutParKeyword | InOutParKeyword
65. MessageKeyword ::= "message"
66. AllOrTypeList ::= AllKeyword | TypeList
/* NOTA - Se desaconseja la utilización de AllKeyword en las definiciones de puertos */
67. AllKeyword ::= "all"
68. TypeList ::= Type {"," Type}
69. ProcedureAttribs ::= ProcedureKeyword
                         "{" {ProcedureList  [SemiColon]}+ "}"
70. ProcedureKeyword ::= "procedure"
71. ProcedureList ::= Direction AllOrSignatureList
72. AllOrSignatureList ::= AllKeyword | SignatureList
73. SignatureList ::= Signature {"," Signature}
74. MixedAttribs ::= MixedKeyword
                     "{" {MixedList [SemiColon]}+ "}"
75. MixedKeyword ::= "mixed"
76. MixedList ::= Direction ProcOrTypeList
77. ProcOrTypeList ::= AllKeyword | (ProcOrType {"," ProcOrType})
78. ProcOrType ::= Signature | Type
79. ComponentDef ::= ComponentKeyword ComponentTypeIdentifier
                     [ExtendsKeyword ComponentType {"," ComponentType}]
                     "{" [ComponentDefList] "}"
80. ComponentKeyword ::= "component"
81. ExtendsKeyword ::= "extends"
82. ComponentType ::= [GlobalModuleId Dot] ComponentTypeIdentifier
83. ComponentTypeIdentifier ::= Identifier
84. ComponentDefList ::= {ComponentElementDef [SemiColon]}
85. ComponentElementDef ::= PortInstance | VarInstance | TimerInstance | ConstDef
86. PortInstance ::= PortKeyword PortType PortElement {"," PortElement}
87. PortElement ::= PortIdentifier [ArrayDef]
88. PortIdentifier ::= Identifier
```

A.1.6.1.2 Definiciones de constantes

```
89. ConstDef ::= ConstKeyword Type ConstList
/* SEMÁNTICA ESTÁTICA - El tipo ha de seguir las reglas establecidas en la cláusula 9.*/
90. ConstList ::= SingleConstDef  {"," SingleConstDef}
91. SingleConstDef ::= ConstIdentifier [ArrayDef] AssignmentChar ConstantExpression
/* SEMÁNTICA ESTÁTICA - El valor de ConstantExpression ha de ser del tipo declarado para las
constantes */
```

```
92. ConstKeyword ::= "const"
93. ConstIdentifier ::= Identifier
```

A.1.6.1.3 Definiciones de plantillas

```
94. TemplateDef ::= TemplateKeyword BaseTemplate  [DerivedDef] AssignmentChar TemplateBody
95. BaseTemplate ::= (Type | Signature) TemplateIdentifier  ["(" TemplateFormalParList ")"]
96. TemplateKeyword ::= "template"
97. TemplateIdentifier ::= Identifier
98. DerivedDef ::= ModifiesKeyword TemplateRef
99. ModifiesKeyword ::= "modifies"
100. TemplateFormalParList ::= TemplateFormalPar {"," TemplateFormalPar}
101. TemplateFormalPar ::= FormalValuePar | FormalTemplatePar
/* SEMÁNTICA ESTÁTICA - FormalValuePar deberá traducirse en un parámetro */
102. TemplateBody ::= (SimpleSpec | FieldSpecList | ArrayValueOrAttrib) | [ExtraMatchingAttributes]
/* SEMÁNTICA ESTÁTICA - En TemplateBody se puede utilizar ArrayValueOrAttrib para tipos array,
record, record of y set of. */
103. SimpleSpec ::= SingleValueOrAttrib
104. FieldSpecList ::= "{"[FieldSpec {"," FieldSpec}] "}"
105. FieldSpec ::= FieldReference AssignmentChar TemplateBody
106. FieldReference ::= StructFieldRef | ArrayOrBitRef | ParRef
/* SEMÁNTICA ESTÁTICA - En FieldReference se puede utilizar ArrayOrBitRef para plantillas record of
y set of /campos de plantilla sólo en las plantilla modificadas*/
107. StructFieldRef ::= StructFieldIdentifier| PredefinedType | TypeReference
/* SEMÁNTICA ESTÁTICA - PredefinedType y TypeReference se han de utilizar sólo para la notación de
valor anytype. PredefinedType no debe ser AnyTypeKeyword.*/
108. ParRef ::= SignatureParIdentifier
/* SEMÁNTICA ESTÁTICA - SignatureParIdentifier ha de ser un parámetro formal Identifier de la
definición de firma asociada */
109. SignatureParIdentifier ::= ValueParIdentifier
110. ArrayOrBitRef ::= "[" FieldOrBitNumber "]"
/* SEMÁNTICA ESTÁTICA - ArrayRef se ha de utilizar (es facultativo) para tipos matriz, los tipos SET
OF y SEQUENCE OF (ASN.1) y los tipos record of y set of (TTCN-3). Se puede utilizar la misma
notación para una referencia de bit dentro de un tipo cadena de bits de ASN.1 o TTCN-3 */
111. FieldOrBitNumber ::= SingleExpression
/* SEMÁNTICA ESTÁTICA - SingleExpression deberá traducirse en un valor de tipo entero */
112. SingleValueOrAttrib ::= MatchingSymbol |
                             SingleExpression |
                             TemplateRefWithParList
/* SEMÁNTICA ESTÁTICA - VariableIdentifier (accesible a través de singleExpression) sólo se puede
utilizar en definiciones de plantilla en línea para referenciar variables en el ámbito actual */
113. ArrayValueOrAttrib ::= "{" ArrayElementSpecList "}"
114. ArrayElementSpecList ::= ArrayElementSpec {"," ArrayElementSpec}
115. ArrayElementSpec ::= NotUsedSymbol | PermutationMatch | TemplateBody
116. NotUsedSymbol ::= Dash
117. MatchingSymbol ::= Complement |
                        AnyValue |
                        AnyOrOmit |
                        ValueOrAttribList |
                        Range |
                        BitStringMatch |
                        HexStringMatch |
                        OctetStringMatch |
                        CharStringMatch |
                        SubsetMatch |
                        SupersetMatch
118. ExtraMatchingAttributes ::= LengthMatch | IfPresentMatch | LengthMatch  IfPresentMatch
119. BitStringMatch ::= "'" {BinOrMatch} "'" "B"
120. BinOrMatch ::= Bin | AnyValue | AnyOrOmit
121. HexStringMatch ::= "'" {HexOrMatch} "'" "H"
122. HexOrMatch ::= Hex | AnyValue | AnyOrOmit
123. OctetStringMatch ::= "'" {OctOrMatch} "'" "O"
124. OctOrMatch ::= Oct | AnyValue | AnyOrOmit
125. CharStringMatch ::= PatternKeyword Cstring
126. PatternKeyword ::= "pattern"
127. Complement ::= ComplementKeyword ValueList
128. ComplementKeyword ::= "complement"
129. ValueList ::= "(" ConstantExpression {"," ConstantExpression} ")"
130. SubsetMatch ::= SubsetKeyword ValueList
/* SEMÁNTICA ESTÁTICA - La concordancia de subconjunto sólo se utilizará con el tipo set of */
131. SubsetKeyword ::= "subset"
132. SupersetMatch ::= SupersetKeyword ValueList
/* SEMÁNTICA ESTÁTICA - La concordancia de superconjunto sólo se utilizará con el tipo set of */
133. SupersetKeyword ::= "superset"
134. PermutationMatch ::= PermutationKeyword PermutationList
135. PermutationKeyword ::= "permutation"
136. PermutationList ::= "(" TemplateBody { "," TemplateBody } ")"
/* SEMÁNTICA ESTÁTICA - Las restricciones del contenido de TemplateBody se indican en la
cláusula B.1.3.3 */
```

```
137. AnyValue ::= "?"
138. AnyOrOmit ::= "*"
139. ValueOrAttribList ::= "(" TemplateBody {"," TemplateBody}+ ")"
140. LengthMatch ::= StringLength
141. IfPresentMatch ::= IfPresentKeyword
142. IfPresentKeyword ::= "ifpresent"
143. Range ::= "(" LowerBound ".." UpperBound ")"
144. LowerBound ::= SingleConstExpression | Minus InfinityKeyword
145. UpperBound ::= SingleConstExpression | InfinityKeyword
/* SEMÁNTICA ESTÁTICA – LowerBound y UpperBound deberán traducirse en tipos integer, charstring,
universal charstring o float. Si LowerBound o UpperBound se traducen en tipos charstring o universal
charstring, sólo puede aparecer SingleConstExpression y la longitud de la cadena será 1*/
146. InfinityKeyword ::= "infinity"
147. TemplateInstance ::= InLineTemplate
148. TemplateRefWithParList ::= [GlobalModuleId Dot] TemplateIdentifier [TemplateActualParList] |
                                TemplateParIdentifier
149. TemplateRef ::= [GlobalModuleId Dot] TemplateIdentifier | TemplateParIdentifier
150. InLineTemplate ::= [(Type | Signature) Colon] [DerivedRefWithParList AssignmentChar]
                        TemplateBody
/* SEMÁNTICA ESTÁTICA – El campo de tipo sólo puede omitirse si el tipo está definido implícitamente
sin ambigüedad */
151. DerivedRefWithParList ::= ModifiesKeyword TemplateRefWithParList
152. TemplateActualParList ::= "(" TemplateActualPar {"," TemplateActualPar} ")"
153. TemplateActualPar ::= TemplateInstance
/* SEMÁNTICA ESTÁTICA – Si el parámetro formal correspondiente no es del tipo plantilla, el
enunciado de TemplateInstance deberá traducirse en una o más SingleExpressions */
154. TemplateOps ::= MatchOp | ValueofOp
155. MatchOp ::= MatchKeyword "(" Expression "," TemplateInstance")"
/* SEMÁNTICA ESTÁTICA – El valor devuelto por la expresión ha de ser del mismo tipo que la
plantilla, y cada campo de la plantilla deberá traducirse en un solo valor */
156. MatchKeyword ::= "match"
157. ValueofOp ::= ValueofKeyword "(" TemplateInstance")"
158. ValueofKeyword ::= "valueof"
```

A.1.6.1.4 Definiciones de funciones

```
159. FunctionDef ::= FunctionKeyword FunctionIdentifier
                     "("[FunctionFormalParList] ")" [RunsOnSpec] [ReturnType]
                     StatementBlock
160. FunctionKeyword ::= "function"
161. FunctionIdentifier ::= Identifier
162. FunctionFormalParList ::= FunctionFormalPar {"," FunctionFormalPar}
163. FunctionFormalPar ::= FormalValuePar |
                           FormalTimerPar |
                           FormalTemplatePar |
                           FormalPortPar
164. ReturnType ::= ReturnKeyword [TemplateKeyword] Type
/* SEMÁNTICA ESTÁTICA – La utilización de la palabra clave template ha de ser conforme con las
restricciones que se establecen en la cláusula 16.1.0 */
165. ReturnKeyword ::= "return"
166. RunsOnSpec ::= RunsKeyword OnKeyword ComponentType
167. RunsKeyword ::= "runs"
168. OnKeyword ::= "on"
169. MTCKeyword ::= "mtc"
170. StatementBlock ::= "{" [FunctionStatementOrDefList] "}"
171. FunctionStatementOrDefList ::= {FunctionStatementOrDef [SemiColon]}+
172. FunctionStatementOrDef ::= FunctionLocalDef |
                                FunctionLocalInst |
                                FunctionStatement
173. FunctionLocalInst ::= VarInstance | TimerInstance
174. FunctionLocalDef ::= ConstDef | TemplateDef
175. FunctionStatement ::= ConfigurationStatements |
                           TimerStatements |
                           CommunicationStatements |
                           BasicStatements |
                           BehaviourStatements |
                           VerdictStatements |
                           SUTStatements
176. FunctionInstance ::= FunctionRef "(" [FunctionActualParList] ")"
177. FunctionRef ::= [GlobalModuleId Dot] (FunctionIdentifier | ExtFunctionIdentifier ) |
                     PreDefFunctionIdentifier
178. PreDefFunctionIdentifier ::= Identifier
/* SEMÁNTICA ESTÁTICA – Se utilizará uno de los identificadores de funciones predefinidas de TTCN-3
del anexo C */
179. FunctionActualParList ::= FunctionActualPar {"," FunctionActualPar}
180. FunctionActualPar ::= TimerRef |
                           TemplateInstance |
                           Port |
                           ComponentRef
```

```
/* SEMÁNTICA ESTÁTICA - Si el parámetro formal correspondiente no es de tipo plantilla, el enunciado
de TemplateInstance deberá traducirse en una o más SingleExpressions, es decir, es equivalente al
enunciado de Expression */
```

A.1.6.1.5 Definiciones de firmas

```
181. SignatureDef ::= SignatureKeyword SignatureIdentifier
                      "("[SignatureFormalParList] ")" [ReturnType | NoBlockKeyword]
                      [ExceptionSpec]
182. SignatureKeyword ::= "signature"
183. SignatureIdentifier ::= Identifier
184. SignatureFormalParList ::= SignatureFormalPar {"," SignatureFormalPar}
185. SignatureFormalPar ::=  FormalValuePar
186. ExceptionSpec ::= ExceptionKeyword "(" ExceptionTypeList ")"
187. ExceptionKeyword ::= "exception"
188. ExceptionTypeList ::= Type {"," Type}
189. NoBlockKeyword ::= "noblock"
190. Signature ::= [GlobalModuleId Dot] SignatureIdentifier
```

A.1.6.1.6 Definiciones de casos de prueba

```
191. TestcaseDef ::= TestcaseKeyword TestcaseIdentifier
                     "("[TestcaseFormalParList] ")" ConfigSpec
                     StatementBlock
192. TestcaseKeyword ::= "testcase"
193. TestcaseIdentifier ::= Identifier
194. TestcaseFormalParList ::= TestcaseFormalPar {"," TestcaseFormalPar}
195. TestcaseFormalPar ::= FormalValuePar |
                           FormalTemplatePar
196. ConfigSpec ::= RunsOnSpec [SystemSpec]
197. SystemSpec ::= SystemKeyword ComponentType
198. SystemKeyword ::= "system"
199. TestcaseInstance ::= ExecuteKeyword "(" TestcaseRef "(" [TestcaseActualParList] ")"
                          ["," TimerValue] ")"
200. ExecuteKeyword ::= "execute"
201. TestcaseRef ::= [GlobalModuleId Dot] TestcaseIdentifier
202. TestcaseActualParList ::= TestcaseActualPar {"," TestcaseActualPar}
203. TestcaseActualPar ::= TemplateInstance
/* SEMÁNTICA ESTÁTICA - Si el parámetro formal correspondiente no es de tipo plantilla, el enunciado
de TemplateInstance deberá traducirse en una o más SingleExpressions, es decir, es equivalente al
enunciado de Expression */
```

A.1.6.1.7 Definiciones de alternativas (altsteps)

```
204. AltstepDef ::= AltstepKeyword AltstepIdentifier
                    "("[AltstepFormalParList] ")" [RunsOnSpec]
                    "{" AltstepLocalDefList AltGuardList "}"
205. AltstepKeyword ::= "altstep"
206. AltstepIdentifier ::= Identifier
207. AltstepFormalParList ::= FunctionFormalParList
/* SEMÁNTICA ESTÁTICA - las altsteps que se activen como opciones por defecto tendrán únicamente
parámetros in, port o timer */
/* SEMÁNTICA ESTÁTICA - las altsteps que se invocan sólo como alternativa en una instrucción alt o
como una instrucción autónoma en una descripción de comportamiento TTCN-3 pueden incluir parámetros
in, out e inout. */
208. AltstepLocalDefList ::= {AltstepLocalDef [SemiColon]}
209. AltstepLocalDef ::= VarInstance | TimerInstance | ConstDef | TemplateDef
/*SEMÁNTICA ESTÁTICA - AltstepLocalDef debe ser conforme con las instrucciones establecidas en la
cláusula  16.2.2.1 */
210. AltstepInstance ::= AltstepRef "(" [FunctionActualParList] ")"
/* SEMÁNTICA ESTÁTICA - todos los ejemplares de timer en FunctionActualParList han de declararse
como temporizadores locales de componente (véase tambien el enunciado de ComponentElementDef) */
211. AltstepRef ::= [GlobalModuleId Dot] AltstepIdentifier
```

A.1.6.1.8 Definiciones de importación

```
212. ImportDef ::= ImportKeyword ImportFromSpec (AllWithExcepts | ("{" ImportSpec "}"))
213. ImportKeyword ::= "import"
214. AllWithExcepts ::= AllKeyword [ExceptsDef]
215. ExceptsDef ::= ExceptKeyword "{" ExceptSpec "}"
216. ExceptKeyword ::= "except"
217. ExceptSpec ::= {ExceptElement [SemiColon]}
/* SEMÁNTICA ESTÁTICA - Los componentes del programa (ExceptGroupSpec, ExceptTypeDefSpec etc.) sólo
pueden aparecer una vez en el enunciado de ExceptSpec */
```

```
218. ExceptElement ::= ExceptGroupSpec |
                       ExceptTypeDefSpec |
                       ExceptTemplateSpec |
                       ExceptConstSpec |
                       ExceptTestcaseSpec |
                       ExceptAltstepSpec |
                       ExceptFunctionSpec |
                       ExceptSignatureSpec |
                       ExceptModuleParSpec
219. ExceptGroupSpec ::= GroupKeyword (ExceptGroupRefList | AllKeyword)
220. ExceptTypeDefSpec ::= TypeDefKeyword (TypeRefList | AllKeyword)
221. ExceptTemplateSpec ::= TemplateKeyword (TemplateRefList | AllKeyword)
222. ExceptConstSpec ::= ConstKeyword (ConstRefList | AllKeyword)
223. ExceptTestcaseSpec ::= TestcaseKeyword (TestcaseRefList | AllKeyword)
224. ExceptAltstepSpec ::= AltstepKeyword (AltstepRefList | AllKeyword)
225. ExceptFunctionSpec ::= FunctionKeyword (FunctionRefList | AllKeyword)
226. ExceptSignatureSpec ::= SignatureKeyword  (SignatureRefList | AllKeyword)
227. ExceptModuleParSpec ::= ModuleParKeyword  (ModuleParRefList | AllKeyword)
228. ImportSpec ::= {ImportElement [SemiColon]}
229. ImportElement ::= ImportGroupSpec |
                       ImportTypeDefSpec |
                       ImportTemplateSpec |
                       ImportConstSpec |
                       ImportTestcaseSpec |
                       ImportAltstepSpec |
                       ImportFunctionSpec |
                       ImportSignatureSpec |
                       ImportModuleParSpec
230. ImportFromSpec ::= FromKeyword ModuleId [RecursiveKeyword]
/* NOTA - Se desaconseja la utilización de RecursiveKeyword */
231. RecursiveKeyword ::= "recursive"
232. ImportGroupSpec ::= GroupKeyword (GroupRefListWithExcept | AllGroupsWithExcept)
233. GroupRefList ::= FullGroupIdentifier {"," FullGroupIdentifier}
234. GroupRefListWithExcept ::= FullGroupIdentifierWithExcept {"," FullGroupIdentifierWithExcept}
235. AllGroupsWithExcept ::= AllKeyword [ExceptKeyword GroupRefList]
236. FullGroupIdentifier ::= GroupIdentifier {Dot GroupIdentifier}
237. FullGroupIdentifierWithExcept ::= FullGroupIdentifier [ExceptsDef]
238. ExceptGroupRefList ::= ExceptFullGroupIdentifier {"," ExceptFullGroupIdentifier}
239. ExceptFullGroupIdentifier ::= FullGroupIdentifier
240. ImportTypeDefSpec ::= TypeDefKeyword (TypeRefList | AllTypesWithExcept)
241. TypeRefList ::= TypeDefIdentifier {"," TypeDefIdentifier}
242. AllTypesWithExcept ::= AllKeyword [ExceptKeyword TypeRefList]
243. TypeDefIdentifier ::= StructTypeIdentifier |
                          EnumTypeIdentifier |
                          PortTypeIdentifier |
                          ComponentTypeIdentifier |
                          SubTypeIdentifier
244. ImportTemplateSpec ::= TemplateKeyword (TemplateRefList | AllTemplsWithExcept)
245. TemplateRefList ::= TemplateIdentifier {"," TemplateIdentifier}
246. AllTemplsWithExcept ::= AllKeyword [ExceptKeyword TemplateRefList]
247. ImportConstSpec ::= ConstKeyword (ConstRefList | AllConstsWithExcept)
248. ConstRefList ::= ConstIdentifier {"," ConstIdentifier}
249. AllConstsWithExcept ::= AllKeyword [ExceptKeyword ConstRefList]
250. ImportAltstepSpec ::= AltstepKeyword (AltstepRefList | AllAltstepsWithExcept)
251. AltstepRefList ::= AltstepIdentifier {"," AltstepIdentifier}
252. AllAltstepsWithExcept ::= AllKeyword [ExceptKeyword AltstepRefList]
253. ImportTestcaseSpec ::= TestcaseKeyword (TestcaseRefList | AllTestcasesWithExcept)
254. TestcaseRefList ::= TestcaseIdentifier {"," TestcaseIdentifier}
255. AllTestcasesWithExcept ::= AllKeyword [ExceptKeyword TestcaseRefList]
256. ImportFunctionSpec ::= FunctionKeyword (FunctionRefList | AllFunctionsWithExcept)
257. FunctionRefList ::= FunctionIdentifier {"," FunctionIdentifier}
258. AllFunctionsWithExcept ::= AllKeyword [ExceptKeyword FunctionRefList]
259. ImportSignatureSpec ::= SignatureKeyword  (SignatureRefList | AllSignaturesWithExcept)
260. SignatureRefList ::= SignatureIdentifier {"," SignatureIdentifier}
261. AllSignaturesWithExcept ::= AllKeyword [ExceptKeyword SignatureRefList]
262. ImportModuleParSpec ::= ModuleParKeyword  (ModuleParRefList | AllModuleParWithExcept)
263. ModuleParRefList ::= ModuleParIdentifier {"," ModuleParIdentifier}
264. AllModuleParWithExcept ::= AllKeyword [ExceptKeyword ModuleParRefList]
```

A.1.6.1.9 Definiciones de grupos

