UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

UIT-T Z.140

(07/2001)

SECTEUR DE LA NORMALISATION
DES TÉLÉCOMMUNICATIONS
DE L'UIT

SÉRIE Z: LANGAGES ET ASPECTS GÉNÉRAUX LOGICIELS DES SYSTÈMES DE TÉLÉCOMMUNICATION

Techniques de description formelle

Notation combinée arborescente et tabulaire version 3 (TTCN-3): Langage noyau

Recommandation UIT-T Z.140

RECOMMANDATIONS UIT-T DE LA SÉRIE Z

LANGAGES ET ASPECTS GÉNÉRAUX LOGICIELS DES SYSTÈMES DE TÉLÉCOMMUNICATION

TECHNIQUES DE DESCRIPTION FORMELLE	
Langage de description et de spécification (SDL)	Z.100–Z.109
Application des techniques de description formelle	Z.110–Z.119
Diagrammes des séquences de messages	Z.120–Z.129
LANGAGES DE PROGRAMMATION	
CHILL: le langage de programmation de l'UIT-T	Z.200–Z.209
LANGAGE HOMME-MACHINE	
Principes généraux	Z.300–Z.309
Syntaxe de base et procédures de dialogue	Z.310–Z.319
LHM étendu pour terminaux à écrans de visualisation	Z.320–Z.329
Spécification de l'interface homme-machine	Z.330–Z.399
QUALITÉ DES LOGICIELS DE TÉLÉCOMMUNICATION	Z.400–Z.499
MÉTHODES DE VALIDATION ET D'ESSAI	Z.500–Z.599

Pour plus de détails, voir la Liste des Recommandations de l'UIT-T.

Recommandation UIT-T Z.140

Notation combinée arborescente et tabulaire version 3 (TTCN-3): Langage noyau

Résumé

La présente Recommandation définit le langage noyau de la version 3 de la notation combinée arborescente et tabulaire (TTCN-3). Celle-ci peut servir à spécifier tous les types de tests réactifs du système à divers accès de communication. Les domaines d'application typiques sont les tests de protocole (y compris les services mobiles et l'Internet), les essais de services (y compris les services complémentaires), les tests de module, les essais de plates-formes en architecture commune de courtage pour les requêtes sur des objets (CORBA), les interfaces de programme d'application (API), etc. La notation TTCN-3 n'est pas limitée aux texts de conformité car elle peut être utilisée pour de nombreuses autres sortes d'essais, portant par exemple sur l'interopérabilité, la robustesse, la régression, les systèmes et l'intégration. La spécification de suites de tests pour les protocoles de couche Physique est hors du domaine d'application de la présente Recommandation.

Source

La Recommandation Z.140 de l'UIT-T, élaborée par la Commission d'études 10 (2001-2004) de l'UIT-T, a été approuvée le 22 juillet 2001 selon la procédure définie dans la Résolution 1 de l'AMNT.

AVANT-PROPOS

L'UIT (Union internationale des télécommunications) est une institution spécialisée des Nations Unies dans le domaine des télécommunications. L'UIT-T (Secteur de la normalisation des télécommunications) est un organe permanent de l'UIT. Il est chargé de l'étude des questions techniques, d'exploitation et de tarification, et émet à ce sujet des Recommandations en vue de la normalisation des télécommunications à l'échelle mondiale.

L'Assemblée mondiale de normalisation des télécommunications (AMNT), qui se réunit tous les quatre ans, détermine les thèmes d'étude à traiter par les Commissions d'études de l'UIT-T, lesquelles élaborent en retour des Recommandations sur ces thèmes.

L'approbation des Recommandations par les Membres de l'UIT-T s'effectue selon la procédure définie dans la Résolution 1 de l'AMNT.

Dans certains secteurs des technologies de l'information qui correspondent à la sphère de compétence de l'UIT-T, les normes nécessaires se préparent en collaboration avec l'ISO et la CEI.

NOTE

Dans la présente Recommandation, l'expression "Administration" est utilisée pour désigner de façon abrégée aussi bien une administration de télécommunications qu'une exploitation reconnue.

DROITS DE PROPRIÉTÉ INTELLECTUELLE

L'UIT attire l'attention sur la possibilité que l'application ou la mise en œuvre de la présente Recommandation puisse donner lieu à l'utilisation d'un droit de propriété intellectuelle. L'UIT ne prend pas position en ce qui concerne l'existence, la validité ou l'applicabilité des droits de propriété intellectuelle, qu'ils soient revendiqués par un Membre de l'UIT ou par une tierce partie étrangère à la procédure d'élaboration des Recommandations.

A la date d'approbation de la présente Recommandation, l'UIT n'avait pas été avisée de l'existence d'une propriété intellectuelle protégée par des brevets à acquérir pour mettre en œuvre la présente Recommandation. Toutefois, comme il ne s'agit peut-être pas de renseignements les plus récents, il est vivement recommandé aux responsables de la mise en œuvre de consulter la base de données des brevets du TSB.

TABLE DES MATIÈRES

Recommandation UIT-T Z.140

Notation combinée arborescente et tabulaire version 3 (TTCN-3): Langage noyau

1 Domaine d'application

La présente Recommandation définit le langage noyau de la version 3 de la notation TTCN (ou TTCN-3). Celle-ci peut servir à spécifier tous les types de tests réactifs de système à divers accès de communication. Les domaines d'application typiques sont les tests de protocole (y compris les services mobiles et l'Internet), les essais de services (y compris les services complémentaires), les tests de module, les essais de plates-formes en architecture CORBA, les interfaces API, etc. La notation TTCN-3 n'est pas limitée aux tests de conformité car elle peut être utilisée pour de nombreuses autres sortes d'essais, portant par exemple sur l'interopérabilité, la robustesse, la régression, les systèmes et l'intégration. La spécification de suites de tests pour les protocoles de couche Physique est hors du domaine d'application de la présente Recommandation.

La notation TTCN-3 est destinée à être utilisée pour la spécification de suites de tests qui sont indépendantes des méthodes d'essai, des couches et des protocoles. Divers formats de présentation sont définis pour la notation TTCN-3, comme un format de présentation tabulaire [1] et un format de présentation graphique [2]. La spécification de ces formats est hors du domaine d'application de la présente Recommandation.

La présente Recommandation définit un moyen normalisé d'utiliser, avec la notation TTCN-3, la notation ASN.1 définie dans la série des Recommandations UIT-T X.680 [7], [8], [9] et [10]. L'harmonisation d'autres langages avec la notation TTCN-3 est hors du domaine d'application de la présente Recommandation.

Bien que la conception de la notation TTCN-3 ait pris en considération l'implémentation ultérieure de convertisseurs et de compilateurs TTCN-3, le moyen de réaliser des suites de tests exécutables (ETS, *realization of executable test suites*) à partir de suites de tests abstraits (ATS, *abstract test suites*) est hors du domaine d'application de la présente Recommandation.

2 Références

Les Recommandations UIT-T et autres références suivantes contiennent des dispositions qui, par suite de la référence qui y est faite, constituent des dispositions valables pour la présente Recommandation.

Les références sont soit spécifiques (identifiées par leur date de publication et/ou d'édition ou par leur numéro de version) soit non spécifiques.

– Pour une référence spécifique, les révisions ultérieures ne sont pas applicables.

– Pour une référence non spécifique, la plus récente version est applicable.

[1] Recommandation UIT-T Z.141 (2001), *La notation combinée arborescente et tabulaire version 3 (TTCN-3): format de présentation tabulaire*.

[2] Recommandation UIT-T Z.142 (Projet), *La notation combinée arborescente et tabulaire version 3 (TTCN-3): format de présentation graphique*.

[3] Recommandation UIT-T X.290 (1995), *Cadre général et méthodologie des tests de conformité d'interconnexion des systèmes ouverts pour les Recommandation sur les protocoles pour les applications de l'UIT-T – Concepts généraux*.

ISO/CEI 9646-1:1994, *Technologies de l'information – Interconnexion de systèmes ouverts (OSI) – Cadre général et méthodologie des tests de conformité – Partie 1: Concepts généraux*.

[4] Recommandation UIT-T X.292 (1998), *Cadre et méthodologie des tests de conformité OSI pour les Recommandations sur les protocoles pour les applications de l'UIT-T – Notation combinée arborescente et tabulaire (TTCN)*.

ISO/CEI 9646-3:1998, *Technologies de l'information – Interconnexion de systèmes ouverts – Essais de conformité – Méthodologie générale et procédures – Partie 3: Notation combinée, arborescente et tabulaire (TTCN)*.

[5] Recommandation UIT-T T.50 (1992), *Alphabet international de référence (ancien alphabet international n° 5 ou AI5) – Technologies de l'information – Jeux de caractères codés à 7 bits pour l'échange d'informations*.

ISO/CEI 646:1991, *Technologies de l'information – Jeu ISO de caractères codés à 7 éléments pour l'échange d'informations*.

[6] ISO/CEI 10646-1:1993, *Technologies de l'information – Jeu universel de caractères codés sur plusieurs octets (JUC) – Partie 1: Architecture et plan multilingue de base*.

[7] Recommandation UIT-T X.680 (1997) | ISO/CEI 8824-1:1998, *Technologies de l'information – Notation de syntaxe abstraite numéro un: spécification de la notation de base*.

[8] Recommandation UIT-T X.681 (1997) | ISO/CEI 8824-2:1998, *Technologies de l'information – Notation de syntaxe abstraite numéro un: spécification des objets informationnels*.

[9] Recommandation UIT-T X.682 (1997) | ISO/CEI 8824-3:1998, *Technologies de l'information – Notation de syntaxe abstraite numéro un: spécification des contraintes*.

[10] Recommandation UIT-T X.683 (1997) | ISO/CEI 8824-4:1998, *Technologies de l'information – Notation de syntaxe abstraite numéro un: paramétrage des spécifications de la notation de syntaxe abstraite numéro un*.

[11] Recommandation UIT-T X.690 (1997) | ISO/CEI 8825-1:1998, *Technologies de l'information – Règles de codage ASN.1: spécification des règles de codage de base, des règles de codage canoniques et des règles de codage distinctives*.

[12] Recommandation UIT-T X.691 (1997) | ISO/CEI 8825-2:1998, *Technologies de l'information – Règles de codage ASN.1: spécification des règles de codage compact*.

3 Définitions et abréviations

3.1 Définitions

La présente Recommandation définit les termes suivants:

3.1.1 type compatible: la notation TTCN-3 n'est pas fortement typée mais ce langage nécessite bien la compatibilité des types. Les variables, les constantes, les modèles, etc. ont des types compatibles s'ils se ramènent au même type de base et, dans le cas d'affectations, d'appariements, etc., aucun sous-typage (par exemple, en étendues, en restrictions de longueur) n'est violé.