```
265. GroupDef ::= GroupKeyword GroupIdentifier
                  "{" [ModuleDefinitionsPart] "}"
266. GroupKeyword ::= "group"
267. GroupIdentifier ::= Identifier
```

A.1.6.1.10 Definiciones de funciones externas

```
268. ExtFunctionDef ::= ExtKeyword FunctionKeyword ExtFunctionIdentifier
                        "("[FunctionFormalParList] ")" [ReturnType]
269. ExtKeyword ::= "external"
270. ExtFunctionIdentifier ::= Identifier
```

A.1.6.1.11 Definiciones de constantes externas

```
271. ExtConstDef ::= ExtKeyword ConstKeyword Type ExtConstIdentifier
/* SEMÁNTICA ESTÁTICA - El tipo ha de seguir las reglas establecidas en la cláusula 9 */
272. ExtConstIdentifier ::= Identifier
```

A.1.6.1.12 Definiciones de parámetros de módulo

```
273. ModuleParDef ::= ModuleParKeyword ( ModulePar | ("{" MultitypedModuleParList "}"))
274. ModuleParKeyword ::= "modulepar"
275. MultitypedModuleParList ::= { ModulePar SemiColon }+
276. ModulePar ::= ModuleParType ModuleParList
/* SEMÁNTICA ESTÁTICA - El valor de ConstantExpression ha de ser del tipo declarado para el
parámetro */
277. ModuleParType ::= Type
/* SEMÁNTICA ESTÁTICA - El tipo no debe ser component, default o anytype. El tipo se traducirá a un
tipo address sólo si se incluye una definición de tipo address en el módulo */
278. ModuleParList ::= ModuleParIdentifier [AssignmentChar ConstantExpression]
                      {","ModuleParIdentifier [AssignmentChar ConstantExpression]}
279. ModuleParIdentifier ::= Identifier
```

A.1.6.2 Parte de control

A.1.6.2.0 Consideraciones generales

```
280. ModuleControlPart ::= ControlKeyword
                           "{" ModuleControlBody "}"
                           [WithStatement] [SemiColon]
281. ControlKeyword ::= "control"
282. ModuleControlBody ::= [ControlStatementOrDefList]
283. ControlStatementOrDefList ::= {ControlStatementOrDef [SemiColon]}+
284. ControlStatementOrDef ::= FunctionLocalDef |
                               FunctionLocalInst |
                               ControlStatement
285. ControlStatement ::= TimerStatements |
                          BasicStatements |
                          BehaviourStatements |
                          SUTStatements |
                          StopKeyword
/* SEMÁNTICA ESTÁTICA - Las restricciones relativas a la utilización de instrucciones en la parte de
control se establecen en el cuadro 11 */
```

A.1.6.2.1 Especificación de ejemplares de variables

```
286. VarInstance ::= VarKeyword ((Type VarList) | (TemplateKeyword Type TempVarList))
287. VarList ::= SingleVarInstance {"," SingleVarInstance}
288. SingleVarInstance ::= VarIdentifier [ArrayDef] [AssignmentChar VarInitialValue]
289. VarInitialValue ::= Expression
290. VarKeyword ::= "var"
291. VarIdentifier ::= Identifier
292. TempVarList ::= SingleTempVarInstance {"," SingleTempVarInstance}
293. SingleTempVarInstance ::= VarIdentifier [ArrayDef] [AssignmentChar TempVarInitialValue]
294. TempVarInitialValue ::= TemplateBody
295. VariableRef ::= (VarIdentifier | ValueParIdentifier) [ExtendedFieldReference]
```

A.1.6.2.2 Especificación de ejemplares de temporizadores

```
296. TimerInstance ::= TimerKeyword TimerList
297. TimerList ::= SingleTimerInstance{"," SingleTimerInstance}
298. SingleTimerInstance ::= TimerIdentifier [ArrayDef] [AssignmentChar TimerValue]
299. TimerKeyword ::= "timer"
300. TimerIdentifier ::= Identifier
301. TimerValue ::= Expression
/* SEMÁNTICA ESTÁTICA - Si Expression se traduce en SingleExpression, ha de ser un valor de tipo
float. Expression sólo podrá traducirse en CompoundExpression en la inicialización, en la asignación
del valor de temporización por defecto para matrices de temporización */
302. TimerRef ::= (TimerIdentifier | TimerParIdentifier) {ArrayOrBitRef}
```

A.1.6.2.3 Operaciones de componentes

```
303. ConfigurationStatements ::= ConnectStatement |
                                 MapStatement |
                                 DisconnectStatement |
                                 UnmapStatement |
                                 DoneStatement |
                                 KilledStatement |
                                 StartTCStatement |
                                 StopTCStatement |
                                 KillTCStatement
304. ConfigurationOps ::= CreateOp | SelfOp | SystemOp | MTCOp | RunningOp | AliveOp
305. CreateOp ::= ComponentType Dot CreateKeyword ["(" SingleExpression ")"] [AliveKeyword]
/* SEMÁNTICA ESTÁTICA - Las restricciones relacionadas con SingleExpression se establecen en la
cláusula 22.1 */
306. SystemOp ::= SystemKeyword
307. SelfOp ::= "self"
308. MTCOp ::= MTCKeyword
309. DoneStatement ::= ComponentId Dot DoneKeyword
310. KilledStatement ::= ComponentId Dot KilledKeyword
311. ComponentId ::= ComponentOrDefaultReference | (AnyKeyword | AllKeyword) ComponentKeyword
312. DoneKeyword ::= "done"
313. KilledKeyword ::= "killed"
314. RunningOp ::= ComponentId Dot RunningKeyword
315. RunningKeyword ::= "running"
316. AliveOp ::= ComponentId Dot AliveKeyword
317. CreateKeyword ::= "create"
318. AliveKeyword ::= "alive"
319. ConnectStatement ::= ConnectKeyword SingleConnectionSpec
320. ConnectKeyword ::= "connect"
321. SingleConnectionSpec ::= "(" PortRef "," PortRef ")"
322. PortRef ::= ComponentRef Colon Port
323. ComponentRef ::= ComponentOrDefaultReference | SystemOp | SelfOp | MTCOp
324. DisconnectStatement ::= DisconnectKeyword [SingleOrMultiConnectionSpec]
325. SingleOrMultiConnectionSpec ::= SingleConnectionSpec |
                                    AllConnectionsSpec |
                                    AllPortsSpec |
                                    AllCompsAllPortsSpec]
326. AllConnectionsSpec ::= "(" PortRef ")"
327. AllPortsSpec ::= "(" ComponentRef ":" AllKeyword PortKeyword ")"
328. AllCompsAllPortsSpec ::= "(" AllKeyword ComponentKeyword ":" AllKeyword PortKeyword ")"
329. DisconnectKeyword ::= "disconnect"
330. MapStatement ::= MapKeyword SingleConnectionSpec
331. MapKeyword ::= "map"
332. UnmapStatement ::= UnmapKeyword [SingleOrMultiConnectionSpec]
333. UnmapKeyword ::= "unmap"
334. StartTCStatement ::= ComponentOrDefaultReference Dot StartKeyword "(" FunctionInstance ")"
/* SEMÁNTICA ESTÁTICA - El ejemplar de Function sólo puede tener parámetros in */
/* SEMÁNTICA ESTÁTICA - El ejemplar de Function no podrá tener parámetros de temporización */
335. StartKeyword ::= "start"
336. StopTCStatement ::= StopKeyword | (ComponentReferenceOrLiteral Dot StopKeyword) |
                         (AllKeyword ComponentKeyword Dot StopKeyword)
337. ComponentReferenceOrLiteral ::= ComponentOrDefaultReference | MTCOp | SelfOp
338. KillTCStatement ::= KillKeyword | (ComponentIdentifierOrLiteral Dot KillKeyword) |
                         (AllKeyword ComponentKeyword Dot KillKeyword)
339. ComponentOrDefaultReference ::= VariableRef | FunctionInstance
/* SEMÁNTICA ESTÁTICA - La variable asociada a VariableRef o el tipo return asociado a
FunctionInstance han de ser del tipo component cuando se emplea en instrucciones de configuración, y
del tipo default cuando se emplea en la instrucción deactivate*/
340. KillKeyword ::= "kill"
```

A.1.6.2.4 Operaciones de puertos

```
341. Port ::= (PortIdentifier | PortParIdentifier) {ArrayOrBitRef}
342. CommunicationStatements ::= SendStatement |
                                 CallStatement |
                                 ReplyStatement |
                                 RaiseStatement |
                                 ReceiveStatement |
                                 TriggerStatement |
                                 GetCallStatement |
                                 GetReplyStatement |
                                 CatchStatement |
                                 CheckStatement |
                                 ClearStatement |
                                 StartStatement |
                                 StopStatement
343. SendStatement ::= Port Dot PortSendOp
344. PortSendOp ::= SendOpKeyword "(" SendParameter ")" [ToClause]
```

```
345. SendOpKeyword ::= "send"
346. SendParameter ::= TemplateInstance
347. ToClause ::= ToKeyword AddressRef |
                  AddressRefList |
                  AllKeyword ComponentKeyword
/* SEMÁNTICA ESTÁTICA - AddressRef no debería incluir mecanismos de concordancia */
348. AddressRefList ::= "(" AddressRef {"," AddressRef} ")"
349. ToKeyword ::= "to"
350. AddressRef ::= TemplateInstance
/* SEMÁNTICA ESTÁTICA - TemplateInstance ha de ser del tipo address o component */
351. CallStatement ::= Port Dot PortCallOp  [PortCallBody]
352. PortCallOp ::= CallOpKeyword "(" CallParameters ")" [ToClause]
353. CallOpKeyword ::= "call"
354. CallParameters ::= TemplateInstance ["," CallTimerValue]
/* SEMÁNTICA ESTÁTICA - Los parámetros out son los únicos que se pueden omitir o especificar con un
atributo de concordancia */
355. CallTimerValue ::= TimerValue | NowaitKeyword
/* SEMÁNTICA ESTÁTICA - El valor ha de ser de tipo float */
356. NowaitKeyword ::= "nowait"
357. PortCallBody ::= "{" CallBodyStatementList "}"
358. CallBodyStatementList ::= {CallBodyStatement [SemiColon]}+
359. CallBodyStatement ::= CallBodyGuard  StatementBlock
360. CallBodyGuard ::= AltGuardChar CallBodyOps
361. CallBodyOps ::= GetReplyStatement | CatchStatement
362. ReplyStatement ::= Port Dot PortReplyOp
363. PortReplyOp ::= ReplyKeyword "(" TemplateInstance [ReplyValue]")" [ToClause]
364. ReplyKeyword ::= "reply"
365. ReplyValue ::= ValueKeyword Expression
366. RaiseStatement ::= Port Dot PortRaiseOp
367. PortRaiseOp ::= RaiseKeyword "(" Signature "," TemplateInstance ")" [ToClause]
368. RaiseKeyword ::= "raise"
369. ReceiveStatement ::= PortOrAny Dot PortReceiveOp
370. PortOrAny ::= Port | AnyKeyword PortKeyword
371. PortReceiveOp ::= ReceiveOpKeyword ["(" ReceiveParameter ")"] [FromClause] [PortRedirect]
/* SEMÁNTICA ESTÁTICA - La opción PortRedirect sólo puede aparecer si también aparece la opción
ReceiveParameter */
372. ReceiveOpKeyword ::= "receive"
373. ReceiveParameter ::= TemplateInstance
374. FromClause ::= FromKeyword AddressRef
375. FromKeyword ::= "from"
376. PortRedirect ::= PortRedirectSymbol (ValueSpec [SenderSpec] | SenderSpec)
377. PortRedirectSymbol ::= "->"
378. ValueSpec ::= ValueKeyword VariableRef
379. ValueKeyword ::= "value"
380. SenderSpec ::= SenderKeyword VariableRef
/* SEMÁNTICA ESTÁTICA - Variable ref ha de ser del tipo de la dirección o del componente */
381. SenderKeyword ::= "sender"
382. TriggerStatement ::= PortOrAny Dot PortTriggerOp
383. PortTriggerOp ::= TriggerOpKeyword ["(" ReceiveParameter ")"] [FromClause] [PortRedirect]
/* SEMÁNTICA ESTÁTICA - La opción PortRedirect sólo puede aparecer si también aparece la opción
ReceiveParameter */
384. TriggerOpKeyword ::= "trigger"
385. GetCallStatement ::= PortOrAny  Dot PortGetCallOp
386. PortGetCallOp ::= GetCallOpKeyword ["(" ReceiveParameter ")"] [FromClause]
                       [PortRedirectWithParam]
/* SEMÁNTICA ESTÁTICA - La opción PortRedirectWithParam sólo puede aparecer si también aparece la
opción ReceiveParameter */
387. GetCallOpKeyword ::= "getcall"
388. PortRedirectWithParam ::= PortRedirectSymbol RedirectWithParamSpec
389. RedirectWithParamSpec ::= ParamSpec [SenderSpec] |
                               SenderSpec
390. ParamSpec ::= ParamKeyword ParamAssignmentList
391. ParamKeyword ::= "param"
392. ParamAssignmentList ::= "(" (AssignmentList | VariableList) ")"
393. AssignmentList ::= VariableAssignment {"," VariableAssignment}
394. VariableAssignment ::= VariableRef AssignmentChar ParameterIdentifier
/* SEMÁNTICA ESTÁTICA - Es obligatorio utilizar parameterIdentifiers de la definición de firma
correspondiente */
395. ParameterIdentifier ::= ValueParIdentifier
396. VariableList ::= VariableEntry  {"," VariableEntry}
397. VariableEntry ::= VariableRef | NotUsedSymbol
398. GetReplyStatement ::= PortOrAny Dot PortGetReplyOp
399. PortGetReplyOp ::= GetReplyOpKeyword ["(" ReceiveParameter [ValueMatchSpec] ")"]
                        [FromClause] [PortRedirectWithValueAndParam]
/* SEMÁNTICA ESTÁTICA - La opción PortRedirectWithParam sólo puede aparecer si también aparece la
opción ReceiveParameter */
400. PortRedirectWithValueAndParam ::= PortRedirectSymbol RedirectWithValueAndParamSpec
401. RedirectWithValueAndParamSpec ::= ValueSpec [ParamSpec] [SenderSpec] |
                                       RedirectWithParamSpec
402. GetReplyOpKeyword ::= "getreply"
```

```
403. ValueMatchSpec ::= ValueKeyword TemplateInstance
404. CheckStatement ::= PortOrAny Dot PortCheckOp
405. PortCheckOp ::= CheckOpKeyword ["(" CheckParameter ")"]
406. CheckOpKeyword ::= "check"
407. CheckParameter ::= CheckPortOpsPresent | FromClausePresent | RedirectPresent
408. FromClausePresent ::= FromClause [PortRedirectSymbol SenderSpec]
409. RedirectPresent ::= PortRedirectSymbol SenderSpec
410. CheckPortOpsPresent ::= PortReceiveOp | PortGetCallOp | PortGetReplyOp | PortCatchOp
411. CatchStatement ::= PortOrAny Dot PortCatchOp
412. PortCatchOp ::= CatchOpKeyword ["("CatchOpParameter ")"] [FromClause] [PortRedirect]
/* SEMÁNTICA ESTÁTICA - La opción PortRedirect sólo puede aparecer si también aparece la opción
CatchOpParameter */
413. CatchOpKeyword ::= "catch"
414. CatchOpParameter ::= Signature "," TemplateInstance | TimeoutKeyword
415. ClearStatement ::= PortOrAll Dot PortClearOp
416. PortOrAll ::= Port | AllKeyword PortKeyword
417. PortClearOp ::= ClearOpKeyword
418. ClearOpKeyword ::= "clear"
419. StartStatement ::= PortOrAll Dot PortStartOp
420. PortStartOp ::= StartKeyword
421. StopStatement ::= PortOrAll Dot PortStopOp
422. PortStopOp ::= StopKeyword
423. StopKeyword ::= "stop"
424. AnyKeyword ::= "any"
```

A.1.6.2.5 Operaciones de temporizadores

```
425. TimerStatements ::= StartTimerStatement | StopTimerStatement | TimeoutStatement
426. TimerOps ::= ReadTimerOp | RunningTimerOp
427. StartTimerStatement ::= TimerRef Dot StartKeyword ["(" TimerValue ")"]
428. StopTimerStatement ::= TimerRefOrAll Dot StopKeyword
429. TimerRefOrAll ::= TimerRef | AllKeyword TimerKeyword
430. ReadTimerOp ::= TimerRef Dot ReadKeyword
431. ReadKeyword ::= "read"
432. RunningTimerOp ::= TimerRefOrAny  Dot RunningKeyword
433. TimeoutStatement ::= TimerRefOrAny  Dot TimeoutKeyword
434. TimerRefOrAny ::= TimerRef | AnyKeyword TimerKeyword
435. TimeoutKeyword ::= "timeout"
```

A.1.6.3 Tipos

```
436. Type ::= PredefinedType | ReferencedType
437. PredefinedType ::= BitStringKeyword |
                        BooleanKeyword |
                        CharStringKeyword |
                        UniversalCharString |
                        IntegerKeyword |
                        OctetStringKeyword |
                        HexStringKeyword |
                        VerdictTypeKeyword |
                        FloatKeyword |
                        AddressKeyword |
                        DefaultKeyword |
                        AnyTypeKeyword
438. BitStringKeyword ::= "bitstring"
439. BooleanKeyword ::= "boolean"
440. IntegerKeyword ::= "integer"
441. OctetStringKeyword ::= "octetstring"
442. HexStringKeyword ::= "hexstring"
443. VerdictTypeKeyword ::= "verdicttype"
444. FloatKeyword ::= "float"
445. AddressKeyword ::= "address"
446. DefaultKeyword ::= "default"
447. AnyTypeKeyword ::= "anytype"
448. CharStringKeyword ::= "charstring"
449. UniversalCharString ::= UniversalKeyword CharStringKeyword
450. UniversalKeyword ::= "universal"
451. ReferencedType ::= [GlobalModuleId Dot] TypeReference [ExtendedFieldReference]
452. TypeReference ::= StructTypeIdentifier[TypeActualParList] |
                       EnumTypeIdentifier |
                       SubTypeIdentifier |
                       ComponentTypeIdentifier
453. TypeActualParList ::= "(" TypeActualPar {"," TypeActualPar} ")"
454. TypeActualPar ::= ConstantExpression
455. ArrayDef ::= {"[" ArrayBounds [".." ArrayBounds] "]"}+
456. ArrayBounds ::= SingleConstExpression
/* SEMÁNTICA ESTÁTICA - ArrayBounds se traducirá en un valor no negativo de tipo entero */
```

A.1.6.4 Valores