3.1.2 accès de communication: mécanisme abstrait qui facilite la communication entre des composants de test. Un accès de communication est modélisé par une file d'attente directe dans le sens réception. Les accès peuvent être à base de messages, à base de procédures, ou à base d'un mélange des deux.

3.1.3 exception: en cas de communication synchrone, une exception est déclenchée (si elle est définie) par une entité de réponse qui ne peut pas envoyer la réponse normalement prévue à un appel de procédure distante.

3.1.4 suite de tests: module de notation TTCN-3 qui spécifie complètement, soit explicitement soit implicitement au moyen d'instructions d'importation, toutes les définitions et toutes les descriptions comportementales nécessaires pour définir un ensemble complet de tests élémentaires.

3.1.5 interface de système de test: composant de test qui assure le mappage des accès disponibles dans le système de test abstrait avec les accès offerts par un système de test réel.

3.1.6 paramétrisation de type: capacité d'introduire un type dans un objet paramétré sous forme de paramètre réel. Ce paramètre de type réel complète alors la spécification de type de cet objet. Noter que le paramètre n'est pas une valeur de type mais constitue le type proprement dit.

3.2 Définitions extraites de la Rec. UIT-T X.290 et l'ISO/CEI 9646-1

Pour les besoins de la présente Recommandation, les définitions suivantes, figurant dans la Rec. UIT-T X.290 et l'ISO/CEI 9646-1 [3] s'appliquent:

– déclaration de conformité d'implémentation (ICS, *implementation conformance statement*);
– informations supplémentaires sur l'implémentation destinées au test (IXIT, *implementation extra information for testing*);
– implémentation sous test (IUT, *implementation under test*);
– système sous test (SUT, *system under test*);
– test élémentaire;
– erreur de test élémentaire;
– système de test.

3.3 Définitions extraites de la Rec. UIT-T X.292 et l'ISO/CEI 9646-3

Pour les besoins de la présente Recommandation, les définitions suivantes, figurant dans la Rec. UIT-T X.292 et l'ISO/CEI 9646-3 [4] s'appliquent:

– composant de test principal (MTC, *main test component)*;
– composant de test parallèle (PTC, *parallel test component)*;
– sémantique d'analyse sélective.

3.4 Abréviations

La présente Recommandation utilise les abréviations suivantes:

API interface de programmation d'application (*application programming interface*)

ASN.1 notation de syntaxe abstraite numéro un (*abstract syntax notation one*)

ASP primitive de service abstraite (*abstract service primitive*)

ATS suite de tests abstraits (*abstract test suite*)

BNF formalisme de Backus-Naur (*Backus-Naur form*)

CORBA architecture du courtier commun de requête sur des objets (*common object request broker architecture*)

ETS suite de tests exécutables (*executable test suite*)

FIFO premier entré, premier sorti (*first in, first out*)

IDL langage de définition d'interface (*interface definition language*)

IUT implémentation sous test (*implementation under test*)

MTC composant de test principal (*master test component*)

PDU unité de données protocolaire (*protocol data unit*)

PICS déclaration de conformité d'implémentation de protocole (*protocol implementation conformance statement*)

PIXIT informations supplémentaires sur l'implémentation de protocole destinées au test (*protocol implementation extra information for testing*)

PTC composant de test parallèle (*parallel test component*)

SUT système sous test (*system under test*)

TTCN notation combinée arborescente et tabulaire (*tree and tabular combined notation*)

4 Introduction

La notation TTCN-3 est un langage flexible et puissant qui peut servir à spécifier tous les types de tests réactifs de système à diverses interfaces de communication. Les domaines d'application typiques sont les tests de protocole (y compris les services mobiles et l'Internet), les essais de services (y compris les services complémentaires), les tests de module, les essais de plates-formes en architecture CORBA, les tests d'interfaces API, etc. La notation TTCN-3 n'est pas limitée aux tests de conformité car elle peut être utilisée pour de nombreuses autres sortes d'essais, portant par exemple sur l'interopérabilité, la robustesse, la régression, les systèmes et l'intégration.

Du point de vue syntaxique, la notation TTCN-3 est très différente des versions antérieures de ce langage, tel que défini dans la Rec. UIT-T X.292 et l'ISO/CEI 9646-3 [4]. Une grande partie de la fonctionnalité de base de la notation TTCN a cependant été conservée et améliorée dans certains cas. La notation TTCN-3 offre les caractéristiques essentielles ci-après:

- capacité de spécifier des configurations dynamiques d'essais concurrents;
- exploitation de communications synchrones et asynchrones;
- capacité de spécifier des informations de codage et d'autres attributs (y compris l'extensibilité d'utilisation);
- capacité de spécifier les modèles de données et de signature avec des mécanismes d'appariement puissants;
- paramétrisation des types et des valeurs;
- affectation et traitement des verdicts de test;
- paramétrisation des suites de tests et mécanismes de sélection des tests élémentaires;
- utilisation combinée de la notation TTCN-3 avec la notation ASN.1 (et utilisation possible avec d'autres langages comme l'IDL);
- syntaxe bien définie, format d'échange et sémantique statique;
- différents formats de présentation (par exemple, présentation tabulaire et graphique);
- algorithme d'exécution précis (sémantique opérationnelle).