```
457. Value ::= PredefinedValue | ReferencedValue
458. PredefinedValue ::= BitStringValue |
                         BooleanValue |
                         CharStringValue |
                         IntegerValue |
                         OctetStringValue |
                         HexStringValue |
                         VerdictTypeValue |
                         EnumeratedValue |
                         FloatValue |
                         AddressValue |
                         OmitValue
459. BitStringValue ::= Bstring
460. BooleanValue ::= "true" | "false"
461. IntegerValue ::= Number
462. OctetStringValue ::= Ostring
463. HexStringValue ::= Hstring
464. VerdictTypeValue ::= "pass" | "fail" | "inconc" | "none" | "error"
465. EnumeratedValue ::= EnumerationIdentifier
466. CharStringValue ::= Cstring | Quadruple
467. Quadruple ::= CharKeyword "(" Group "," Plane "," Row "," Cell ")"
468. CharKeyword ::= "char"
469. Group ::= Number
470. Plane ::= Number
471. Row ::= Number
472. Cell ::= Number
473. FloatValue ::= FloatDotNotation | FloatENotation
474. FloatDotNotation ::= Number Dot DecimalNumber
475. FloatENotation ::= Number [Dot DecimalNumber] Exponential [Minus] Number
476. Exponential ::= "E"
477. ReferencedValue ::= ValueReference [ExtendedFieldReference]
478. ValueReference ::= [GlobalModuleId Dot] (ConstIdentifier | ExtConstIdentifier |
                        ModuleParIdentifier ) |
                        ValueParIdentifier |
                        VarIdentifier
479. Number ::= (NonZeroNum {Num}) | "0"
480. NonZeroNum ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
481. DecimalNumber ::= {Num}+
482. Num ::= "0" | NonZeroNum
483. Bstring ::= "'" {Bin} "'" "B"
484. Bin ::= "0" | "1"
485. Hstring ::= "'" {Hex} "'" "H"
486. Hex ::= Num | "A" | "B" | "C" | "D" | "E" | "F"| "a" | "b" | "c" | "d" | "e" | "f"
487. Ostring ::= "'" {Oct} "'" "O"
488. Oct ::= Hex Hex
489. Cstring ::= """ {Char} """
490. Char ::= /* REFERENCIA – Un carácter determinado por el tipo CharacterString pertinente. Para
charstring: un carácter del juego definido en la Rec. UIT-T T.50. Para universal charstring: un
carácter de cualquier juego definido en ISO/CEI 10646 */
491. Identifier ::= Alpha{AlphaNum | Underscore}
492. Alpha ::= UpperAlpha | LowerAlpha
493. AlphaNum ::= Alpha | Num
494. 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"
495. 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"
496. ExtendedAlphaNum ::= REFERENCIA – Un carácter gráfico de los juegos LATIN BASICO o SUPLEMENTO
LATIN-1 SUPPLEMENT definidos en ISO/CEI 10646  (desde el carácter (0,0,0,32) hasta (0,0,0,126),
desde el carácter (0,0,0,161) hasta (0,0,0,172) y desde el carácter (0,0,0,174) hasta (0,0,0,255))
*/
497. FreeText ::= """ {ExtendedAlphaNum} """
498. AddressValue ::= "null"
499. OmitValue ::= OmitKeyword
500. OmitKeyword ::= "omit"
```

A.1.6.5 Parametrización

```
501. InParKeyword ::= "in"
502. OutParKeyword ::= "out"
503. InOutParKeyword ::= "inout"
504. FormalValuePar ::= [(InParKeyword | InOutParKeyword | OutParKeyword)] Type ValueParIdentifier
505. ValueParIdentifier ::= Identifier
506. FormalPortPar ::= [InOutParKeyword] PortTypeIdentifier PortParIdentifier
507. PortParIdentifier ::= Identifier
508. FormalTimerPar ::= [InOutParKeyword] TimerKeyword TimerParIdentifier
509. TimerParIdentifier ::= Identifier
```

```
510. FormalTemplatePar ::= [(InParKeyword | OutParKeyword | InOutParKeyword )]
                           TemplateKeyword Type TemplateParIdentifier
511. TemplateParIdentifier ::= Identifier
```

A.1.6.6 La instrucción With

```
512. WithStatement ::= WithKeyword WithAttribList
513. WithKeyword ::= "with"
514. WithAttribList ::= "{" MultiWithAttrib "}"
515. MultiWithAttrib ::= {SingleWithAttrib [SemiColon]}
516. SingleWithAttrib ::= AttribKeyword  [OverrideKeyword] [AttribQualifier] AttribSpec
517. AttribKeyword ::= EncodeKeyword |
                       VariantKeyword |
                       DisplayKeyword |
                       ExtensionKeyword
518. EncodeKeyword ::= "encode"
519. VariantKeyword ::= "variant"
520. DisplayKeyword ::= "display"
521. ExtensionKeyword ::= "extension"
522. OverrideKeyword ::= "override"
523. AttribQualifier ::= "(" DefOrFieldRefList ")"
524. DefOrFieldRefList ::= DefOrFieldRef {"," DefOrFieldRef}
525. DefOrFieldRef ::= DefinitionRef | FieldReference | AllRef
/* SEMÁNTICA ESTÁTICA - DefOrFieldRef ha de referirse a una definición o un campo incluidos dentro
del módulo, el grupo o la definición a los que se asocia la instrucción with */
526. DefinitionRef ::= StructTypeIdentifier |
                      EnumTypeIdentifier |
                      PortTypeIdentifier |
                      ComponentTypeIdentifier |
                      SubTypeIdentifier |
                      ConstIdentifier |
                      TemplateIdentifier |
                      AltstepIdentifier |
                      TestcaseIdentifier |
                      FunctionIdentifier |
                      SignatureIdentifier |
                      VarIdentifier |
                      TimerIdentifier |
                      PortIdentifier |
                      ModuleParIdentifier |
                      FullGroupIdentifier
527. AllRef ::= ( GroupKeyword AllKeyword [ExceptKeyword "{" GroupRefList "}"]) |
                ( TypeDefKeyword AllKeyword [ExceptKeyword "{" TypeRefList "}"]) |
                ( TemplateKeyword AllKeyword [ExceptKeyword "{" TemplateRefList "}"]) |
                ( ConstKeyword AllKeyword [ExceptKeyword "{" ConstRefList "}"]) |
                ( AltstepKeyword AllKeyword [ExceptKeyword "{" AltstepRefList "}"]) |
                ( TestcaseKeyword AllKeyword [ExceptKeyword "{" TestcaseRefList "}"]) |
                ( FunctionKeyword AllKeyword [ExceptKeyword "{" FunctionRefList "}"]) |
                ( SignatureKeyword AllKeyword [ExceptKeyword "{" SignatureRefList "}"]) |
                ( ModuleParKeyword AllKeyword [ExceptKeyword "{" ModuleParRefList "}"])
528. AttribSpec ::= FreeText
```

A.1.6.7 Instrucciones de comportamiento

```
529. BehaviourStatements ::= TestcaseInstance |
                             FunctionInstance |
                             ReturnStatement |
                             AltConstruct |
                             InterleavedConstruct |
                             LabelStatement |
                             GotoStatement |
                             RepeatStatement |
                             DeactivateStatement |
                             AltstepInstance |
                             ActivateOp
/* SEMÁNTICA ESTÁTICA - No se podrá solicitar TestcaseInstance desde un caso de prueba existente en
ejecución ni desde una cadena de función solicitada por un caso de prueba; en otras palabras, los
casos de prueba sólo se pueden ejemplificar desde la parte de control o desde funciones solicitadas
directamente desde la parte de control */
/* SEMÁNTICA ESTÁTICA - No se podrá solicitar ActivateOp desde la parte de control del módulo */
530. VerdictStatements ::= SetLocalVerdict
531. VerdictOps ::= GetLocalVerdict
532. SetLocalVerdict ::= SetVerdictKeyword "(" SingleExpression ")"
/* SEMÁNTICA ESTÁTICA - SingleExpression ha de traducirse en un valor de tipo veredicto */
/* SEMÁNTICA ESTÁTICA - SetLocalVerdict no se ha utilizar para asignar el valor error */
533. SetVerdictKeyword ::= "setverdict"
534. GetLocalVerdict ::= "getverdict"
535. SUTStatements ::= ActionKeyword "(" [ActionText ] {StringOp ActionText} ")"
536. ActionKeyword ::= "action"
```

```
537. ActionText ::= FreeText | Expression
/*SEMÁNTICA ESTÁTICA - Expression ha de tener el tipo básico charstring o universal charstring */
538. ReturnStatement ::= ReturnKeyword  [Expression]
539. AltConstruct ::= AltKeyword "{" AltGuardList "}"
540. AltKeyword ::= "alt"
541. AltGuardList ::= {GuardStatement | ElseStatement [SemiColon]}
542. GuardStatement ::= AltGuardChar (AltstepInstance [StatementBlock] | GuardOp StatementBlock)
543. ElseStatement ::= "["ElseKeyword "]" StatementBlock
544. AltGuardChar ::= "[" [BooleanExpression] "]"
/*SEMÁNTICA ESTÁTICA - BooleanExpression ha de ser conforme a las restricciones establecidas en la
cláusula 20.1.2 */
545. GuardOp ::= TimeoutStatement |
                 ReceiveStatement |
                 TriggerStatement |
                 GetCallStatement |
                 CatchStatement |
                 CheckStatement |
                 GetReplyStatement |
                 DoneStatement |
                 KilledStatement
/* SEMÁNTICA ESTÁTICA – GuardOp utilizado en la parte de control del módulo ha de contener
únicamente la instrucción timeoutStatement */
546. InterleavedConstruct ::= InterleavedKeyword "{" InterleavedGuardList "}"
547. InterleavedKeyword ::= "interleave"
548. InterleavedGuardList ::= {InterleavedGuardElement [SemiColon]}+
549. InterleavedGuardElement ::= InterleavedGuard InterleavedAction
550. InterleavedGuard ::= "[" "]" GuardOp
551. InterleavedAction ::= StatementBlock
/* SEMÁNTICA ESTÁTICA – StatementBlock no puede contener instrucciones de bucle, instrucciones goto,
activate, deactivate, stop o return, ni solicitudes de funciones */
552. LabelStatement ::= LabelKeyword LabelIdentifier
553. LabelKeyword ::= "label"
554. LabelIdentifier ::= Identifier
555. GotoStatement ::= GotoKeyword LabelIdentifier
556. GotoKeyword ::= "goto"
557. RepeatStatement ::= "repeat"
558. ActivateOp ::= ActivateKeyword "(" AltstepInstance ")"
559. ActivateKeyword ::= "activate"
560. DeactivateStatement ::= DeactivateKeyword ["(" ComponentOrDefaultReference ")"]
561. DeactivateKeyword ::= "deactivate"
```

A.1.6.8 Instrucciones básicas

```
562. BasicStatements ::= Assignment | LogStatement | LoopConstruct | ConditionalConstruct |
                         SelectCaseConstruct
563. Expression ::= SingleExpression | CompoundExpression
/* SEMÁNTICA ESTÁTICA – Expression no podrá tener operaciones de configuración, activación o
veredicto en la parte de control del módulo */
564. CompoundExpression ::= FieldExpressionList | ArrayExpression
/* SEMÁNTICA ESTÁTICA – En el enunciado de CompoundExpression, ArrayExpression puede utilizarse para
tipos Arrays, record, record of y set of. */
565. FieldExpressionList ::= "{" FieldExpressionSpec {"," FieldExpressionSpec} "}"
566. FieldExpressionSpec ::= FieldReference AssignmentChar NotUsedOrExpression
567. ArrayExpression ::= "{" [ArrayElementExpressionList] "}"
568. ArrayElementExpressionList ::= NotUsedOrExpression  {"," NotUsedOrExpression}
569. NotUsedOrExpression ::= Expression | NotUsedSymbol
570. ConstantExpression ::= SingleConstExpression | CompoundConstExpression
571. SingleConstExpression ::= SingleExpression
/* SEMÁNTICA ESTÁTICA – SingleConstExpression no podrá contener Variables ni parámetros de Módulo, y
ha de traducirse en un valor constante en el momento de la compilación */
572. BooleanExpression ::= SingleExpression
/* SEMÁNTICA ESTÁTICA – BooleanExpression ha de traducirse en un valor de tipo Boolean */
573. CompoundConstExpression ::= FieldConstExpressionList | ArrayConstExpression
/* SEMÁNTICA ESTÁTICA – En el enunciado de CompoundConstExpression, ArrayConstExpression puede
utilizarse para tipos Arrays, record, record of y set of. */
574. FieldConstExpressionList ::= "{" FieldConstExpressionSpec {"," FieldConstExpressionSpec} "}"
575. FieldConstExpressionSpec ::= FieldReference AssignmentChar ConstantExpression
576. ArrayConstExpression ::= "{" [ArrayElementConstExpressionList] "}"
577. ArrayElementConstExpressionList ::= ConstantExpression {"," ConstantExpression}
578. Assignment ::= VariableRef AssignmentChar (Expression | TemplateBody)
/* SEMÁNTICA ESTÁTICA -  La expresión a la derecha de Assignment ha de traducirse en un valor
explícito de un tipo compatible con el tipo a la izquierda de las variables de valor, y ha de
traducirse a un valor explícito, plantilla (literal o un ejemplar de plantilla) o a un mecanismo de
concordancia compatible con el tipo a la izquierda de las variables template. */
579. SingleExpression ::= XorExpression { "or" XorExpression }
/* SEMÁNTICA ESTÁTICA – Si hay varias XorExpression, éstas han de traducirse en valores específicos
de tipos compatibles */
580. XorExpression ::= AndExpression { "xor" AndExpression }
/* SEMÁNTICA ESTÁTICA – Si hay varias AndExpression, éstas han de traducirse en valores específicos
de tipos compatibles */
```

```
581. AndExpression ::= NotExpression { "and" NotExpression }
/* SEMÁNTICA ESTÁTICA - Si hay varias NotExpression, éstas han de traducirse en valores específicos
de tipos compatibles */
582. NotExpression ::= [ "not" ] EqualExpression
/* SEMÁNTICA ESTÁTICA - Los operandos del operador no han de ser de tipo booleano (TTCN o ASN.1) o
derivados del tipo booleano */
583. EqualExpression ::= RelExpression { EqualOp RelExpression }
/* SEMÁNTICA ESTÁTICA - Si hay más de una RelExpression, éstas han de traducirse en valores
específicos de tipos compatibles */
584. RelExpression ::= ShiftExpression [ RelOp ShiftExpression ]
/* SEMÁNTICA ESTÁTICA - Si aparecen ambas ShiftExpressions, cada una de ellas ha de traducirse a un
entero específico, valor Enumerated o float (pueden ser valores TTCN o ASN.1) o derivados de esos
tipos */
585. ShiftExpression ::= BitOrExpression { ShiftOp BitOrExpression }
/* SEMÁNTICA ESTÁTICA - Cada Result ha de traducirse en un valor específico. Si hay varios Result,
el operando de la derecha ha de ser del tipo entero o derivados; si el operador de desplazamiento
es '<<' o '>>' el operando de la izquierda ha de traducirse en un tipo bitstring, hexstring u
octetstring o derivados de esos tipos; si el operador de desplazamiento es '<@' o '@>' el operando
de la izquierda ha de ser de tipo bitstring, hexstring, charstring o universal charstring o
derivados de esos tipos */
586. BitOrExpression ::= BitXorExpression { "or4b" BitXorExpression }
/* SEMÁNTICA ESTÁTICA - Si hay varias BitXorExpression, éstas han de traducirse en valores
específicos de tipos compatibles */
587. BitXorExpression ::= BitAndExpression { "xor4b" BitAndExpression }
/* SEMÁNTICA ESTÁTICA - Si hay varias BitAndExpression, éstas han de traducirse en valores
específicos de tipos compatibles */
588. BitAndExpression ::= BitNotExpression { "and4b" BitNotExpression }
/* SEMÁNTICA ESTÁTICA - Si hay varias BitNotExpression, éstas han de traducirse en valores
específicos de tipos compatibles */
589. BitNotExpression ::= [ "not4b" ] AddExpression
/* SEMÁNTICA ESTÁTICA - Si se incluye el operador not4b, el operando ha de ser de tipo bitstring,
octetstring o hexstring o derivados de estos tipos. */
590. AddExpression ::= MulExpression { AddOp MulExpression }
/* SEMÁNTICA ESTÁTICA - Cada MulExpression ha de traducirse en un valor específico. Si hay varios
MulExpression y AddOp se traduce en StringOp, las MulExpression han de traducirse en el mismo tipo
ha de ser bitstring, hexstring, octetstring, charstring o universal charstring o derivados de estos
tipos. Si hay varios MulExpression y AddOp no se traduce en StringOp, los dos MulExpression han de
traducirse en un tipo entero,  de coma flotante o en derivados de estos tipos. */
591. MulExpression ::= UnaryExpression { MultiplyOp UnaryExpression }
/* SEMÁNTICA ESTÁTICA - Cada UnaryExpression ha de traducirse en un valor específico. Si hay varias
UnaryExpression, todas deben traducirse en un tipo entero, de coma flotante o derivados de estos
tipos. */
592. UnaryExpression ::= [ UnaryOp ] Primary
/* SEMÁNTICA ESTÁTICA - Primary ha de traducirse en un valor específico  de tipo integer, float o
derivados de estos tipos.*/
593. Primary ::= OpCall | Value | "(" SingleExpression ")"
594. ExtendedFieldReference ::= {(Dot ( StructFieldIdentifier | TypeDefIdentifier))
                               | ArrayOrBitRef }+
/* SEMÁNTICA ESTÁTICA - TypeDefIdentifier sólo se utilizará si el tipo VarInstance o ReferencedValue
en el que se utiliza ExtendedFieldReference es anytype */.
595. OpCall ::= ConfigurationOps |
                VerdictOps |
                TimerOps |
                TestcaseInstance |
                FunctionInstance |
                TemplateOps |
                ActivateOp
596. AddOp ::= "+" | "-" | StringOp
/* SEMÁNTICA ESTÁTICA - Los operandos de los operadores "+" o "-" han de ser de tipo entero,  coma
flotante  o derivaciones de entero o flotante (es decir, una subgama) */
597. MultiplyOp ::= "*" | "/" | "mod" | "rem"
/* SEMÁNTICA ESTÁTICA - Los operandos de los operadores "*", "/", rem o mod han de ser de tipo
entero, coma flotante  o derivaciones de entero o flotante (es decir, una subgama) */
598. UnaryOp ::= "+" | "-"
/* SEMÁNTICA ESTÁTICA - Los operandos de los operadores "+" o "-" han de ser de tipo entero,  coma
flotante  o derivaciones de entero o flotante (es decir, una subgama) */
599. RelOp ::= "<" | ">" | ">=" | "<="
/* SEMÁNTICA ESTÁTICA - La precedencia de los operadores se define en el cuadro 7 */
600. EqualOp ::= "==" | "!="
601. StringOp ::= "&"
/* SEMÁNTICA ESTÁTICA - Los operandos del operador de cadena han de ser de tipo bitstring,
hexstring, octetstring o character string */
602. ShiftOp ::= "<<" | ">>" | "<@" | "@>"
603. LogStatement ::= LogKeyword "(" LogItem { "," LogItem } ")"
604. LogKeyword ::= "log"
605. LogItem ::= FreeText | TemplateInstance
606. LoopConstruct ::= ForStatement |
                       WhileStatement |
                       DoWhileStatement
```

```
607. ForStatement ::= ForKeyword "(" Initial SemiColon Final SemiColon Step ")"
                      StatementBlock
608. ForKeyword ::= "for"
609. Initial ::= VarInstance | Assignment
610. Final ::= BooleanExpression
611. Step ::= Assignment
612. WhileStatement ::= WhileKeyword "(" BooleanExpression ")"
                        StatementBlock
613. WhileKeyword ::= "while"
614. DoWhileStatement ::= DoKeyword StatementBlock
                          WhileKeyword "(" BooleanExpression ")"
615. DoKeyword ::= "do"
616. ConditionalConstruct ::= IfKeyword "(" BooleanExpression ")"
                              StatementBlock
                              {ElseIfClause}[ElseClause]
617. IfKeyword ::= "if"
618. ElseIfClause ::= ElseKeyword  IfKeyword "(" BooleanExpression ")"  StatementBlock
619. ElseKeyword ::= "else"
620. ElseClause ::= ElseKeyword  StatementBlock
621. SelectCaseConstruct ::= SelectKeyword "(" SingleExpression ")" SelectCaseBody
622. SelectKeyword ::= "select"
623. SelectCaseBody ::= "{" { SelectCase }+ "}"
624. SelectCase ::= CaseKeyword ( '(' TemplateInstance {"," TemplateInstance } ')' | ElseKeyword )
                    StatementBlock
625. CaseKeyword ::= "case"
```

A.1.6.9 Otras instrucciones de programación