4.1 Le langage noyau et les formats de présentation

Historiquement, la notation TTCN a toujours été associée aux essais de conformité. Afin d'ouvrir ce langage à une plus large étendue d'applications d'essais aussi bien dans le domaine des normes que dans celui de l'industrie, la présente Recommandation subdivise la spécification de la notation TTCN-3 en plusieurs parties. La première, définie dans la présente Recommandation, est le langage noyau. La deuxième, définie dans la Rec. UIT-T Z.141 [1], est le format de présentation tabulaire, d'apparence et de fonctionnalité similaires aux versions antérieures de la notation TTCN.

La troisième partie, définie dans le projet de Rec. UIT-T Z.142 [2] est le format de présentation graphique.

Le langage noyau vise trois objectifs:

a) constituer un langage de test généralisé en mode alphanumérique à part entière;

b) constituer un format d'échange normalisé de suites de tests TTCN entre outils TTCN;

c) constituer la base sémantique (et, le cas échéant, la base syntaxique) de divers formats de présentation.

Le langage noyau peut être utilisé indépendamment des formats de présentation. Ni le format tabulaire ni le format graphique ne peuvent cependant être utilisés sans le langage noyau. L'utilisation et l'implémentation de ces formats de présentation doivent être assurées sur la base du langage noyau.

Les formats tabulaire et graphique sont les premiers d'une série de différents formats de présentation envisagés. Ces autres formats de présentation pourront être normalisés ou non normalisés, selon les définitions données par les utilisateurs de la notation TTCN-3 eux-mêmes. Ces formats additionnels ne sont pas définis dans la présente Recommandation.

La notation TTCN-3 est pleinement harmonisée avec la notation ASN.1, qui peut aussi être utilisée avec des modules TTCN-3 sous la forme d'une syntaxe de types et de valeurs de données en variante. L'utilisation de la notation ASN.1 dans les modules TTCN-3 est définie dans l'Annexe E. La méthode employée pour combiner les notations ASN.1 et TTCN-3 pourra être appliquée de façon à faciliter l'utilisation, avec la notation TTCN-3, d'autres systèmes à types et valeurs. Les détails n'en sont cependant pas définis dans la présente Recommandation.

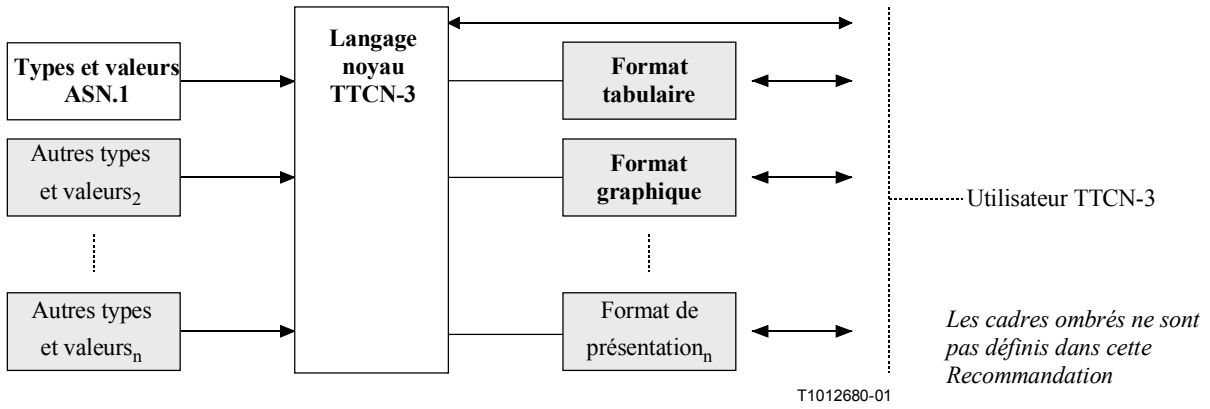


Figure 1/Z.140 – Vue d'utilisateur du langage noyau et des divers formats de présentation

Le langage noyau est défini par une syntaxe complète (voir Annexe A) et par une sémantique opérationnelle (voir Annexe B). Il contient une sémantique statique minimale (reproduite dans le corps de la présente Recommandation et dans son Annexe A) qui ne limite pas l'emploi du langage en raison d'un domaine d'application sous-jacent ou d'une méthode d'application sous-jacente. La fonctionnalité des versions antérieures de la notation TTCN, comme les indices de suite de tests que l'on peut réaliser au moyen d'outils non normalisés, ne fait pas partie de la notation TTCN-3.

5 Eléments linguistiques de base

L'unité de niveau supérieur de la notation TTCN-3 est un module. Celui-ci ne peut pas être structuré en sous-modules. Un module peut importer des définitions d'autres modules. Ceux-ci peuvent contenir des listes de paramètres de façon à offrir une sorte de paramétrisation de suite de tests qui est similaire aux mécanismes de paramétrisation PICS et PIXIT de la notation TTCN-2.

Un module se compose d'une partie définitions et d'une partie commande. La partie définitions d'un module définit les composants de test, les accès de communication, les types de données, les constantes, les modèles de données de test, les fonctions, les signatures pour les appels de procédure aux accès, les tests élémentaires, etc.

La partie commande d'un module appelle les tests élémentaires et en contrôle l'exécution. Elle peut également déclarer des variables (locales), etc. Des instructions de programmation (comme **`if-else`** et **`do-while`**) peuvent servir à spécifier l'ordre de sélection et d'exécution de tests élémentaires individuels. Le concept de variable globale n'est pas pris en charge en notation TTCN-3.