```
626. Dot ::= "."
627. Dash ::= "-"
628. Minus ::= Dash
629. SemiColon ::= ";"
630. Colon ::= ":"
631. Underscore ::= "_"
632. AssignmentChar ::= ":="
```

Anexo B

Concordancia de valores entrantes

B.1 Mecanismos de concordancia de plantillas

B.1.0 Consideraciones generales

En este anexo se especifican los mecanismos de concordancia que se pueden utilizar en plantillas TTCN-3 (sólo en plantillas).

B.1.1 Concordancia de valores específicos

El mecanismo básico de las plantillas TTCN-3 es la concordancia de valores específicos. 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 es exactamente igual al valor resultante de la expresión de la plantilla.

EJEMPLO:

```
// Considérese la definición de tipo de mensaje
type record MyMessageType
{
    integer           field1,
    charstring        field2,
    boolean           field3 optional,
    integer[4]        field4
}

// Posible expresión de plantilla de mensaje con valores específicos
template MyMessageType MyTemplate:=
{
    field1 := 3+2,                // valor específico de tipo entero
    field2 := "My string",        // valor específico de tipo charstring
    field3 := true,               // valor específico de tipo booleano
    field4 := {1,2,3}             // valor específico de tipo matriz de enteros
}
```

B.1.1.1 Omisión de valores

La palabra clave **omit** indica que no se deberá incluir un campo de plantilla facultativo. Puede utilizarse con valores de cualquier tipo siempre que el campo de plantilla sea facultativo.

EJEMPLO:

```
template Mymessage MyTemplate:=
{   :
    :
    field3 := omit,         // omitir este campo
    :
}
```

B.1.2 Otros mecanismos de concordancia, no por valores

B.1.2.0 Consideraciones generales

Es posible utilizar los siguientes mecanismos de concordancia en lugar de valores explícitos.

B.1.2.1 Listas de valores

En estas listas se especifican valores entrantes aceptables. Se pueden utilizar para todos los tipos de valores. Un campo de plantilla que utiliza una lista de valores concuerda con el campo entrante correspondiente solamente si el valor de este campo entrante concuerda con cualquiera de los valores de la lista. Todos los valores de la lista han de ser del tipo declarado para el campo de plantilla en el que se utiliza este mecanismo.

EJEMPLO:

```
template Mymessage MyTemplate:=
{
    field1 := (2,4,6),                      // lista de valores integer
    field2 := ("String1", "String2"),       // lista de valores charstring
    :
    :
}
```

B.1.2.2 Lista de valores complemento

La palabra clave **complement** indica una lista de valores que no serán aceptados como valores entrantes (es el complemento de una lista de valores). Se puede utilizar para todos los tipos de valores.

Todos los valores de esta lista han de ser del tipo declarado para el campo de plantilla en el que se utiliza el complemento. Un campo de plantilla que utiliza complemento concuerda con el campo entrante correspondiente solamente si este campo entrante no concuerda con ninguno de los valores incluidos 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 enteros inaceptables
    :
    field3 not(true)        // concordancia con false
    :
}
```

B.1.2.3 Cualquier valor

El símbolo de concordancia "?" (*AnyValue*) se utiliza para indicar que es aceptable cualquier valor entrante válido. Se puede utilizar para todos los tipos de valores. Un campo de plantilla que utilice este mecanismo "cualquier valor" concuerda con el campo entrante correspondiente solamente si este campo entrante se traduce en un solo elemento del tipo especificado.

EJEMPLO:

```
template Mymessage MyTemplate:=
{
    field1 := ?,            // concordancia con cualquier entero
    field2 := ?,            // concordancia con cualquier valor cadena de car.
                            // no vacía
    field3 := ?,            // concordancia con true o false
    field4 := ?             // concordancia con cualquier secuencia de enteros
}
```

B.1.2.4 Cualquier valor o ningún valor

El símbolo de concordancia "*" (*AnyValueOrNone*) se utiliza para indicar que es aceptable cualquier valor entrante válido o la presentación sin ningún valor. Se puede utilizar para todos los tipos de valores siempre que se declare que el campo de plantilla es facultativo.

Un campo de plantilla que utilice este símbolo concordará con el campo entrante correspondiente solamente si el campo entrante se traduce en cualquier elemento del tipo especificado, y cuando no se incluye el campo entrante.

EJEMPLO:

```
template Mymessage MyTemplate:=
{   :
    field3 := *,            // concordancia con true o false, y cuando no se incluya
                            // el campo
    :
}
```

B.1.2.5 Gama de valores

Las gamas son series de valores aceptables entre dos valores límite de tipo entero o coma flotante (**integer** o **float**) (así como los subtipos de integer y float). Los valores límite han de ser:

a) infinito o –infinito;

b) una expresión que se traduzca en un valor específico entero o de coma flotante.

El límite inferior aparecerá en el lado izquierdo de la gama y el límite superior a la derecha. Es condición que el límite inferior sea menor que el límite superior. Un campo de plantilla que utilice una gama concordará con el campo entrante correspondiente solamente si el valor de este campo entrante es igual a uno de los valores de la gama.

Si los valores límite se utilizan en plantillas o campos de plantillas de tipos **charstring** o **universal charstring**, deberán traducirse en posiciones de caracteres válidas conforme a la tabla de caracteres codificados de ese tipo (por ejemplo, esta posición no podrá estar vacía). Las posiciones vacías entre los límites inferior y superior no se consideran valores válidos de la gama especificada.

EJEMPLO:

```
template Mymessage MyTemplate:=
{
    field1 := (1 .. 6),     // gama de tipo integer
    :
    :
    :
}
// para field1 también se podría indicar (-infinity a 8) o (12 a infinity)
```

B.1.2.6 Superconjunto (SuperSet)

La operación de concordancia SuperSet sólo se utilizará para valores de tipo **set of**. Se utiliza la palabra clave **superset**. Los campos que utilizan la opción SuperSet concuerdan con el campo entrante correspondiente solamente si este campo entrante contiene todos los elementos definidos en SuperSet (puede contener más). El argumento de SuperSet ha de ser del tipo declarado para el campo en el que se utiliza el mecanismo SuperSet.

EJEMPLO:

```
type set of integer MySetOfType;

template MySetOfType MyTemplate1 := superset ( 1, 2, 3 );
// concuerda con cualquier secuencia de enteros en la que aparezcan al menos
// una vez los números 1, 2 y 3 en cualquier orden y cualquier posición
```

B.1.2.7 Subconjunto (SubSet)

La operación de concordancia SubSet sólo se utilizará para valores de tipo **set of**. Se utiliza la palabra clave **subset**.

Los campos que utilizan la opción SubSet concuerdan con el campo entrante correspondiente solamente si este campo entrante contiene exclusivamente elementos definidos en SubSet (puede contener menos). El argumento de SubSet ha de ser del tipo declarado para el campo en el que se utiliza el mecanismo SubSet.

EJEMPLO:

```
template MySetOfType MyTemplate1:= subset ( 1, 2, 3 );
// concuerda con cualquier secuencia de enteros en la que aparezcan una vez
// los números 1, 2 y 3 en cualquier orden y cualquier posición, o en la
// que no aparezcan estos números
```

B.1.3 Mecanismos de concordancia dentro de valores

B.1.3.0 Consideraciones generales

Los siguientes mecanismos de concordancia se pueden utilizar dentro de valores explícitos de cadenas, registros, registros de, conjuntos, conjuntos de y matrices.

B.1.3.1 Cualquier elemento

El símbolo de concordancia "?" (*AnyElement*) se utiliza como representación de un elemento particular de una cadena (excepto cadenas de caracteres), de un registro (**record of**), un conjunto (**set of**) o una matriz. Se utilizará solamente dentro de valores de los tipos cadena, **record of** o **set of** y en matrices.

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** se puede interpretar como cualquier valor entero *(AnyValue)* o cualquier elemento (*AnyElement)* dentro de un tipo **record of** o **set of** o una matriz. No hay problema porque ambas interpretaciones resultan en la misma concordancia.

B.1.3.1.1 Utilización de comodines de un carácter

Si es necesario expresar el comodín "?" en cadenas de caracteres, hay que utilizar el sistema de patrones de caracteres (véase B.1.5). Por ejemplo, las cadenas "abcdxyz", "abccxyz" "abcxxyz" y otras similares concordarán con el patrón (**pattern**) "abc?xyz", pero no las cadenas "abcxyz", "abcdefxyz" ni otras similares.

B.1.3.2 Cualquier número de elementos o ningún elemento

El símbolo de concordancia "*" (*AnyElementsOrNone*) se utiliza como representación de ningún elemento o cualquier número de elementos consecutivos de una cadena (excepto cadenas de caracteres), de un registro (**record of**), un conjunto (**set of**) o una matriz. El símbolo "*" concuerda con la secuencia más larga de elementos posible, de acuerdo con el esquema especificado por los símbolos que rodean el "*".

EJEMPLO:

```
template Mymessage MyTemplate:=
{   :
    field2 := "abcxyz",
    field3 := '10*11'B,          // donde "*" puede ser cualquier secuencia de bits
                                 // (posiblemente vacía)
    field4 := {*, 2, 3}          // donde "*" puede ser cualquier número de valores
                                 // enteros o ningún valor
}

var charstring MyStrings[4];
MyPCO.receive(MyStrings:{"abyz", *, "abc" });
```

Si el símbolo "*" aparece en el nivel más alto dentro de una cadena, un registro (**record of**), un conjunto (**set of**) o una matriz, significa cualquier número de elementos o ninguno (*AnyElementsOrNone)*.

NOTA – Esta regla impide que el símbolo "*" se interprete con el significado del comodín cualquier valor o ninguno (*AnyValueOrNone)* que significa un elemento dentro de una cadena, un registro (**record of**), un conjunto (**set of**) o una matriz.

B.1.3.2.1 Utilización de comodines de múltiples caracteres

Si es necesario expresar el comodín "*" en cadenas de caracteres, hay que utilizar el sistema de patrones de caracteres (véase B.1.5). Por ejemplo, las cadenas "abcxyz", "abcdefxyz" "abcabcxyz" y otras similares concordarán con el patrón (**pattern**) "abc*xyz".

B.1.3.3 Permutación

Se trata de una operación para efectos de concordancia que ha de utilizarse sólo en valores de tipo **record of**. La permutación se designa mediante la palabra clave **permutation**. Los elementos de permutación válidos son: Expressions, *AnyElement* y *AnyElementsOrNone*. Cada elemento enumerado en la permutación ha de ser del tipo calcado por el tipo **record of**.

La permutación en lugar de un solo elemento significa que cualquier serie de elementos es válida, siempre que contenga los mismos elementos de la lista de valores en la permutación, aunque posiblemente en un orden diferente. Si la permutación y *AnyElementsOrNone* se emplean dentro de un valor, ambos serán evaluados conjuntamente.

La utilización de *AnyElementsOrNone* en la permutación permite reemplazar a ninguno o cualquier número de elementos en el segmento del registro del valor concordado por la permutación. Cuando se emplea *AnyElementsOrNone* en una permutación, ésta se evaluará en último lugar (cuando los otros elementos de la lista de la permutación ya han concordado con un elemento en la lista evaluada).

NOTA 1 – La utilización de *AnyElementsOrNone* en una permutación provoca un efecto distinto que cuando se emplea junto con una permutación, ya que en el segundo caso *AnyElementsOrNone* reemplaza sólo a elementos consecutivos. Por ejemplo, {**permutation**(1,2,*)} equivale a ({*,1,*,2,*},{*,2,*,1,*}), mientras que {**permutation**(1,2),*} equivale a ({1,2},{2,1},*).

NOTA 2 – Cuando se utiliza *AnyElementsOrNone* con una permutación podrá aplicarse un atributo de longitud a *AnyElementsOrNone* a fin de restringir el número de elementos que concuerdan con él (véase también B.1.4.1). En caso contrario, no se añadirá un atributo de longitud al *AnyElementsOrNone* empleado dentro de una permutación (no obstante, podrá aplicarse a toda la permutación).

EJEMPLO:

```
type record of integer MySequenceOfType;

template MySequenceOfType MyTemplate1 := { permutation ( 1, 2, 3 ), 5 };
// concuerda con cualquiera de las secuencias de 4 enteros a continuación:
// 1,2,3,5; 1,3,2,5; 2,1,3,5; 2,3,1,5; 3,1,2,5; ó 3,2,1,5

template MySequenceOfType MyTemplate2 := { permutation ( 1, 2, ? ), 5 };
// concuerda con cualquier secuencia de 4 enteros que termine con 5 y
// contenga 1 y 2 al menos una vez en otras posiciones

template MySequenceOfType MyTemplate3 := { permutation ( 1, 2, 3 ), * };
// concuerda con cualquier secuencia de enteros que comience con 1,2,3;
// 1,3,2; 2,1,3; 2,3,1; 3,1,2 ó 3,2,1
```

```
template MySequenceOfType MyTemplate4 := { *, permutation ( 1, 2, 3 )};
// concuerda con cualquier secuencia de enteros que termine con 1,2,3; 1,3,2;
// 2,1,3; 2,3,1; 3,1,2 ó 3,2,1

template MySequenceOfType MyTemplate5 := { *, permutation ( 1, 2, 3 ),* };
// concuerda con cualquier secuencia de enteros que contenga cualquiera de
// las siguientes subcadenas en cualquier posición: 1,2,3; 1,3,2; 2,1,3;
// 2,3,1; 3,1,2 ó 3,2,1

template MySequenceOfType MyTemplate6 := { permutation ( 1, 2, * ), 5 };
// concuerda con cualquier cadena de enteros que termine con 5 y contenga 1 y 2
// al menos una vez en otras posiciones

template MySequenceOfType MyTemplate7 := { permutation ( 1, 2, 3 ), * length (0..5)};
// concuerda con cualquier secuencia de tres a ocho enteros que comiencen con
// 1,2,3; 1,3,2; 2,1,3; 2,3,1; 3,1,2 ó 3,2,1

template MySequenceOfType MyTemplate9 := { permutation ( 1, 2, *) length (3..5), 5 };
// concuerda con cualquier secuencia de cuatro a seis enteros que termine con 5
// y contenga 1 y 2 al menos una vez en otra posición
```

B.1.4 Concordancia de atributos de valores

B.1.4.0 Consideraciones generales

Es posible asociar los siguientes atributos a los mecanismos de concordancia.

B.1.4.1 Restricciones de longitud (length)

Se utiliza para restringir la longitud de valores de cadenas y el número de elementos en una estructura de conjunto (**`set of`**), de registro (**`record of`**) o de matriz. Se utilizará solamente como atributo de los siguientes mecanismos: *AnyValue*, *AnyValueOrNone*, *AnyElement* y *AnyElementsOrNone* (pero no dentro de una permutación), permutación, superconjunto y subconjunto. También se puede utilizar junto con el mecanismo de concordancia complementario y con el atributo **`ifpresent`** ("si se incluye"). La sintaxis de **`length`** se indica en 6.2.3 y 6.3.3.

NOTA – Cuando se utilizan los mecanismos de concordancia complementario y de restricción de longitud para una plantilla o un campo de plantilla, las restricciones introducidas por los mismos han de aplicarse a la plantilla o al campo de plantilla independientemente.

Las unidades de longitud han de ser interpretadas conforme al cuadro 4 en el caso de valores de cadenas. Para los tipos **`set of`** y **`record of`** y las matrices, la unidad de longitud es el tipo calcado. Los límites se indicarán con expresiones que se traduzcan en valores enteros específicos no negativos (**`integer`**). También se puede utilizar la palabra clave **`infinity`** como valor límite superior para indicar que no hay ningún límite de longitud superior.

No podrá haber conflicto entre las especificaciones de longitud para la plantilla y las restricciones de longitud del tipo correspondiente (en su caso). Un campo de plantilla que utilice la longitud como atributo de un símbolo concuerda con el correspondiente campo entrante solamente si este campo entrante concuerda con el símbolo y también con el atributo asociado. El atributo de longitud concuerda si la longitud del campo entrante está entre el límite inferior y el límite superior especificados, incluidos estos valores límite. Cuándo sólo se indica un valor de longitud, el atributo de longitud concuerda solamente si la longitud del campo recibido es exactamente el valor especificado.

Se permite utilizar una restricción de longitud junto con el valor especial **`omit`**, pese a que, en este caso, el atributo de longitud no produce efecto alguno (es decir, con **`omit`** es redundante). En el caso del mecanismo "cualquier valor o ninguno" (*AnyValueOrNone*) con el indicador **`ifpresent`**, el atributo de longitud restringe el valor entrante (en su caso).

EJEMPLO:

```
template Mymessage MyTemplate:=
{
    field1 := complement ({4,5},{1,4,8,9}) length (1 .. 6), // se acepta cualquier valor
                                                            // que contenga 1,2,3,4
    // 5 ó 6 elementos siempre que no sean {4,5} o {1,4,8,9}
    field2 := "ab*ab" length(13)  // la longitud máxima de la cadena AnyElementsOrNone
                                  // es 9 caracteres
    :
}
```

B.1.4.2 El indicador IfPresent ("si se incluye")

El indicador **`ifpresent`** significa que puede haber concordancia si un campo facultativo está presente (si no se ha omitido). Este atributo se puede utilizar con todos los mecanismos de concordancia, siempre que se declare que el tipo es facultativo.

Un campo de plantilla con el indicador **ifpresent** concuerda con el campo entrante correspondiente solamente si este campo entrante satisface el mecanismo de concordancia asociado, y también cuando el campo entrante no se incluye.

EJEMPLO:

```
template Mymessage:MyTemplate:=
{   :
    field2 := "abcd" ifpresent, //concuerda con "abcd" si no se omite
    :
    :
}
```

NOTA – El mecanismo *AnyValueOrNone* tiene exactamente el mismo significado que **? ifpresent**.

B.1.5 Concordancia de patrones de caracteres

B.1.5.0 Consideraciones generales

Se pueden utilizar patrones de caracteres en plantillas para definir el formato de una cadena de caracteres que se ha de recibir. Se pueden utilizar para la concordancia con valores **charstring** y **universal charstring**. En los patrones de caracteres se pueden incluir caracteres literales y metacaracteres (por ejemplo, ? y * dentro de un patrón de caracteres para significar la concordancia con, respectivamente, cualquier carácter y cualquier número de caracteres de cualquier clase).

EJEMPLO 1:

```
template charstring MyTemplate:= pattern "ab??xyz*0";
```

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 caracteres de cualquier clase (incluyendo cualquier número de "0") antes del carácter "0" de cierre.

Para que un metacarácter sea interpretado literalmente hay que colocar antes el metacarácter '\'.

EJEMPLO 2:

```
template charstring MyTemplate:= pattern "ab?\?xyz*";
```

Esta plantilla concordaría con cualquier cadena de caracteres formada por los caracteres 'ab', seguidos por cualquier carácter, seguido por los caracteres '?xyz', seguidos por cualquier número de caracteres de cualquier clase.

En el cuadro B.1 se indican los metacaracteres para patrones de la notación TTCN-3. Los metacaracteres no deben contener espacios en blanco salvo cuando alguno de ellos está precedido por un carácter de nueva línea antes o dentro de una expresión de un conjunto (set expression).