La notation TTCN-3 contient un certain nombre de types prédéfinis de données de base ainsi que des types structurés comme des enregistrements, des ensembles, des réunions, des types énumérés et des séquences tabulaires. Les types et valeurs ASN.1 peuvent, le cas échéant, être utilisés en notation TTCN-3 par importation.

Une sorte spéciale de valeur de donnée, appelée *modèle*, offre des mécanismes de paramétrisation et d'appariement permettant de spécifier des données de test à envoyer ou à recevoir par les accès de test. Les opérations effectuées à ces accès fournissent des capacités de communication aussi bien asynchrones que synchrones. Des appels de procédure peuvent être utilisés afin de tester des réalisations autres qu'en mode message.

Le comportement dynamique de test est exprimé par des tests élémentaires. Les instructions de programmation TTCN-3 comprennent de puissants mécanismes de description comportementale comme la réception en variante d'événements de communication et de temporisation, l'entrelacement et le comportement par défaut. Les mécanismes d'affectation de verdict de test et d'enregistrement chronologique sont également pris en charge.

Finalement, la plupart des éléments linguistiques de la notation TTCN-3 peuvent être affectés d'attributs comme ceux d'information de codage et ceux d'affichage. Il est également possible de spécifier des attributs définis par l'utilisateur (non normalisés).

Tableau 1/Z.140 – Aperçu général des éléments linguistiques de la notation TTCN-3

Elément linguistique	Mot clé associé	Spécifié dans les définitions de module	Spécifié dans la commande de module	Spécifié dans les fonctions/tests élémentaires
Définition de module TTCN-3	**`module`**			
Importation de définitions d'un autre module	**`import`**	Oui		
Groupement de définitions	**`group`**	Oui		
Définitions de type de donnée	**`type`**	Oui		
Définitions d'accès de communication	**`port`**	Oui		
Définitions de composant de test	**`component`**	Oui		
Définitions de signature	**`signature`**	Oui		

Tableau 1/Z.140 – Aperçu général des éléments linguistiques de la notation TTCN-3

Elément linguistique	Mot clé associé	Spécifié dans les définitions de module	Spécifié dans la commande de module	Spécifié dans les fonctions/tests élémentaires
Définitions de fonction/constante externe	**external**	Oui		
Définitions de constante	**const**	Oui	Oui	Oui
Définitions de modèle de données/signature	**template**	Oui		
Définitions de fonction	**function**	Oui		
Définitions de variante nommée	**named alt**	Oui		
Définitions de test élémentaire	**testcase**	Oui		
Déclarations de variable	**var**		Oui	Oui
Déclaration de temporisateur	**timer**		Oui	Oui

5.1 Définitions, instances et déclarations

Dans la présente Recommandation, le terme *déclaration* est utilisé de manière générale afin de désigner la création d'une définition statique ou d'une sorte d'instanciation dynamique où un nom est donné à un objet TTCN-3. Par exemple, les types et les constantes sont définis et une instruction comme l'appel d'une fonction ou la déclaration d'une variable est une instanciation. Dans les deux cas ces actions peuvent être considérées comme l'énonciation d'une déclaration.

5.2 Classement des éléments linguistiques

De manière générale, l'ordre dans lequel les déclarations peuvent être faites et l'association des déclarations avec des instructions de programmation sont arbitraires. Cependant, à l'intérieur d'un bloc d'instructions, comme une branche d'instruction **if-else**, toutes les (éventuelles) déclarations ne doivent être faites qu'au début du bloc d'instructions.

Exemple:

```
// Voici une association correcte de déclarations TTCN-3
:
var MyVarType MyVar2 := 3;
const integer MyConst:= 1;
if (x > 10)
{
     var integer MyVar1:= 1;
     :
     MyVar1:= MyVar1 + 10;
     :
}

     :
```

5.2.1 Références anticipées

Les définitions contenues dans la partie définitions d'un module peuvent être énoncées dans un ordre quelconque. Bien qu'il soit recommandé (pour des raisons de lisibilité) d'éviter les références anticipées, cela n'est pas obligatoire. Par exemple, des éléments récurrents comme des fonctions qui appellent d'autres fonctions et d'autres paramétrisations de module peuvent rendre inévitable d'énoncer des références anticipées.

Celles-ci ne sont autorisées que pour des déclarations dans la partie définitions du module. Les références anticipées ne doivent jamais être insérées dans la partie commande du module, dans les définitions de test élémentaire, dans les fonctions et dans les variantes nommées. Autrement dit, il ne doit jamais y avoir de références anticipées à des variables locales, à des temporisateurs locaux et à des constantes locales.

5.3 Paramétrisation

La notation TTCN-3 prend en charge aussi bien la paramétrisation de *type* que la paramétrisation de *valeur*, conformément aux limitations suivantes:

a) les éléments linguistiques qui ne peuvent pas être paramétrés sont: **`const`**, **`var`**, **`timer`**, **`control`**, **`group`** et **`import`**;

b) l'élément linguistique **`module`** permet à une paramétrisation de valeur *statique* de prendre en charge des paramètres de suite de tests. En d'autres termes, cette paramétrisation peut être déjà résolue en phase de compilation ou ne pas l'être encore mais doit l'être au commencement de l'exécution (c'est-à-dire qu'elle doit être devenue *statique* au moment de l'exécution et qu'à ce moment les valeurs paramétriques du module sont visibles globalement mais non modifiables);