Cuadro B.1/Z.140 – Lista de metacaracteres para patrones TTCN-3

Metacarácter	Descripción
?	Concordancia con cualquier carácter (véanse notas 1 y 2)
*	Concordancia con ningún carácter o con cualquier número de veces de cualquier carácter; debe concordar con el número de caracteres más largo posible (véase el ejemplo 1 anterior) (véanse notas 1 y 2)
\	Hace que se interprete literalmente el metacarácter que sigue (véase nota 3). Cuando precede a un carácter sin significado de metacarácter definido, '\' junto con el carácter concuerdan con el carácter a continuación de '\' (véase nota 4)
[ ]	Concordancia con cualquier carácter del conjunto especificado (más información en B.1.5.1)
-	Sólo tiene significado de metacarácter entre corchetes cuadrados ("[" y "]"), salvo en la primera y última posiciones. Permite especificar una serie de caracteres (más información en B.1.5.1)
^	Sólo tiene un significado de metacarácter como primer carácter, entre corchetes cuadrados, a continuación del corchete de apertura ("[" y "]") y provoca concordancia con cualquier carácter que complemente la serie de caracteres que siguen a este comodín (más información en B.1.5.1)
\q{group,plane,row, cell}	Concordancia con el carácter universal especificado por el cuádruplo
{reference}	Insertar la cadena referenciada definida por el usuario e interpretarla como una expresión regular (más información en B.1.5.2)
\N{reference}	Concordancia con cualquier carácter del conjunto de caracteres, donde el conjunto se define mediante la definición referenciada; (más información en B.1.5.4)

Cuadro B.1/Z.140 – Lista de metacaracteres para patrones TTCN-3

Metacarácter	Descripción
\d	Concordancia con cualquier cifra (lo mismo que [0-9])
\w	Concordancia con cualquier carácter alfanumérico (lo mismo que [0-9a-zA-Z])
\t	Concordancia con el carácter de control HT(9) de C0 (véase ISO/CEI 6429 [11])
\n	Concordancia con cualquiera de los siguientes caracteres de control de C0: LF(10), VT(11), FF(12), CR(13) (véase ISO/CEI 6429 [11]) (se denominan conjuntamente caracteres de nueva línea
\r	Concordancia con el carácter de control CR de C0 (véase ISO/CEI 6429 [11])
\s	Concordancia con cualquiera de los siguientes caracteres de control de C0: HT(9), LF(10), VT(11), FF(12), CR(13), SP(32) (véase ISO/CEI 6429 [11], Rec. UIT-T T.50 [9]) (se denominan conjuntamente caracteres de espacio en blanco)
\b	Concordancia con un límite de palabra (cualquier carácter gráfico, excepto SP o DEL, está precedido o seguido por cualquiera de los caracteres de espacio en blanco o de nueva línea)
\"	Concordancia con el carácter doble comillas
""	Concordancia con el carácter doble comillas
|	Se utiliza para indicar dos expresiones alternativas
( )	Se utiliza para agrupar una expresión
#(n, m)	Concordancia con la expresión anterior al menos n veces, pero no más de m veces (sufijo) (más información en B.1.5.3).
#n	Concordancia con la expresión anterior exactamente n veces (donde n es una sola cifra) (sufijo); idéntico a #(n)
+	Concordancia con la expresión anterior una o varias veces (sufijo); idéntico a #(1,)
NOTA 1 – Los metacaracteres ? y * pueden concordar con cualesquiera caracteres del conjunto de caracteres de tipo raíz de plantilla o de campo de plantilla en el que se utilizan (es decir, sin tener en cuenta las restricciones de tipo que se aplican). No obstante, no debe olvidarse que las operaciones de recepción exigen la comprobación del tipo de mensaje recibido antes de intentar la concordancia con él. Por esa razón nunca se proveen para concordancia, los valores recibidos que no cumplen con la especificación de subtipo de la plantilla o del campo de plantilla. NOTA 2 – En algunos lenguajes/notaciones ? y * tienen diferente significado como metacaracteres. No obstante, en TTCN se suelen utilizar para la concordancia en el sentido que se especifica en este cuadro. NOTA 3 – Por tanto, para indicar concordancia con la barra hacia atrás se escribirán dos barras hacia atrás sin espacio entre ellas (\\), por ejemplo, el patrón '\\d' concordará con la cadena '\d'; la apertura o cierre de paréntesis cuadrados puede concordar con '\[' y '\]' respectivamente, etc. NOTA 4 – Se desaconseja ese empleo del metacarácter '\' pues más adelante se pueden definir otros metacaracteres.	

B.1.5.1 Expresión de un conjunto

Una lista de caracteres delimitada por los símbolos '[' y ']' concuerda (o corresponde) con cualquier carácter en dicha lista. La expresión de un conjunto está delimitada por los símbolos '[' ']'. Se pueden especificar caracteres literales y series de caracteres utilizando el guión '-' como separador. El intervalo está compuesto por los caracteres que se encuentran inmediatamente antes y después del separador, y todos aquellos caracteres cuyo código de carácter está entre los códigos de los dos caracteres de los bordes. Un guión '-' que se encuentra en la lista, pero que no precede o antecede un carácter, pierde su significado especial.

También es posible negar el conjunto utilizando el carácter signo de intercalación '^' en primer lugar después del paréntesis cuadrado de apertura. La negación tiene prioridad sobre cualquier gama de caracteres, por lo que un guión '-' situado inmediatamente después de un signo de intercalación '^' de negación ha de ser procesado como un carácter literal.

No se permiten listas vacías ni listas negadas vacías. En consecuencia, un paréntesis cuadrado de cierre ']' que vaya inmediatamente después de uno de apertura '[', o un signo de intercalación '^' que vaya después de un paréntesis cuadrado de apertura '[' e inmediatamente antes de uno de cierre ']', serán procesados como caracteres literales.

Todos los metacaracteres, salvo los que se enumeran a continuación, pierden su significado especial dentro de la lista:

- ']' si no está en la primera posición ni inmediatamente después de un '^' en la primera posición;
- '-' si no está en la primera ni en la última posición de la lista;
- '^' en la primera posición de la lista, excepto cuando vaya seguido inmediatamente por un paréntesis cuadrado de cierre;

- '\', \d, \t, \w, \r, \n, \s y \b;
- \q{grupo,plano,fila,célula};
- **\ N**{referencia}.

NOTA 1 – No se permiten las listas incrustadas (por ejemplo, en el patrón '[ab[r-z]]' el segundo '[' indica un '[' literal, el primero ']' cierra la lista y el segundo ']' provoca un error al no haber paréntesis de apertura relacionado en el patrón).

NOTA 2 – Para incluir un carácter literal símbolo de intercalación "^", hay que ponerlo en cualquier lugar, salvo en la primera posición o precedido por una barra inclinada inversa. Para incluir un guión "-" literal, hay que ponerlo de primero o de último en la lista, o precedido por una barra inclinada inversa. Para incluir un paréntesis cuadrado de cierre "]" literal, hay que ponerlo de primero o precedido por una barra inclinada inversa. Si el primer carácter de la lista es el símbolo de intercalación "^", los caracteres "-" y "]" también concuerdan ellos mismos cuando van justo después de dicho símbolo.

EJEMPLO:

```
template charstring RegExp1:= pattern '[a-z]';      // concordancia con cualquier carácter
                                                    // entre a y z

template charstring RegExp2:= pattern '[^a-z]';     // concordancia con cualquier carácter,
                                                    // excepto entre a y z

template charstring RegExp3:= pattern '[AC-E][0-9][0-9][0-9]YKE';

// RegExp3 concordará con una cadena que empiece por la letra A o una letra entre C y E
// (aunque no, por ejemplo B)seguidas por tres cifras y las letras YKE
```

B.1.5.2 Expresión de una referencia

Además de valores de cadenas directos, en un patrón se pueden utilizar referencias a plantillas, constantes, variables o parámetros de módulo existentes. La referencia aparece entre los caracteres "{" "}" y debe traducirse en uno de los tipos de cadena de caracteres. Los contenidos de las plantillas, las constantes o las variables a las que se hace referencia han de tratarse como una expresión regular. Cada expresión se dejará de referenciar sólo una vez.

EJEMPLO:

```
const charstring MyString:= "ab?";

template charstring MyTemplate:= pattern '{MyString}';
```

Esta plantilla concordaría con cualquier cadena formada por los caracteres 'ab' seguidos por cualesquiera caracteres. En efecto, las cadenas de caracteres precedidas por la palabra clave **pattern** explícitamente o mediante referencia serán interpretadas conforme a las reglas definidas en esta cláusula.

```
template universal charstring MyTemplate1:= pattern '{MyString}de\q{1, 1, 13, 7}';
```

Esta plantilla concordaría con cualquier cadena formada por los caracteres 'ab' seguidos por cualquier carácter, los caracteres 'de', y el carácter de ISO/CEI 10646 que corresponde a grupo=1, plano=1, fila=13 y célula=7.

Si una expresión de referencia se refiere a una plantilla, constante o variable que contiene una o varias expresiones de referencia, se dejarán de referenciar recursivamente las referencias en la plantilla, constante o variable antes de añadir sus contenidos en el patrón de referencia.

EJEMPLO:

```
const charstring MyConst2 := pattern "ab";
template charstring RegExp1 := pattern "{MyConst2}";
   // matches the string "ab"
template charstring RegExp2 := pattern "{RegExp1}{RegExp1}";
   // matches the string "abab"
template charstring RegExp3 := pattern "c{RegExp2}d";
   // matches the string "cababd"

template charstring RegExp4 := pattern "{Reg";
template charstring RegExp5 := pattern "Exp1}";
template charstring RegExp6 := pattern "{RegExp4}{RegExp5}";
   // matches the string "{RegExp1}" only (i.e. shall not be handled as a
   // reference expression to the template RegExp1)
```

B.1.5.3 Concordancia con una expresión n veces

Se utilizará una de las sintaxis: '#(n, m)', '#(n,)', '#(, m)', '#(n)', '#n' o '+' para indicar que ha de haber un determinado número de concordancias con la expresión que precede. La forma '#(n, m)' especifica al menos n concordancias con la expresión que precede, pero no más de m veces. El metacarácter posfijo '#(n,)' especifica que la expresión precedente debe concordar por lo menos n veces, en tanto que '#(, m)' indica que la expresión precedente puede concordar a lo sumo m veces. Los metacaracteres (posfijos) '#(n)' y '#n' indican que la expresión precedente debe concordar

exactamente n veces (son equivalentes a '#(n, n)'). En la forma '#n' n será una sola cifra. El metacarácter posfijo '+' señala que la expresión precedente debe concordar por lo menos 1 vez (equivalente a '#(1,)').

EJEMPLO:

```
template charstring RegExp4:= pattern '[a-z]#(9, 11)'; // concordancia con mínimo 9 y
                                                       // máximo 11 caracteres de a a z
template charstring RegExp5A:= pattern '[a-z]#(9)';    // concordancia con 9 caracteres
                                                       // de a a z
template charstring RegExp5b:= pattern '[a-z]#9';      // concordancia con  9 caracteres
                                                       // de a a z

template charstring RegExp6:= pattern '[a-z]#(9, )';   // concordancia con mínimo
                                                       // 9 caracteres de a a z
template charstring RegExp7:= pattern '[a-z]#(, 11)';  // concordancia con máximo
                                                       // 11 caracteres de a a z
template charstring RegExp8:= pattern '[a-z]+';        // concordancia con mínimo
                                                       // 1 carácter de a a z
```

B.1.5.4 Concordancia de un juego de caracteres referenciado

Una notación de la forma **"\N{**reference**}"**, donde por reference se entiende una plantilla, constante, variable o parámetro de módulo de un carácter de longitud, concuerda con el carácter en el valor o la plantilla a los que se hace referencia.

La referencia a una plantilla, constante, variable o parámetro de módulo cuya longitud sea diferente de 1, provocará un error.

Una notación de la forma **"\N{**typereference**}"**, donde "typereference" es una referencia al tipo **charstring** o **universal charstring**, corresponde a cualquier carácter del juego de caracteres indicado por el tipo referenciado.

NOTA 1 – Los casos en los que el juego de caracteres referenciado no sea un verdadero subconjunto de valores permitido por la definición del tipo de plantilla o campo de plantilla para el que se emplea el patrón de caracteres, no se tratarán como error (aunque, por ejemplo, nunca habrá concordancia si estos dos juegos tienen una parte común).

NOTA 2 – \N{**charstring**} es equivalente a ?, para una plantilla o campo de plantilla de tipo **charstring**, y \N{**universal charstring**} equivalente a ?, para una plantilla o campo de plantilla de tipo **universal charstring** (no obstante, puede provocar un error si se aplica a una plantilla o campo de plantilla de tipo **charstring**).

EJEMPLO:

```
type charstring MyCharRange ('a'..'z');
type charstring MyCharList ('a', 'z');
const MyCharRange myCharR := 'r';

template charstring myTempPatt1 := pattern '\N { myCharR }';
// myTempPatt1 shall match the string 'r' only

template charstring myTempPatt2 := pattern '\N { MyCharRange }';
// myTempPatt2 shall match any string containing a single character from a to z

template MyCharRange myTempPatt3 := pattern '\N { MyCharList }';
// myTempPatt3 and shall match strings 'a' and 'r' only

template MyCharList myTempPatt4 := pattern '\N { MyCharRange }';
// myTempPatt4 shall match strings 'a' and 'r' only
```

B.1.5.5 Reglas de compatibilidad de tipo para patrones

En caso de patrones referenciados (véase B.1.5.2) y de referencias a los juegos de caracteres (véase B.1.5.4), se aplican reglas específicas de compatibilidad de tipo, a saber: siempre se puede utilizar un tipo, plantilla, constante, variable o módulo de parámetro referenciados del tipo **charstring** en la especificación de patrón de una plantilla o campo de plantilla de tipo **universal charstring**; se puede utilizar un tipo, plantilla o valor referenciados del tipo **universal charstring** en la especificación de patrón de una plantilla o campo de plantilla de tipo **charstring**, si todos los caracteres empleados en la plantilla o valor de referencia y el juego de caracteres permitido por el tipo referenciado tienen sus caracteres correspondientes en el tipo **charstring** (véase la definición de caracteres correspondientes en 6.7.1).

Anexo C

Funciones predefinidas de la notación TTCN-3

En este anexo se definen las funciones predefinidas de la notación TTCN-3.

C.0 Procedimientos generales de tratamiento de excepción

Las situaciones de error (por ejemplo el parámetro de entrada está fuera del intervalo aceptado o es de un tipo erróneo, el valor de entrada contiene caracteres no válidos, etc.) para las que no se definan en las cláusulas pertinentes de este anexo reglas explícitas de tratamiento de excepción, provocarán un error de tiempo de compilación o de tiempo de ejecución TTCN-3. Cuál de ellas provoque cuál tipo de error, se deja a la potestad del implementador.

C.1 Entero a carácter

```
int2char(integer value) return charstring
```

Esta función convierte un valor **integer** de la gama 0 a 127 (codificación de 8 bits) en un valor **charstring** cuya longitud sea un solo carácter. El valor entero describe la codificación de 8 bits del carácter.

C.2 Carácter a entero

```
char2int(charstring value) return integer
```

Esta función convierte un valor **charstring** cuya longitud sea un solo carácter en un valor entero de la gama 0 a 127. El valor entero describe la codificación de 8 bits del carácter.

C.3 Entero a carácter universal

```
int2unichar(integer value) return universal charstring
```

Esta función convierte un valor **integer** de la gama de 0 a 2 147 483 647 (codificación de 32 bits) en un valor **universal charstring** cuya longitud sea un solo carácter. El valor entero describe la codificación de 32 bits del carácter.

C.4 Carácter universal a entero

```
unichar2int(universal charstring value) return integer
```

Esta función convierte un valor **universal charstring** cuya longitud sea un solo carácter en un valor entero de la gama 0 a 2 147 483 647. El valor entero describe la codificación de 32 bits del carácter.

C.5 Cadena de bits a entero

```
bit2int(bitstring value) return integer
```

Esta función convierte un solo valor **bitstring** en un solo valor **integer**.

Se convierte interpretando la **cadena de bits** como un valor **entero** positivo de base 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.

C.6 Cadena hexadecimal a entero

```
hex2int(hexstring value) return integer
```

Esta función convierte un solo valor **hexstring** en un solo valor **integer**.

Se convierte interpretando la **cadena hexadecimal** como un valor **entero** positivo de base 16. 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 a F representan los valores decimales 0 a 15, respectivamente.

C.7 Cadena de octetos a entero

```
oct2int(octetstring value) return integer
```

Esta función convierte un valor **octetstring** en un valor **integer**.

Se convierte interpretando la **cadena de octetos** como un valor **entero** positivo de base 16. 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 ha de ser un múltiplo de 2 porque un octeto se compone de dos dígitos hexadecimales. Los dígitos hexadecimales 0 a F representan los valores decimales 0 a 15, respectivamente.

C.8 Cadena de caracteres a entero

```
str2int(charstring value) return integer
```

Esta función convierte una **charstring** representativa de un valor **integer** al valor **entero** equivalente.

EJEMPLO:

```
str2int("66")         // devolverá el valor entero 66

str2int("-66")        // devolverá el valor entero -66

str2int("abc")        // generará error de compilador o caso de prueba

str2int("0")          // devolverá el valor entero 0
```

C.9 Entero a cadena de bits

```
int2bit(in integer value, in integer length) return bitstring
```

Esta función convierte un solo valor **integer** en un solo valor **bitstring**. La longitud en bits de la cadena resultante viene dada por `length`.

Se convierte interpretando la **cadena de bits** como un valor **entero** positivo de base 2. 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 que tiene menos bits que los especificados en el parámetro `length`, la **bitstring** será rellenada a la izquierda con ceros.

C.10 Entero a cadena hexadecimal

```
int2hex(in integer value, in integer length) return hexstring
```

Esta función convierte un solo valor **integer** en un solo valor **hexstring**. La longitud en dígitos hexadecimales de la cadena resultante viene dada por **length**.

Se convierte interpretando la **cadena hexadecimal** como un valor **entero** positivo de base 16. 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 a F representan los valores decimales 0 a 15 respectivamente. Si la conversión da un valor que tiene menos dígitos hexadecimales que los especificados en el parámetro `length`, la **cadena hexadecimal** se rellenará a la izquierda con ceros.

C.11 Entero a cadena de octetos

```
int2oct(in integer value, in integer length) return octetstring
```

Esta función convierte un solo valor **integer** en un solo valor **octetstring**. La longitud en octetos de la cadena resultante viene dada por `length`.

Se convierte interpretando la **cadena de octetos** como un valor **entero** positivo de base 16. 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 ha de ser un múltiplo de 2 porque un octeto se compone de 2 dígitos hexadecimales. Los dígitos hexadecimales 0 a F representan los valores decimales 0 a 15, respectivamente. Si la conversión da un valor que tiene menos dígitos hexadecimales que los especificados en el parámetro `length`, la **cadena hexadecimal** se rellenará a la izquierda con ceros.

C.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).

EJEMPLO:

```
int2str(66)     // devolverá el valor charstring "66"

int2str(-66)    // devolverá el valor  charstring "-66"

int2str(0)      // devolverá el valor charstring "0"
```

C.13 Longitud del tipo 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.

La longitud de una cadena **universal charstring** deberá calcularse contando separadamente todos los caracteres de combinación y los caracteres de sílabas hangul (incluidos los caracteres de relleno) (véanse ISO/CEI 10646 [10] y cláusulas 23 y 24).

EJEMPLO:

```
lengthof('010'B) // devuelve 3

lengthof('F3'H) // devuelve 2

lengthof('F2'O) // devuelve 1

lengthof (universal charstring : "Length_of_Example") // devuelve 17
```

C.14 Número de elementos en un valor estructurado

```
sizeof(any_type value) return integer
```

Esta función devuelve el número de elementos real de un parámetro de módulo, constante, variable o plantilla de un tipo **record**, **record of**, **set**, **set of** o matriz (véase la nota). En el caso de valores o plantillas **record of** y **set of**, o matrices, el valor efectivo que la función va a devolver es el número secuencial del último elemento definido (índice de ese elemento más 1).

NOTA – Sólo se calculan los elementos del objeto TTCN-3 que es el parámetro de la función, es decir, al determinar el valor que se devuelve no se tienen en cuenta elementos de tipos/valores de un subnivel de jerarquía.