c) toutes les définitions de **`type`** définies par l'utilisateur (y compris les définitions de types structurés comme **`record`**, **`set`**, etc.) ainsi que le type spécial de configuration **`address`** prennent en charge la paramétrisation de type *statique* et de valeur *statique*, c'est-à-dire que cette paramétrisation doit être résolue au moment de la compilation;

d) les éléments linguistiques **`signature`**, **`testcase`** et **`function`** prennent en charge la paramétrisation de valeur *dynamique* (c'est-à-dire que cette paramétrisation doit pouvoir être résolue au moment de l'exécution);

e) les variantes nommées prennent en charge la paramétrisation de valeur *dynamique* (c'est-à-dire que cette paramétrisation doit pouvoir être résolue au moment de l'exécution). Comme les variantes nommées ne sont pas des unités de portée, les paramètres formels qui ont été définis sont simplement remplacés par les paramètres réels fournis au moment de l'exécution de (la macro de) l'expansion du mot clé **`named alt`**.

Le Tableau 2 donne un résumé des éléments linguistiques qui peuvent être paramétrés et des valeurs qui peuvent leur être affectées comme paramètres.

Tableau 2/Z.140 – Aperçu général des éléments linguistiques TTCN-3 paramétrisables

Mot clé	Paramétrisation de type	Paramétrisation de valeur	Types de valeurs autorisés à apparaître dans des listes de paramètres formels/réels
`module`		Statique au début de l'exécution	*Valeurs de:* tous les types de base, tous les types définis par utilisateur et type **`address`**.
`type`	Statique au moment de la compilation	Statique au moment de la compilation	*Valeurs de:* tous les types de base, tous les types définis par utilisateur et type **`address`**. NOTE – Les définitions de **`record of`**, **`set of`**, **`enumerated`**, **`port`**, **`component`** et **`subtype`** ne permettent pas la paramétrisation.
`template`		Dynamique au moment de l'exécution	*Valeurs de:* tous les types de base, tous les types définis par utilisateur, type **`address`**, type **`component`** et type **`template`**.

Tableau 2/Z.140 – Aperçu général des éléments linguistiques TTCN-3 paramétrisables

Mot clé	Paramétrisation de type	Paramétrisation de valeur	Types de valeurs autorisés à apparaître dans des listes de paramètres formels/réels
function		Dynamique au moment de l'exécution	*Valeurs de:* tous les types de base, tous les types définis par utilisateur, type **address**, type **component**, type **port**, type **template** et type **timer**.
testcase		Dynamique au moment de l'exécution	*Valeurs de:* tous les types de base et de tous les types définis par utilisateur, type **address** et type **template**.
signature		Dynamique au moment de l'exécution	*Valeurs de:* tous les types de base, tous les types définis par utilisateur, type **address** et type **component**.
named alt		Expansion de macro statique	*Valeurs de:* tous les types de base, tous les types définis par utilisateur, type **address**, type **component**, type **port**, type **template** et type **timer**.
NOTE – Des exemples de syntaxe et d'utilisation spécifique de la paramétrisation avec les différents éléments linguistiques sont donnés dans les paragraphes correspondants de la présente Recommandation.			

5.3.1 Paramétrisation par référence et par valeur

Par défaut, tous les paramètres des types de base, des types chaîne de base, des types structurés définis par utilisateurs, des types adresse et des types composant sont transmis par valeur, ce qui peut (facultativement) être indiqué par le mot clé **in**. Afin de transmettre par référence des paramètres des types susmentionnés, le mot clé **out** ou **inout** doit être utilisé.

Les temporisateurs et les accès sont toujours transmis par référence. Ils sont identifiés par les mots clés **timer** et **port**. Le mot clé **inout** peut (facultativement) être utilisé pour indiquer une transmission par référence.

5.3.1.1 Paramétrisation par référence

La paramétrisation par référence est limitée comme suit:

a) seules les listes de paramètres formels associées aux mots clés **function, signature** et **testcase** peuvent contenir des paramètres transmis par référence.

NOTE – D'autres restrictions s'appliquent à la façon d'utiliser dans les signatures les paramètres transmis par référence (voir paragraphe 22).

b) les paramètres réels ne peuvent être que des variables (et non des constantes ou des modèles, par exemple);

c) seuls les paramètres de valeur (c'est-à-dire à l'exclusion des paramètres de type) doivent être transmis par référence.

Exemple:

```
function MyFunction (inout boolean MyReferenceParameter) { … };
// Le paramètre MyReferenceParameter est transmis par référence. Le paramètre
// réel peut être lu et réglé à partir de la fonction

function MyFunction (out boolean MyReferenceParameter) { … };
// Le paramètre MyReferenceParameter est transmis par référence. Le paramètre
// réel ne peut être réglé qu'à partir de la fonction
```

5.3.1.2 Paramétrisation par valeur

Les paramètres réels qui sont transmis par valeur peuvent être des variables aussi bien que des constantes, des modèles, etc.

```
function MyFunction (in template MyTemplateType MyValueParameter) { … };
// Le paramètre MyReferenceParameter est transmis par valeur. Le mot clé in est
// facultatif
```

5.3.2 Listes de paramètres formels et de paramètres réels

Le nombre des éléments et l'ordre dans lesquels ils apparaissent dans la liste des paramètres réels doivent être les mêmes que dans la liste de paramètres formels correspondante. Par ailleurs, le type de chaque paramètre réel doit être compatible avec le type de chaque paramètre formel correspondant.