EJEMPLO:

```
// Siendo
type record MyPDU
    {     boolean field1 optional,
          integer field2
    };

template MyPDU MyTemplate
    { field1 omit,
      field2 5
    };

var integer numElements;

// se dan los siguientes casos
numElements := sizeof(MyTemplate);     // devuelve 1

// Siendo

type record length(0..10) of integer MyList;
var MyList MyRecordVar;
MyRecordVar := { 0, 1, omit, 2, omit };

// se dan los siguientes casos
numElements := sizeof(MyRecordVar);
// devuelve 4 aunque el elemento MyRecordVar[2] no está definido
```

C.15 La función IsPresent (está presente)

```
ispresent(any_type value) return boolean
```

Esta función está permitida solamente para los tipos **record** y **set** y devuelve el valor **true** solamente si el valor del campo referenciado está presente en el ejemplar efectivo del objeto de datos referenciado. El argumento de la función **ispresent** será una referencia a un campo de tipo **record** o **set**.

```
// Siendo
type record MyRecord
    {     boolean     field1 optional,
          integer field2
    }
// y dado que MyPDU es una plantilla de tipo MyRecord
// y la PDU recibida también es de tipo MyRecord,
// entonces
MyPort.receive(MyPDU) -> value received_PDU
ispresent(received_PDU.field1)
// devuelve true si field1 está presente en la instancia efectiva de MyPDU
```

C.16 La función IsChosen (está seleccionado)

```
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 ha seleccionado efectivamente para un determinado objeto de datos.

EJEMPLO:

```
// Siendo
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 también es de tipo MyUnion,
// entonces
MyPort.receive(MyPDU) -> value received_PDU
ischosen(received_PDU.p2)
// devuelve true si la instancia efectiva de MyPDU incluye una PDU del tipo PDU_type2
```

C.17 La función Regexp (expresión regular)

```
regexp (any_character_string_type instr, charstring expression, integer groupno) return
character_string_type
```

Esta función devuelve una subcadena de la cadena de caracteres de entrada instr, que corresponde al contenido del enésimo grupo de la expresión. La cadena de entrada instr es una cadena de caracteres de cualquier tipo. La cadena de caracteres devuelta tiene el tipo de raíz de instr. La expresión es un patrón de caracteres conforme a la descripción de B.1.5. El número del grupo que se va a devolver se especifica en `groupno` y ha de ser un entero positivo. Los números de grupo se asignan en el orden en que aparece el corchete de apertura de un grupo y se cuentan a partir de cero con variaciones de 1. Si ninguna subcadena de la cadena de entrada satisface todas las condiciones (patrón y número de grupo), se devuelve una cadena vacía.

EJEMPLO:

```
// Siendo
var charstring mypattern2 := '
var charstring myinput := '      date: 2001-10-20 ;  msgno: 17; exp '
var charstring mypattern := ' [ /t]#(,)date:[ \d\-]#(,);[ /t]#(,)msgno:    (\d#(1,3));
                                                                           [exp]#(0,1) '

// la expresión
var charstring mystring := regexp(myinput, mypattern,1)
// devolverá el valor '17'.
```

C.18 Bitstring a charstring

```
bit2str (bitstring value) return charstring
```

Esta función convierte un solo valor **cadena de bits** en un solo valor **cadena de caracteres**. La **cadena de caracteres** resultante tiene la misma longitud que la **cadena de bits** y sólo contiene los caracteres '0' y '1'.

La **cadena de bits** se convierte en una **cadena de caracteres** convirtiendo cada bit de **bitstring** en un carácter '0' ó '1' a imagen del valor 0 ó 1 del bit. El orden consecutivo de los caracteres de la **charstring** resultante reproduce el orden de los bits de la **bitstring**.

EJEMPLO:

```
bit2str ('1110101'B) will return "1110101"
```

C.19 Hexstring a charstring

```
hex2str (hexstring value) return charstring
```

Esta función convierte un solo valor cadena hexadecimal en una sola cadena de caracteres. La cadena de caracteres resultante tiene la misma longitud que la cadena hexadecimal y sólo contiene los caracteres '0' a '9' y 'A' a 'F'.

La **cadena hexadecimal** se convierte en una **cadena de caracteres** convirtiendo cada dígito hexadecimal de la primera en un carácter '0' a '9' y 'A' a 'F' a imagen de los valores 0 a 9 y A a F de las cifras hexadecimales. El orden consecutivo de los caracteres de la **charstring** resultante reproduce el orden de los dígitos de la **hexstring**.

EJEMPLO:

```
hex2str ('AB801'H) will return "AB801"
```

C.20 Octetstring a character string

```
oct2str (octetstring invalue) return charstring
```

Esta función convierte un `invalue` **cadena de octetos** en una **cadena de caracteres** que representa la cadena equivalente del valor de entrada. La **cadena de caracteres** resultante ha de tener la misma longitud que la **cadena de octetos** entrante.

A estos efectos, se convierte cada dígito hex de `invalue` en un carácter '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E' o 'F', que replica el valor de dicho dígito. El orden consecutivo de los caracteres en la **charstring** resultante es igual al de los dígitos hex en la **octetstring**.

EJEMPLO:

```
oct2str ('4469707379'O) = "4469707379"
```

C.21 Character string a octetstring

```
str2oct (charstring invalue) return octetstring
```

Esta función convierte una cadena de caracteres del tipo **charstring** en una **cadena de octetos**. La cadena `invalue` contendrá un número par de caracteres, cada uno de los cuales podrá ser únicamente uno de los caracteres gráficos siguientes: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e' 'f', 'A', 'B', 'C', 'D', 'E' o 'F'. La **cadena de octetos** resultante tiene la misma longitud que la **cadena de caracteres** entrante.

EJEMPLO:

```
str2oct ("54696E6B792D57696E6B79") = '54696E6B792D57696E6B79'O
```

C.22 Bitstring a hexstring

```
bit2hex (bitstring value) return hexstring
```

Esta función convierte un solo valor **cadena de bits** en una sola **cadena hexadecimal**. La **cadena hexadecimal** resultante representa el mismo valor que la **bitstring**.

La cadena de bits se ha de convertir en una cadena hexadecimal dividiéndola en grupos de cuatro bits, empezando por el bit del extremo derecho. Cada grupo de cuatro bits se convierte en un dígito hexadecimal como se indica a continuación:

'0000'B → '0'H, '0001'B → '1'H, '0010'B → '2'H, '0011'B → '3'H, '0100'B → '4'H, '0101'B → '5'H,
'0110'B → '6'H, '0111'B → '7'H, '1000'B → '8'H, '1001'B → '9'H, '1010'B → 'A'H, '1011'B → 'B'H,
'1100'B → 'C'H, '1101'B → 'D'H, '1110'B → 'E'H, and '1111'B → 'F'H.

Si el grupo más a la izquierda tiene menos de cuatro bits, se rellena con '0'B a la izquierda para tener exactamente cuatro bits y convertirlo. El orden consecutivo de los caracteres hexadecimales de la cadena hexadecimal resultante reproduce el orden de los grupos de cuatro bits de la bitstring.

EJEMPLO:

```
bit2hex ('111010111'B)= '1D7'H
```

C.23 Hexstring a octetstring

```
hex2oct (hexstring value) return octetstring
```

Esta función convierte un solo valor de **cadena hexadecimal** en una sola **cadena de octetos**. La **cadena de octetos** resultante representa el mismo valor que la **hexstring**.

La **cadena hexadecimal** se ha de convertir en una **cadena de octetos** que contiene la misma secuencia de dígitos hexadecimales que la primera cuando el resultado de la operación longitud de **hexstring** modulo 2 es 0. Si no es así, la **cadena de octetos** resultante tiene 0 en la primera posición de dígito hexadecimal de la izquierda, seguido por la misma secuencia de dígitos hexadecimales de la **hexstring**.

EJEMPLO:

```
hex2oct ('1D7'H)= '01D7'O
```

C.24 Bitstring a octetstring

```
bit2oct (bitstring value) return octetstring
```

Esta función convierte un solo valor de **cadena de bits** en una sola **cadena de octetos**. La **cadena de octetos** resultante representa el mismo valor que la **bitstring**.

Principio de conversión: bit2oct(value)=hex2oct(bit2hex(value)).

EJEMPLO:

```
bit2oct ('111010111'B)= '01D7'O
```

C.25 Hexstring a bitstring

```
hex2bit (hexstring value) return bitstring
```

Esta función convierte un solo valor de **cadena hexadecimal** en una sola **cadena de bits**. La **cadena de bits** resultante representa el mismo valor que la **hexstring**.

La **cadena hexadecimal** se ha de convertir en una **cadena de bits** convirtiendo los dígitos hexadecimales de la primera en grupos de bits como se indica a continuación:

'0'H → '0000'B, '1'H → '0001'B, '2'H → '0010'B, '3'H → '0011'B, '4'H → '0100'B, '5'H → '0101'B,
'6'H → '0110'B, '7'H → '0111'B, '8'H → '1000'B, '9'H → '1001'B, 'A'H → '1010'B, 'B'H → '1011'B,
'C'H → '1100'B, 'D'H → '1101'B, 'E'H → '1110'B, and 'F'H → '1111'B.

El orden consecutivo de los grupos de cuatro bits de la **cadena de bits** resultante reproduce el orden de los dígitos hexadecimales de la **hexstring**.

EJEMPLO:

```
hex2bit ('1D7'H)= '000111010111'B
```

C.26 Octetstring a hexstring

```
oct2hex (octetstring value) return hexstring
```

Esta función convierte un solo valor de **cadena de octetos** en una sola **cadena hexadecimal**. La **cadena hexadecimal** resultante representa el mismo valor que la **octetstring**.

La **cadena de octetos** se ha de convertir en una **cadena hexadecimal** que contiene la misma secuencia de dígitos hexadecimales que la primera.

EJEMPLO:

```
oct2hex ('1D74'O)= '1D74'H
```

C.27 Octetstring a bitstring

```
oct2bit (octetstring value) return bitstring
```

Esta función convierte un solo valor de **cadena de octetos** en una sola **cadena de bits.** La **cadena de bits** resultante representa el mismo valor que la **octetstring**.

Principio de conversión: : oct2bit(value)=hex2bit(oct2hex(value)).

EJEMPLO:

```
oct2bit ('01D7'O)='0000000111010111'B
```

C.28 Integer a float

```
int2float (integer value) return float
```

Esta función convierte un valor **entero** en un valor de **coma flotante**.

EJEMPLO:

```
int2float(4) = 4.0
```

C.29 Float a integer

```
float2int (float value) return integer
```

Esta función convierte un valor de **coma flotante** en un valor **entero**; suprime la fracción del argumento y devuelve el **entero** resultante.

EJEMPLO:

```
float2int(3.12345E2) = float2int(312.345) = 312
```

C.30 La función rnd (generación de números aleatorios)

```
rnd ([float seed]) return float
```

La función **rnd** devuelve un número (seudo)aleatorio menor que 1 pero no menor que 0. El generador de números aleatorios se inicializa con un valor de arranque facultativo. Después, si no se indica otro valor, el último número generado será considerado como valor de arranque para el siguiente número aleatorio. Si no se ha inicializado, el valor de arranque para la primera utilización de **rnd** será un valor calculado por el sistema.

NOTA – La función **rnd** repetirá la misma secuencia de números aleatorios cada vez que se inicialice con el mismo valor de arranque.

Puede utilizarse la siguiente fórmula para producir una determinada gama de enteros aleatorios:

```
float2int(int2float(upperbound - lowerbound +1)*rnd()) + lowerbound
// En esta expresión, upperbound y lowerbound son los límites superior e
// inferior de la gama.
```

C.31 La función Substring (subcadena)

```
substr (any_string_type value, in integer index, in integer returncount) return
input_string_type
```

Esta función devuelve una subcadena a partir de un valor de tipo **bitstring**, **hexstring**, **octetstring** o cualquier cadena de caracteres. El tipo de esta subcadena será el tipo raíz del valor entrante. El segundo parámetro in (índice) determina el punto inicial de la subcadena que devuelve esta función. La indexación empieza en cero. El tercer parámetro de entrada define la longitud de la subcadena que devuelve esta función. Se utilizan las unidades de longitud del cuadro 4.

EJEMPLO:

```
substr ('00100110'B, 3, 4)       // devuelve '0011'B

substr ('ABCDEF'H, 2, 3)         // devuelve 'CDE'H

substr ('01AB23CD'O, 1, 2)       // devuelve 'AB23'O

substr ("My name is JJ", 11, 2)  // devuelve "JJ"
```

C.32 Número de elementos en un tipo estructurado

```
sizeoftype(any_type value) return integer
```

Esta función devuelve el número declarado de elementos de un parámetro de módulo, una constante, una variable o una plantilla de un tipo o arreglo **record of** o **set of** (véase la nota). Se debe aplicar esta función a valores de tipos con restricción de longitud. El número real que hay que devolver es el número secuencial del último elemento sin importar si su valor está definido o no (es decir, el índice de longitud superior de la definición de tipo en la cual se basa el parámetro de la función, más 1).

NOTA – Sólo se calculan elementos del objeto TTCN-3, que es el parámetro de la función; es decir, no se tienen en cuenta, al calcular el valor devuelto, elementos de tipos o valores anidados.

EJEMPLO:

```
// Siendo
type record of integer MyPDU1;
type set length(1..8) of integer MyPDU2;
type record length(10) of integer MyPDU3;

var MyPDU1 MyRecordOfVar1;
var MyPDU2 MyRecordOfVar2;
var MyPDU3 MyRecordOfVar3;

var integer numElements;

// entonces
numElements := sizeoftype(MyRecordOfVar1);   // devuelve error, pues MyPDU1 no tiene
                                             // restricción
numElements := sizeoftype(MyRecordOfVar2);   // devuelve 8
numElements := sizeoftype(MyRecordOfVar3);   // devuelve 10
```

C.33 Cadena de caracteres a float

```
str2float (charstring value) return float
```

Esta función convierte una **charstring** que contiene un número de punto flotante en un valor **float**. El formato el número en la **charstring** será conforme a las reglas de 6.1.0, con las siguientes excepciones:

- se permiten los ceros iniciales,
- se permite el signo '+' inicial, antes de valores positivos,
- se permite la expresión '-0.0'.

EJEMPLO:

```
str2float('12345.6') //  is the same as str2float('123.456E+02')
```

C.34 La función Reemplazar

```
replace (in any_string_type str, in integer ind, in integer len, in any_string_type repl)
return any_string_type
```

Esta función reemplaza la subcadena de valor **str**, en el índice **ind** de longitud **len**, por el valor de cadena **repl** y devuelve la cadena resultante. **str** no podrá ser modificada. Si **len** es **0**, se inserta la cadena **repl**. Si ind es **0**, se inserta **repl** al comienzo de **str**. Si **ind** es **lengthof(str)**, se inserta **repl** al final de **str**. **str** y **repl** deben ser cadenas del mismo tipo y tener como tipo base **bitstring**, **hexstring**, **octetstring**, o cualquier otra cadena de caracteres. La cadena devuelta es del mismo tipo que **str** y **repl**. Obsérvese que la indexación de cadenas empieza desde cero.

Los siguientes casos de error conducen a un error en el tiempo de compilación o de funcionamiento:

- **str** o **repl** no son de tipo cadena;
- **str** y **repl** son de diferente tipo;
- **ind** es menor que **0** o mayor que **lengthof(str)**;
- **len** es menor que **0** o mayor que **lengthof(str)**;
- **ind+len** es mayor que **lengthof(str)**.

EJEMPLO:

```
replace ('00000110'B, 1, 3, '111'B)          // devuelve '01110110'B

replace ('ABCDEF'H, 0, 2, '123'H)      // devuelve '123CDEF'H

replace ('01AB23CD'O, 2, 1, 'FF96'O)   // devuelve '01ABFF96CD'O

replace ("My name is JJ", 11, 1, "xx") // devuelve "My name is xxJ"

replace ("My name is JJ", 11, 0, "xx") // devuelve "My name is xxJJ"

replace ("My name is JJ", 2, 2, "x")   // devuelve "Myxame is JJ",

replace ("My name is JJ", 12, 2, "xx") // produce error de caso de prueba

replace ("My name is JJ", 13, 2, "xx") // produce error de caso de prueba

replace ("My name is JJ", 13, 0, "xx") // devuelve "My name is JJxx"
```

C.35 Octetstring a cadena de caracteres

```
oct2char (octetstring invalue) return charstring
```

Esta función convierte un invalue **octetstring** en una **charstring**. El parámetro de entrada invalue no contendrá valores de octeto mayores que 7F. La **charstring** resultante tendrá la misma longitud que la **octetstring** inicial. Los octetos se interpretan como códigos conforme a la Rec. UIT-T T.50 [9] (según el IRV) y los caracteres resultantes se agregan al valor devuelto.

EJEMPLO:

```
oct2char ('4469707379'O) = "Dipsy"
```

NOTA – Es posible que la cadena de caracteres devuelta contenga caracteres no gráficos, los cuales no pueden presentarse entre comillas dobles.

C.36 Cadena de caracteres a octetstring

```
char2oct (charstring invalue) return octetstring
```

Esta función convierte un `invalue` **charstring** en una **octetstring**. Cada octeto de la **octetstring** contendrá los códigos de la Rec.UIT-T T.50 [9] (conforme al IRV) de los caracteres de `invalue` adecuados.

EJEMPLO:

```
char2oct ("Tinky-Winky") = '54696E6B792D57696E6B79'O
```

Anexo D (informativo)

Vacío

NOTA – El contenido de este anexo se incluye en la Rec. UIT-T Z.146 [6].

Anexo E (informativo)

Biblioteca de tipos útiles

E.1 Limitaciones

Los nombres de los tipos que se incorporen en esta biblioteca han de ser únicos en todo el lenguaje y toda la biblioteca (es decir, no se podrá utilizar ninguno de los nombres definidos en el anexo C). Los usuarios de la notación TTCN-3 no deberían utilizar los nombres definidos en esta biblioteca como identificadores para definiciones diferentes de las de este anexo.

NOTA – Visto lo anterior, las definiciones de tipos de este anexo se pueden repetir en módulos TTCN-3, pero los identificadores de este anexo no se podrán utilizar para definir otros tipos que no se especifiquen en este anexo.

E.2 Tipos TTCN-3 útiles

E.2.1 Tipos básicos simples útiles

E.2.1.0 Enteros de un solo byte con o sin signo

Estos tipos soportan valores enteros entre –128 y 127 para tipos con signo, y entre 0 y 255 para tipos sin signo. Tienen la misma notación de valor que el tipo integer. Los valores de estos tipos se han de codificar y decodificar como si estuvieran representados en un solo byte dentro del sistema, sea cual sea la forma de representación efectivamente utilizada.

NOTA – Los valores de estos tipos se pueden codificar de la misma forma o de distintas formas, y también con una codificación diferente de la codificación del tipo integer (tipo raíz de estos tipos útiles), según las reglas de codificación utilizadas. No entra en el ámbito de esta Recomendación tratar las reglas de codificación.

Definiciones de estos tipos:

```
type integer    byte            (-128 .. 127)     with { variant "8 bit" };

type integer    unsignedbyte    (0 .. 255)        with { variant "unsigned 8 bit" };
```

E.2.1.1 Enteros cortos con o sin signo

Estos tipos soportan valores enteros entre –32 768 y 32 767 para tipos con signo, y entre 0 y 65 535 para tipos sin signo. Tienen la misma notación de valor que el tipo integer. Los valores de estos tipos se han de codificar y decodificar como si estuvieran representados en dos bytes dentro del sistema, sea cual sea la forma de representación efectivamente utilizada.

NOTA – Los valores de estos tipos se pueden codificar de la misma forma o de distintas formas, y también con una codificación diferente de la codificación del tipo integer (tipo raíz de estos tipos útiles), según las reglas de codificación utilizadas. No entra en el ámbito de esta Recomendación tratar las reglas de codificación.

Definiciones de estos tipos:

```
type integer    short           (-32768 .. 32767)  with { variant "16 bit" };

type integer    unsignedshort   (0 .. 65535)       with { variant "unsigned 16 bit" };
```

E.2.1.2 Enteros largos con o sin signo

Estos tipos soportan valores enteros entre –2 147 483 648 y 2 147 483 647 para tipos con signo, y entre 0 y 4 294 967 295 para tipos sin signo. Tienen la misma notación de valor que el tipo integer. Los valores de estos tipos se han de codificar y decodificar como si estuvieran representados en cuatro bytes dentro del sistema, sea cual sea la forma de representación efectivamente utilizada.

NOTA – Los valores de estos tipos se pueden codificar de la misma forma o de distintas formas, y también con una codificación diferente de la codificación del tipo integer (tipo raíz de estos tipos útiles), según las reglas de codificación utilizadas. No entra en el ámbito de esta Recomendación tratar las reglas de codificación.

Definiciones de estos tipos:

```
type integer    long (-2147483648 .. 2147483647)
                with { variant "32 bit" };

type integer    unsignedlong (0 .. 4294967295)
                with { variant "unsigned 32 bit" };
```

E.2.1.3 Enteros longlong con o sin signo

Estos tipos soportan valores enteros entre –9 223 372 036 854 775 808 y 9 223 372 036 854 775 807 para tipos con signo, y entre 0 y 18 446 744 073 709 551 615 para tipos sin signo. Tienen la misma notación de valor que el tipo integer. Los valores de estos tipos se han de codificar y decodificar como si estuvieran representados en ocho bytes dentro del sistema, sea cual sea la forma de representación efectivamente utilizada.

NOTA – Los valores de estos tipos se pueden codificar de la misma forma o de distintas formas, y también con una codificación diferente de la codificación del tipo integer (tipo raíz de estos tipos útiles), según las reglas de codificación utilizadas. No entra en el ámbito de esta Recomendación tratar las reglas de codificación.

Definiciones de estos tipos:

```
type integer    longlong (-9223372036854775808 .. 9223372036854775807)
                with { variant "64 bit" };

type integer    unsignedlonglong (0 .. 18446744073709551615)
                with { variant "unsigned 64 bit" };
```

E.2.1.4 Valores de coma flotante IEEE 754 (float)

Estos tipos soportan la norma ANSI/IEEE 754 (véase la Bibliografía) para la aritmética de coma flotante. El tipo IEEE 754 float soporta números de coma flotante de base 10, exponente 8, mantisa 23 y un bit de signo. El tipo IEEE 754 doble soporta números de coma flotante de base 10, exponente 11, mantisa 52 y un bit de signo. El tipo IEEE 754 `extfloat` soporta números de coma flotante de base 10, exponente mínimo 11, mantisa mínima 32 y un bit de signo. El tipo IEEE 754 `extdouble` soporta números de coma flotante de base 10, exponente mínimo 15, mantisa mínima 64 y un bit de signo.

Los valores de estos tipos se han de codificar y decodificar conforme a las definiciones de IEEE 754. Para estos tipos se utiliza la misma notación de valor del tipo float (base 10).

NOTA – Los detalles de codificación de este tipo dependen de las reglas de codificación utilizadas efectivamente. No entra en el ámbito de esta Recomendación definir las reglas de codificación.

Definiciones de estos tipos:

```
type      float IEEE754float      with { variant "IEEE754 float" };

type      float IEEE754double     with { variant "IEEE754 double" };

type      float IEEE754extfloat   with { variant "IEEE754 extended float" };

type      float IEEE754extdouble  with { variant "IEEE754 extended double" };
```

E.2.2 Tipos útiles de cadena de caracteres

E.2.2.0 Cadena de caracteres UTF-8 "utf8string"

Este tipo soporta todo el juego de caracteres del tipo TTCN-3 **`universal charstring`** (véase el párrafo d de 6.1.1). Los valores diferenciados son cero, uno o más caracteres de este juego. Los valores de este tipo se tienen que codificar y decodificar completamente (cada carácter del valor particularmente) conforme al formato de transformación UCS 8 (UTF-8) definido en el anexo R de ISO/CEI 10646 [10]. Para este valor se utiliza la misma notación de valor del tipo **`universal charstring`**.

Definición de este tipo:

```
type universal charstring utf8string with { variant "UTF-8" };
```

E.2.2.1 Cadena de caracteres BMP "bmpstring"

Este tipo soporta el juego de caracteres del plano multilingüe básico (BMP, *basic multilingual plane*) de ISO/CEI 10646 [10]. El BMP representa todos los caracteres del plano 00 del grupo 00 del juego de caracteres universal codificado en varios octetos. Los valores diferenciados son cero, uno o más caracteres del BMP. Los valores de este tipo se tienen que codificar y decodificar completamente (cada carácter del valor particularmente) conforme al formato de representación codificada UCS-2 (véase 14.1 de ISO/CEI 10646 [10]). Para este valor se utiliza la misma notación de valor del tipo **`universal charstring`**.

NOTA – El tipo "bmpstring" soporta un subconjunto del tipo TTCN-3 **`universal charstring`**.

Definición de este tipo:

```
type universal charstring bmpstring          ( char ( 0,0,0,0 ) .. char ( 0,0,255,255) )
                     with { variant "UCS-2" };
```

E.2.2.2 Cadena de caracteres UTF-16 "utf16string"

Este tipo soporta todos los caracteres de los planos 00 a 16 del grupo 00 del juego de caracteres universal codificado en varios octetos (véase ISO/CEI 10646 [10]). Los valores diferenciados son cero, uno o más caracteres de este conjunto. Los valores de este tipo se tienen que codificar y decodificar completamente (cada carácter del valor particularmente) conforme al formato de transformación UCS 16 (UTF-16) definido en el anexo Q de ISO/CEI 10646 [10]. Para este valor se utiliza la misma notación de valor del tipo **universal charstring**.

NOTA – El tipo "utf16string" soporta un subconjunto del tipo TTCN-3 **universal charstring**.

Definición de este tipo:

```
type universal charstring utf16string  ( char ( 0,0,0,0 ) .. char ( 0,16,255,255) )
                    with { variant "UTF-16" };
```

E.2.2.3 Cadena de caracteres ISO/CEI 8859 "iso8859string"

Este tipo soporta todos los caracteres de todos los alfabetos definidos en la norma múltiple ISO/CEI 8859 (véase la bibliografía). Los valores diferenciados son cero, uno o más caracteres del conjunto ISO/CEI 8859. Los valores de este tipo se tienen que codificar y decodificar completamente (cada carácter del valor particularmente) conforme a la representación codificada especificada en ISO/CEI 8859 (codificación de 8 bits). Para este valor se utiliza la misma notación de valor del tipo **universal charstring**.

NOTA 1 – El tipo "iso8859string" soporta un subconjunto del tipo TTCN-3 **universal charstring**.

NOTA 2 – Para cada alfabeto ISO/CEI 8859, la parte inferior de la tabla del juego de caracteres (posiciones 02/00 a 07/14) es compatible con el juego de caracteres de la Rec. UIT-T T.50 [9]. Por tanto, todos los caracteres específicos de otros idiomas se definen únicamente para la parte superior de la tabla (posiciones 10/00 a 15/15). Dado que el tipo "iso8859string" se considera como un subconjunto del tipo TTCN-3 universal charstring, todas las representaciones codificadas de los alfabetos ISO/CEI 8859 concuerdan con un carácter equivalente (que tiene la misma representación codificada cuando se utiliza el código de 8 bits) de las tablas de caracteres Latín Básico y Suplemento Latín 1 de ISO/CEI 10646 [10].

Definición de este tipo:

```
type universal charstring iso8859string      ( char ( 0,0,0,0 ) .. char ( 0,0,0,255) )
                    with { variant "8 bit" };
```

E.2.3 Tipos estructurados útiles

E.2.3.0 Literal decimal de coma fija

Este tipo soporta la utilización de un literal decimal de coma fija conforme a la definición de la sintaxis y la semántica del lenguaje IDL versión 2.6 (véase la bibliografía). Comprende un entero, la coma decimal y una fracción. Tanto el entero como la fracción están formados por una serie de dígitos decimales (base 10). El número de dígitos se registra en "digits" y el tamaño de la fracción se indica en "scale". Los dígitos propiamente dichos se registran en "value_". Para este tipo se utiliza la misma notación de valor del tipo record. Hay que codificar y decodificar los valores de este tipo como valores decimales IDL de coma fija.

NOTA – Los detalles de codificación de valores de este tipo dependen de las reglas de codificación utilizadas efectivamente. No entra en el ámbito de esta Recomendación definir las reglas de codificación.

Definición de este tipo:

```
type record     IDLfixed {
                      unsignedshort     digits,
                      short             scale,
                      charstring        value_
                }
                with { variant "IDL:fixed FORMAL/01-12-01 v.2.6" };
```

E.2.4 Tipos útiles de cadena atómica

E.2.4.1 Tipo de carácter IRV único

Tipo cuyos valores distinguidos son caracteres únicos de la versión de la Rec. UIT-T T.50 [9] que es conforme a la Versión internacional de referencia (IRV, *international reference version*), como se especifica en 8.2/T.50 [9] (véase también la nota 2 de 6.1.1).

La definición de tipo de este tipo es:

```
type charstring char length (1);
```

NOTA 1 – El nombre de este tipo útil es el mismo que el de la palabra clave TTCN-3 empleada para indicar valores **universal charstring** en la forma cuádruple. En general, no se permite utilizar palabras clave TTCN-3 como identificadores. El tipo útil "char" constituye la única excepción, y sólo se permite en aras de lograr la compatibilidad con versiones anteriores de la norma TTCN-3.

NOTA 2 – Se puede utilizar con este tipo la cadena especial "8 bit", definida en 28.2.3, para especificar una determinada codificación de sus valores. Asimismo, es posible modificar otras propiedades del tipo de base, utilizando mecanismos de atributo.

E.2.4.2 Tipo de carácter universal único

Tipo cuyos valores distinguidos son caracteres únicos ISO/CEI 10646 [10].

La definición de tipo de este tipo es:

```
type universal charstring uchar length (1);
```

NOTA – Con este tipo se pueden utilizar las cadenas especiales definidas en 28.2.3, salvo "8 bit", para especificar una determinada codificación de sus valores. Asimismo, es posible modificar otras propiedades del tipo de base, utilizando mecanismos de atributo.

E.2.4.3 Tipo de bit único

Tipo cuyos valores distinguidos son dígitos binarios únicos.

La definición de tipo de este tipo es:

```
type bitstring bit length (1);
```

E.2.4.4 Tipo hex único

Tipo cuyos valores distinguidos son dígitos hexadecimales únicos.

La definición de tipo de este tipo es:

```
type hexstring hex length (1);
```

E.2.4.5 Tipo octet único

Tipo cuyos valores distinguidos son pares de dígitos hexadecimales.

La definición de tipo de este tipo es:

```
type octetstring octet length (1);
```

Anexo F (informativo)

Operaciones sobre objetos activos TTCN-3

F.1 Generalidades

En este anexo se describe someramente la semántica de las operaciones sobre objetos activos TTCN-3 que son componentes de prueba, temporizadores y puertos. Este comportamiento dinámico se escribe en forma de máquinas de estado en las que:

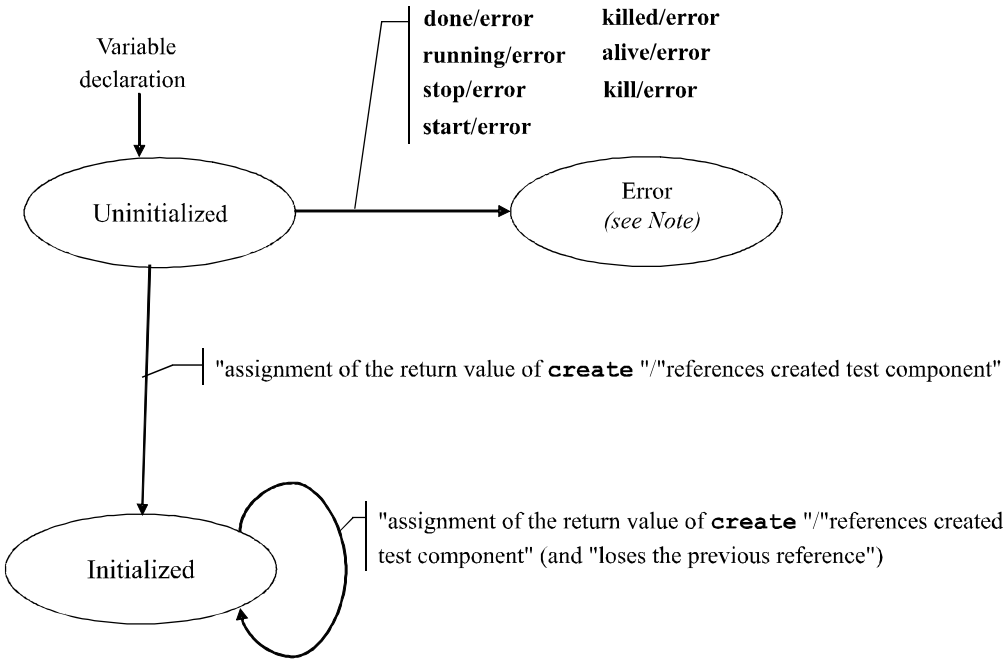
- los estados se denominan e identifican como nodos;
- el estado inicial se identifica mediante una flecha entrante;
- las transiciones entre estados que conectan dos estados (no necesariamente diferentes) se identifican mediante flechas;
- las transiciones que se señalan con la condición que habilita dicha transición (es decir, invocaciones de operación o de declaración), así como la condición resultante (por ejemplo, un error de caso de prueba), se separan ambas mediante '/':
  - las invocaciones de operación y declaración son las operaciones y declaraciones TTCN-3 que se aplican al objeto (en negritas);
  - un error, que sea la condición resultante, implica un error de caso de prueba (en negritas);
  - un null, que sea la condición resultante, implica que, salvo un posible cambio de estado, no se aplican otros resultados (en negritas);
  - match/no match indica el resultado de correspondencia de una transición (en negritas);
  - los valores concretos son resultados boolean o float (en cursivas);
  - todas las demás condiciones se describen textualmente (en tipo de letra estándar);
- se utilizan las notas para proporcionar más detalles sobre la máquina de estado.

Para más detalles, véase la semántica operacional de TTCN-3 [3]. De haber cualquier contradicción entre este anexo y dicha semántica operacional, esta última prevalece.

F.2 Componentes de prueba

F.2.1 Referencias de componentes de prueba

Las variables de tipos de componentes de prueba, las operaciones **`self`** y **`mtc`** se emplean para hacer referencia a componentes de prueba. Las operaciones **`start`**, **`stop`**, **`done`** y **`running`** no se aplican directamente a componentes de prueba sino a referencias de componente. El sistema de prueba tiene que decidir si la petición de operación ha de afectar el objeto de componente propiamente dicho, o si lo adecuado es ejecutar otra acción (por ejemplo, se presenta un error cuando se utiliza la referencia de un PTC interrumpido en una operación de inicio de componente). La operación **`create`**, que sirve para crear los PTC, devuelve una referencia única al PTC creado, que suele estar vinculado a un componente de prueba variable. En la figura F.1 se muestran los componentes de prueba variables.

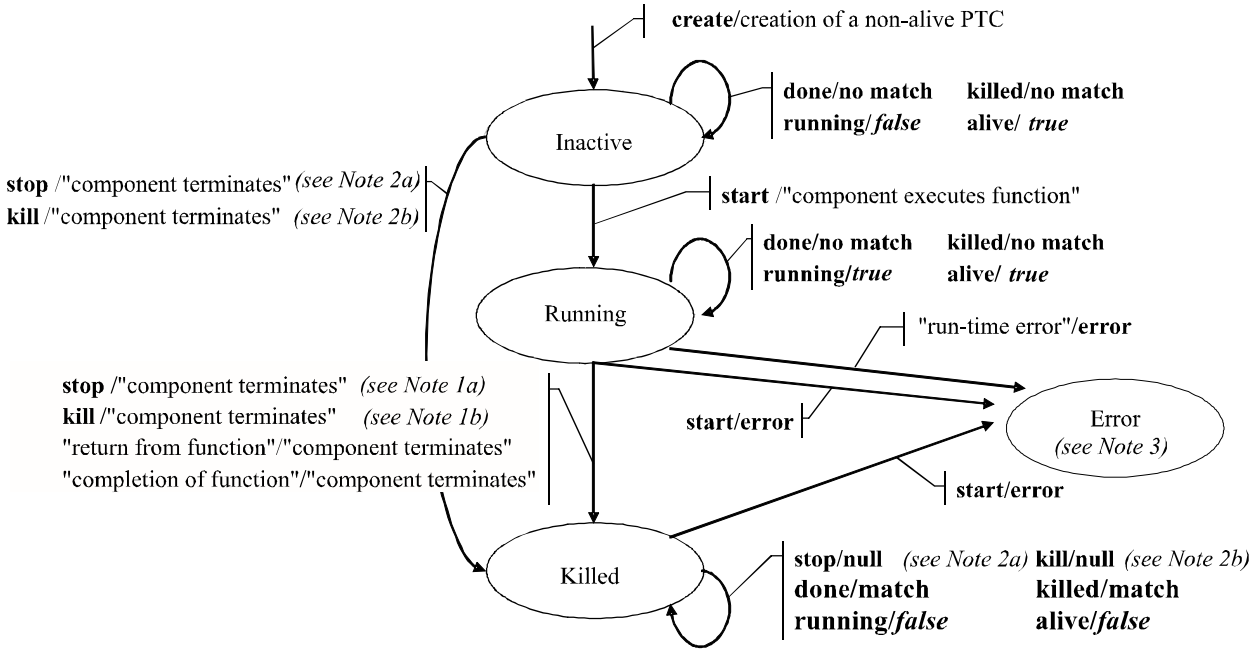


NOTA – Cuando un componente de prueba entra en estado de error, se atribuye el veredicto de error a su veredicto local, termina el caso de prueba y el resultado global del caso de prueba es error.

Figura F.1/Z.140 – Procesamiento de referencias de componentes de prueba

F.2.2 Comportamiento dinámico de los PTC

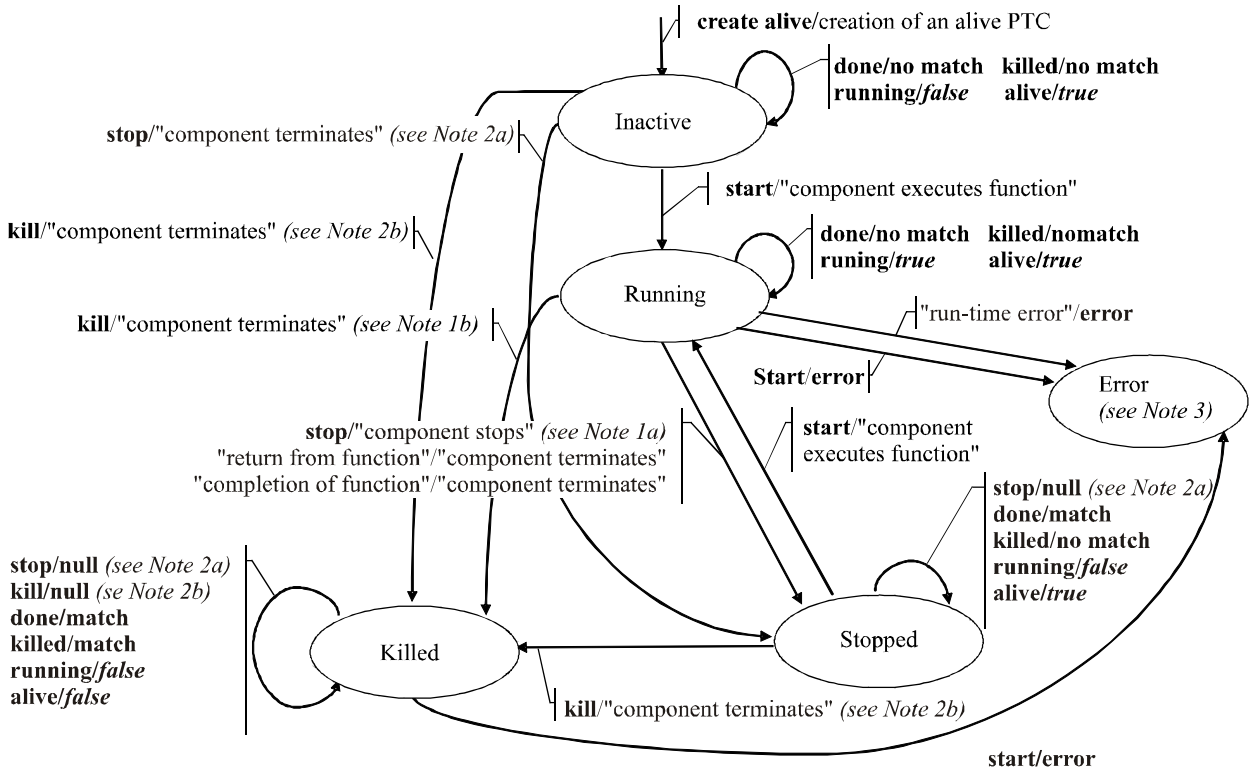
Los PTC pueden ser de tipo inactivo (non-alive) o activo (alive). Los PTC inactivos pueden estar en estados Inactive, Running y Killed. En la figura F.2 se muestra su comportamiento dinámico.



NOTA 1 – a) Stop puede ser un stop, un self.stop o un stop de otro componente de prueba;
b) Kill puede ser un kill, un self.kill, un kill de otro componente de prueba o un kill del sistema de prueba (en caso de error).
NOTA 2 – a) Stop sólo puede provenir de otro componente de prueba;
b) Kill sólo puede provenir de otro componente de prueba o del sistema de prueba (en caso de error).
NOTA 3 – Cuando un componente de prueba entra en estado de error, se atribuye el veredicto de error a su veredicto local, termina el caso de prueba y el resultado global del caso de prueba es error.

Figura F.2/Z.140 – Comportamiento dinámico de los PTC de tipo inactivo

Los PTC activos pueden estar en estados Inactive, Running, Stopped y Killed. En la figura F.3 se muestra su comportamiento dinámico.

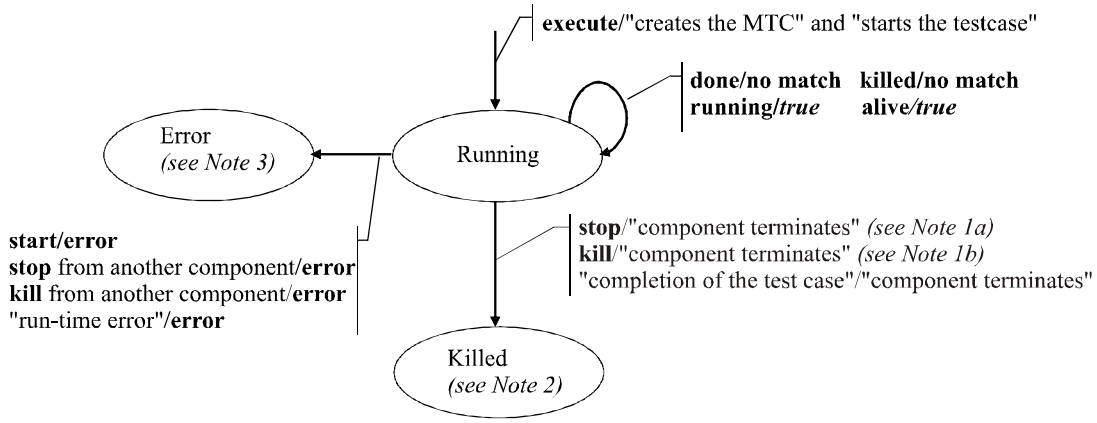


NOTA 1 – a) Stop puede ser un stop, un self.stop o un stop de otro componente de prueba;
b) Kill puede ser un kill, un self.kill, un kill de otro componente de prueba o un kill del sistema de prueba (en caso de error).
NOTA 2 – a) Stop sólo puede provenir de otro componente de prueba;
b) Kill sólo puede provenir de otro componente de prueba o del sistema de prueba (en caso de error).
NOTA 3 – Cuando un componente de prueba entra en estado de error, se atribuye el veredicto de error a su veredicto local, termina el caso de prueba y el resultado global del caso de prueba es error.

Figura F.3/Z.140 – Comportamiento dinámico de los PTC de tipo activo

F.2.3 Comportamiento dinámico del MTC

El MTC puede estar en el estado Running o en el Killed. En la figura F.4 se muestra su comportamiento dinámico.

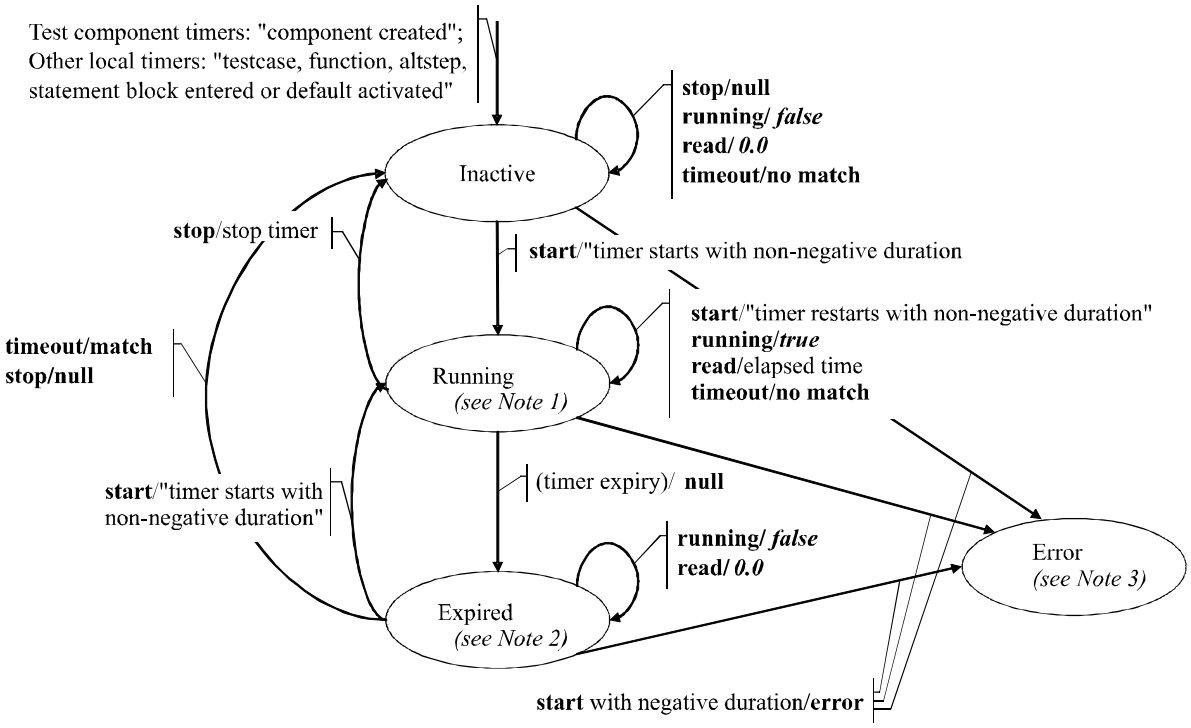


NOTA 1 – a) Stop puede ser un stop, un self.stop o un stop de otro componente de prueba;
b) Kill puede ser un kill, un self.kill, un kill de otro componente de prueba o un kill del sistema de prueba (en caso de error).
NOTA 2 – Todos los demás PTC también pueden estar en el estado killed y termina el caso de prueba.
NOTA 3 – Cuando un MTC entra en estado de error, se atribuye el veredicto de error a su veredicto local, termina el caso de prueba y el resultado global del caso de prueba es error.

Figura F.4/Z.140 – Comportamiento dinámico del MTC

F.3 Temporizadores

Los temporizadores pueden estar en uno de los estados: Inactive, Running o Expired. En la figura F.5 se muestra el comportamiento dinámico de un temporizador.



NOTA 1 – Dada una unidad de ámbito, todos los temporizadores de dicho ámbito que están en estado Running conforman la lista de temporizadores en funcionamiento (running-timer list).
NOTA 2 – Dada una unidad de ámbito, todos los temporizadores de dicho ámbito que están en estado Expired conforman la lista de temporizadores expirados (timeout-list).
NOTA 3 – Cuando un temporizador entra en estado de error, el componente de prueba al que pertenece también entra en estado de error, atribuye un veredicto local, termina el caso de prueba y el resultado global del caso de prueba es error.

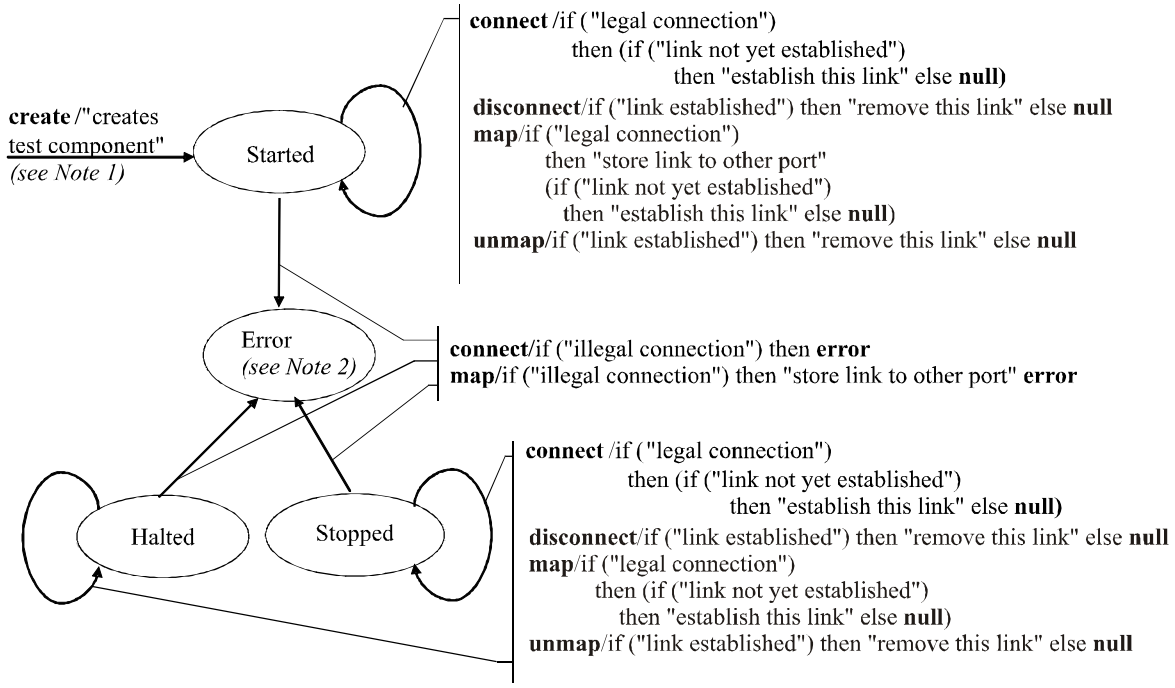
Figura F.5/Z.140 – Comportamiento dinámico de temporizadores

F.4 Puertos

Los puertos pueden estar en un estado Started o en uno Stopped. Puesto que su comportamiento es relativamente complejo, se ha dividido la máquina de estado en varias máquinas de estado, a saber una correspondiente al comportamiento dinámico de operaciones de configuración (es decir, connect, disconnect, map y unmap), una al de operaciones que controlan puertos (es decir, start, stop y clear) y otra al de operaciones de comunicación (es decir, send, receive, call, getcall, raise, catch, reply, getreply y check). Tratándose la operación trigger de una combinación de alt y receive, no se tiene en cuenta aquí.

F.4.1 Operaciones de configuración

Las operaciones de configuración de puertos (es decir, connect, disconnect, map y unmap) son independientes del estado del puerto. Su comportamiento se describe en la figura F.6.



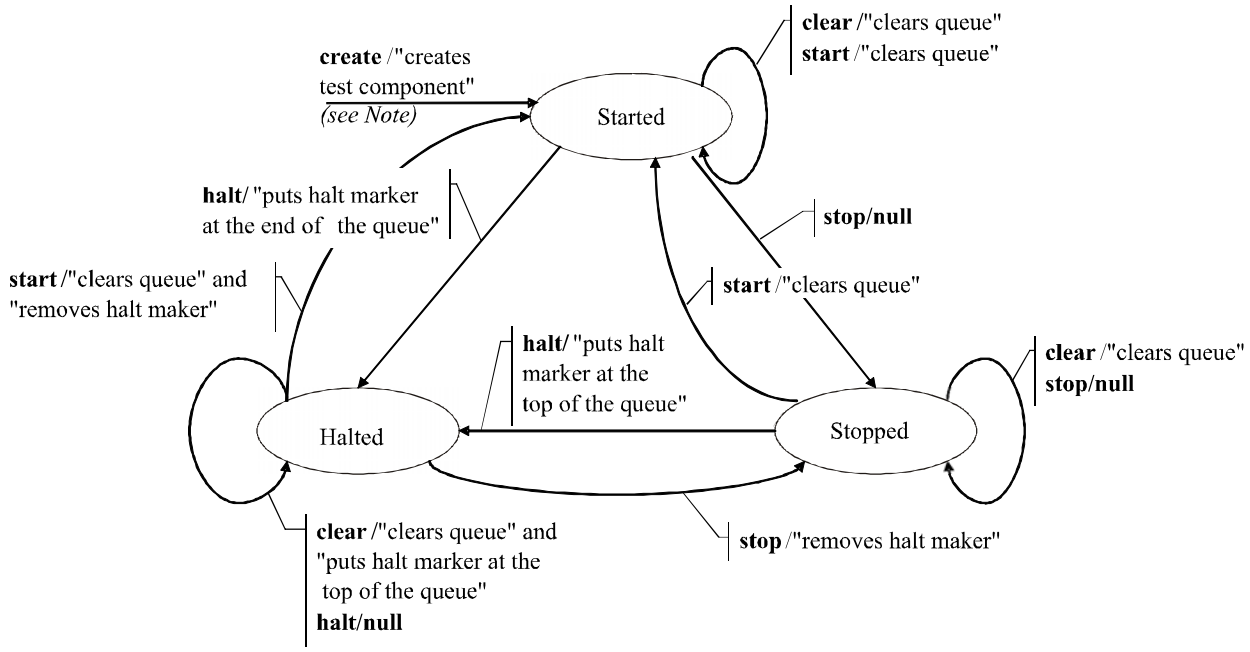
NOTA 1 – Al crear un PTC, se crean e inician también sus puertos; cuando se crea el MTC, se crean e inician los puertos del MTC y de la TSI.
NOTA 2 – Cuando un puerto entra en su estado de error, el componente de prueba al que pertenece también entra en estado de error, atribuye un veredicto local de error, termina el caso de pruebas y el resultado global del caso de pruebas es error.

Figura F.6/Z.140 – Comportamiento dinámico de puertos: Operaciones de configuración de puerto

Las transiciones no modifican el estado principal del puerto, es decir éste sigue siendo Started o Stopped.

F.4.2 Operaciones de control de puertos

En la figura F.7 se indican los resultados de las operaciones de control de puertos.

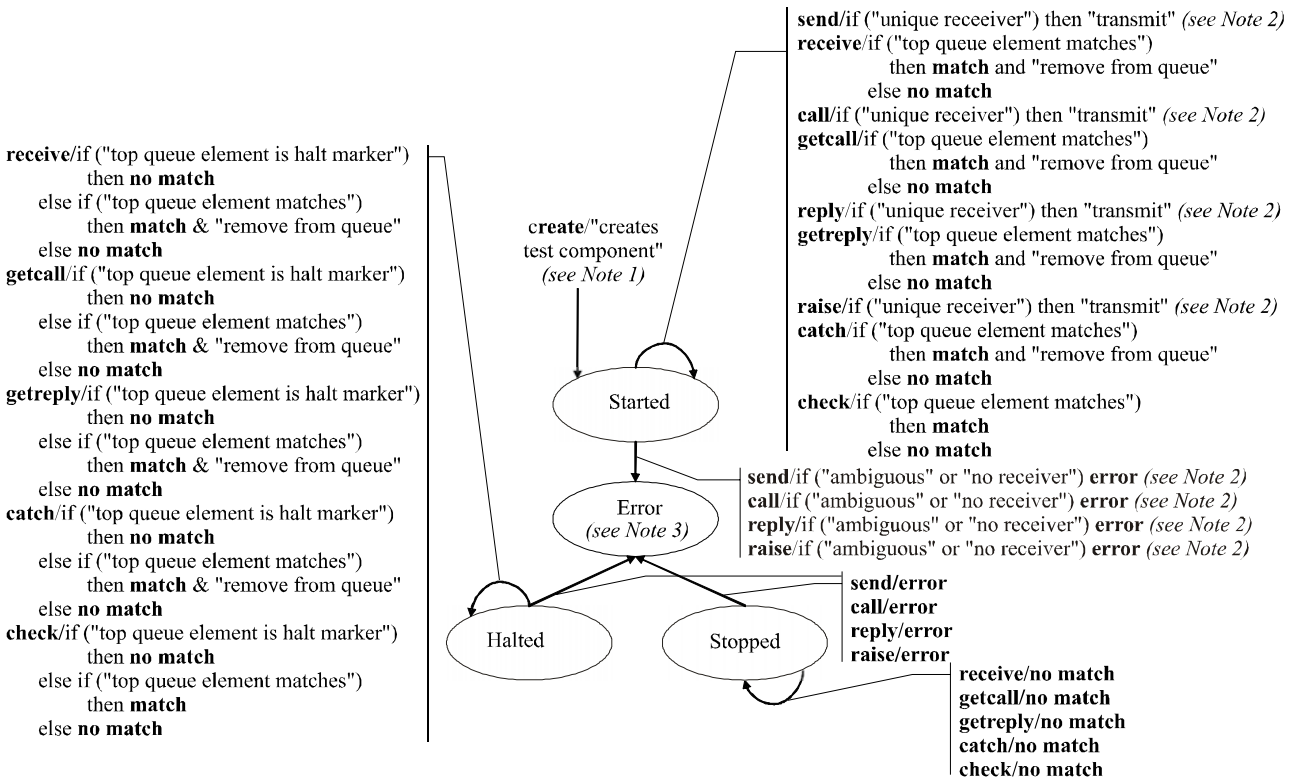


NOTA – Al crear un PTC, se crean e inician también sus puertos; cuando se crea el MTC, se crean e inician los puertos del MTC y de la TSI.

Figura F.7/Z.140 – Comportamiento dinámico de puertos: Operaciones de control de puertos

F.4.3 Operaciones de comunicación

En la figura F.8 se describen los resultados de las operaciones de comunicación send, receive, call, getcall, raise, catch, reply, getreply y check.



NOTA 1 – Al crear un PTC, se crean e inician también sus puertos; cuando se crea el MTC, se crean e inician los puertos del MTC y de la TSI.
NOTA 2 – Siempre que exista solamente un enlace para este puerto o la expresión "to address" se refiera a un componente cuyo puerto esté conectado a este puerto (un componente de prueba terminado no es un receptor válido), habrá un solo receptor.
NOTA 3 – Cuando un puerto entra en estado de error, el componente de prueba al que pertenece también entra en estado de error, se atribuye un veredicto local de error, termina el caso de pruebas y el resultado global del caso de pruebas es error.
NOTA 4 – Puesto que trigger es una combinación de alt y receive, no se trata aquí.

Figura F.8/Z.140 – Comportamiento dinámico de puertos: Operaciones de comunicación

Anexo G (informativo)

Características de lenguaje desaconsejadas

G.1 Definición de estilo de grupo de parámetros de módulo

En la versión anterior de esta Recomendación se requería el empleo de una sintaxis de tipo grupo, como se muestra en el ejemplo a continuación, para declarar los parámetros de módulo. Si bien en la presente versión se ha unificado la sintaxis de dichos parámetros de módulo con la de declaración de constantes y variables, la sintaxis de tipo grupo no se ha suprimido completamente, para dejar un período de tiempo durante el cual los fabricantes de equipos y los usuarios podrán pasar de la anterior a la nueva sintaxis. Se espera suprimir completamente la sintaxis de declaración de parámetro de módulo de tipo grupo en la próxima edición que se publique de esta Recomendación.

EJEMPLO (sintaxis superflua):

```
module MyModuleWithParameters
{
    modulepar { integer TS_Par0, TS_Par1 := 0;
                boolean TS_Par2 := true
              };
    modulepar { hexstring TS_Par3 };
}
```

G.2 Importación recursiva

En la anterior versión de esta Recomendación se permitía importar implícitamente definiciones nominadas, a través de la importación de otras definiciones del mismo módulo que se utilizan de una manera recursiva. En la presente versión de esta Recomendación se desaconseja dicha característica, que se espera suprimir completamente en la próxima versión.

G.3 Utilización de la palabra clave `all` en las definiciones de tipo de puerto

En la anterior versión de esta Recomendación se permitía emplear la palabra clave **`all`** en las definiciones de tipo de puerto, en lugar de una lista explícita de tipos y firmas permitida a través del puerto en cuestión. En la presente versión de esta Recomendación se desaconseja dicha característica, que se espera suprimir completamente en la próxima versión.

BIBLIOGRAFÍA

- ETSI ES 201 873-1 V1.1.2 (2001-06), *Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3; Part 1: TTCN-3 Core Language*.
- ETSI ES 201 873-1 V2.2.1 (2003-02), *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*.
- ISO/CEI 8859-1:1998, *Information technology – 8-bit single-byte coded graphic character sets – Part 1: Latin alphabet No. 1*.
- Object Management Group (OMG): *The Common Object Request Broker: Architecture and Specification*, Chapter 3 – IDL Syntax and Semantics. Version 2.6, FORMAL/01-12-01, diciembre de 2001.
- IEEE 754 (1985), *Binary Floating-Point Arithmetic*.

SERIES DE RECOMENDACIONES DEL UIT-T

Serie A	Organización del trabajo del UIT-T
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 multimedia
Serie I	Red digital de servicios integrados
Serie J	Redes de cable y transmisión de programas radiofónicos y televisivos, y de otras señales multimedia
Serie K	Protección contra las interferencias
Serie L	Construcción, instalación y protección de los cables y otros elementos de planta exterior
Serie M	Gestión de las telecomunicaciones, incluida la RGT y el mantenimiento de redes
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, comunicaciones de sistemas abiertos y seguridad
Serie Y	Infraestructura mundial de la información, aspectos del protocolo Internet y Redes de la próxima generación
Serie Z	**Lenguajes y aspectos generales de soporte lógico para sistemas de telecomunicación**

Impreso en Suiza
Ginebra, 2007