Exemple:

```
// Une définition de fonction avec une liste de paramètres formels
function MyFunction (integer FormalPar1, boolean FormalPar2, bitstring
// FormalPar3)
{ … }

// Un appel de fonction avec liste de paramètres réels
MyFunction(123, true, '1100'B);
```

5.3.3 Liste vide de paramètres formels

Si la liste de paramètres formels d'un élément linguistique TTCN-3 paramétrisable et analogue à une fonction (c'est-à-dire de type **function, testcase, signature, named alt** ou **external**) est vide, les parenthèses vides doivent être insérées aussi bien dans la déclaration que dans l'invocation de cet élément. Dans tous les autres cas, les parenthèses vides doivent être omises.

Exemple:

```
// Une définition de fonction avec une liste paramétrique vide doit être
// écrite sous la forme
function MyFonction(){ … }

// Une définition d'enregistrement avec une liste paramétrique vide doit être
// écrite sous la forme
type record MyRecord { … }
```

5.3.4 Listes paramétriques imbriquées

Généralement, toutes les entités paramétrées qui sont spécifiées par un paramètre réel doivent avoir leurs propres paramètres résolus dans la liste de paramètres réels.

Exemple:

```
// Soit la définition de message
type record MyMessageType
{
     integer     field1,
     charstring  field2,
     boolean     field3
}

// Un modèle de message pourrait être:
template MyMessageType MyTemplate(integer MyValue) :=
{
     field1 := MyValue,
     field2 := pattern "abc*xyz",
```

```
     field3 := true
}

// Un test élémentaire paramétré par un modèle pourrait être
testcase TC001(template MyMessageType RxMsg) runs on PTC1 system TS1
{    :
     MyPCO.receive(RxMsg);
}

// Lorsque le test élémentaire est appelé dans la partie commande et lorsque
// le modèle paramétré est utilisé comme paramètre réel, les paramètres réels
// du modèle doivent être fournis
control
{    :
     TC001(MyTemplate(7));
     :
}
```

5.4 Règles de portée

La notation TTCN-3 offre cinq unités de portée de base:

a) les modules;

NOTE – Il existe des règles additionnelles de détection pour les groupes (voir § 7.3.1).

b) partie commande d'un module;

c) fonctions;

d) tests élémentaires;

e) blocs d'instructions dans la partie commande, dans les fonctions et dans les tests élémentaires.

Chaque unité de portée est formée de déclarations (facultatives) plus une certaine forme de description fonctionnelle (facultative). Toutes les unités de portée, sauf les modules, sont hiérarchisées. Chaque niveau hiérarchique définit sa propre portée locale. Les déclarations contenues dans un niveau de portée supérieur sont visibles par les niveaux inférieurs (à l'intérieur du même niveau hiérarchique de portée). Les déclarations contenues dans un niveau de portée inférieur ne sont pas visibles par les niveaux de portée supérieurs.

Exemple:

```
module MyModule
{    :
     const integer MyConst := 0;  // MyConst est visible par MyBehaviourA etd
                                  // MyBehaviourB
     :
     function MyBehaviourA()
     { :
       const integer A := 1;      // La constante A n'est visible que par
                                  // MyBehaviourA
       :
     }

     function MyBehaviourB()
     { :
       const integer B := 1;      // La constante B n'est visible que par
                                  // MyBehaviourB
       :

     }
}
```

5.4.1 Portées et reprise d'identificateurs

La notation TTCN-3 ne prend pas en charge la reprise d'identificateurs, c'est-à-dire que tous les identificateurs contenus dans un même niveau hiérarchique de portée doivent être uniques. En d'autres termes, une déclaration effectuée dans un niveau de portée inférieur ne doit pas réutiliser le même identificateur comme déclaration à un niveau de portée supérieur (et dans la même hiérarchie de portée).

Exemple:

```
module MyModule
{    :
     const integer A := 1;
     :
     function MyBehaviourA()
     { :
       const integer A := 1; // N'est pas permis
       :
       if(...)
       {  :
          const boolean A := true; // N'est pas permis
          :
       }
     }
}

// Ce qui suit EST permis car les constantes ne sont pas déclarées dans la
// même hiérarchie de portée (en supposant qu'il n'y a pas de déclaration
// de A dans l'en-tête de module)
function MyBehaviourA()
{    :
     const integer A := 1;
     :
}

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

5.4.2 Portée des paramètres formels

La portée des paramètres formels dans un élément linguistique (par exemple, un appel de fonction) doit être limitée à la définition dans laquelle ces paramètres apparaissent et aux niveaux inférieurs de portée dans la même hiérarchie de portée. En d'autres termes, ces paramètres suivent les règles de portée normales (voir § 5.4). Les règles de reprise d'identificateur (voir § 5.4.1) doivent également s'appliquer aux paramètres formels.

5.5 Identificateurs et mots clés

Les identificateurs de la notation TTCN-3 sont sensibles à l'inversion majuscules/minuscules et les mots clés TTCN-3 doivent être écrits en minuscules (voir Annexe A).

6 Types et valeurs

La notation TTCN-3 prend en charge un certain nombre de types de base prédéfinis qui comprennent ceux qui sont normalement associés à un langage de programmation, comme **integer**, **boolean** et les types chaîne, ainsi que certains types spécifiques de la notation TTCN-3 comme **objid** et **verdicttype**. Les types structurés comme **record**, **set** et **enumerated** peuvent être construits à partir de ces types de base.

Il est possible d'utiliser des types spéciaux en association avec des configurations comme **address**, **port** et **component** afin de définir l'architecture du système de test (voir § 21).

Les types TTCN-3 sont résumés dans le Tableau 3.

Tableau 3/Z.140 – Aperçu général des types TTCN-3

Classe de type	Mot clé	Sous-type
Types de base	**integer**	étendue, liste
	char	étendue, liste
	universal char	étendue, liste
	float	liste
	boolean	liste
	objid	liste
	verdicttype	liste
Types chaîne de base	**bitstring**	liste, longueur
	hexstring	liste, longueur
	octetstring	liste, longueur
	charstring	liste, longueur
	universal charstring	liste, longueur
Types structurés définis par l'utilisateur	**record**	liste
	record of	liste
	set	liste
	set of	liste
	enumerated	liste
	union	liste
Types de configuration spéciale	**address**	
	port	
	component	

6.1 Types et valeurs de base

La notation TTCN-3 prend en charge les types de base suivants:

a) **integer**: type possédant des valeurs distinctives qui sont les nombres entiers positifs et négatifs, y compris zéro.

Les valeurs de type entier doivent être désignées par un ou par plusieurs chiffres; le premier chiffre ne doit pas être zéro à moins que la valeur ne soit zéro. La valeur zéro doit être représentée par un seul zéro.

b) **char**: type dont les valeurs distinctives sont des caractères extraits de la Rec. UIT-T T.50 et l'ISO/CEI 646 [5].

Les valeurs du type **char** peuvent être englobées dans des guillemets droits (") ou être calculées au moyen d'une fonction de conversion prédéfinie utilisant comme argument la valeur d'entier positif de leur codage.

Un ordre est défini entre les valeurs de type **char** selon la valeur d'entier de leur codage. C'est-à-dire que les opérateurs relationnels = =, <, >, ! =, >= et <= peuvent être utilisés afin de comparer des valeurs de type **char**.

c) **universal char**: type dont les valeurs distinctives sont des caractères extraits de l'ISO/CEI 10646-1 [6].

Les valeurs du type **universal char** peuvent être englobées dans des guillemets droits (") ou être calculées au moyen d'une fonction de conversion prédéfinie utilisant comme argument la valeur d'entier positif de leur codage.

Un ordre est défini entre les valeurs de type **char** selon la valeur d'entier de leur codage. C'est-à-dire que les opérateurs relationnels = =, <, >, ! =, >= et <= peuvent être utilisés afin de comparer des valeurs de type **universal char**.

d) **float**: type décrivant les nombres en virgule flottante.

Les nombres en virgule flottante sont représentés sous la forme: $<mantisse>*<base>^{<exposant>}$

où la mantisse est un entier positif ou négatif, la base est un entier positif (le plus souvent 2, 10 ou 16) et où l'exposant est un entier positif ou négatif.

La représentation d'un nombre en virgule flottante est limitée à une base de valeur 10. Les valeurs à virgule flottante peuvent être exprimées soit:

- en notation normale avec un point dans une séquence de nombre, comme 1.23 (qui représente $123*10^{-2}$), 2.783 (c'est-à-dire $2783*10^{-3}$) ou −123.456789 (qui représente $-123456789*10^{-6}$);
- ou par deux nombres séparés par la lettre E, le premier nombre spécifiant la mantisse et le second spécifiant l'exposant, comme dans 12.3E4 (qui représente $12.3*10^{4}$) ou −12.3E-4 (qui représente $-12.3*10^{-4}$).

e) **boolean**: type composé de deux valeurs distinctives.

Les valeurs de type booléen doivent être exprimées par **true** ou **false**.

f) **objid**: type dont les valeurs distinctives sont l'ensemble de tous les identificateurs d'objet attribués conformément aux règles de [7], [8], [9] et [10]. Par exemple:

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

ou, en variante: { itu-t identified-organization etsi }

ou, en variante: { 0 4 0 }.

g) **verdicttype**: type à utiliser avec des verdicts de test composés de 4 valeurs distinctives.

Les valeurs de type verdict doivent être exprimées par **pass**, **fail**, **inconc**, **none** et **error**.

6.1.1 Types et valeurs de chaîne de base

La notation TTCN-3 prend en charge les trois types de chaîne de base suivants:

NOTE – En notation TTCN-3, le terme général *chaîne* ou *type (de) chaîne* se rapporte aux mots clés **bitstring**, **hexstring**, **octetstring**, **charstring** et **universal charstring**.

a) **bitstring**: type dont les valeurs distinctives sont les séquences ordonnées de zéro, un ou plusieurs éléments binaires.

Les valeurs de type **bitstring** doivent être désignées par un nombre arbitraire (éventuellement zéro) de zéros et de uns, précédés d'une apostrophe (') et suivis de la paire de caractères 'B. Par exemple:

```
'01101'B
```

b) **hexstring**: type dont les valeurs distinctives sont les séquences ordonnées de zéro, un ou plusieurs chiffres hexadécimaux, correspondant chacun à une séquence ordonnée de quatre éléments binaires.

Les valeurs de type **hexstring** doivent être désignées par un nombre arbitraire (éventuellement zéro) de chiffres hexadécimaux:

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

précédés par une apostrophe (') et suivis par la paire de caractères 'H. Chaque chiffre hexadécimal est utilisé pour indiquer la valeur d'un demi-octet en représentation hexadécimale. Par exemple:

```
'AB01D'H
```

c) **octetstring**: type dont les valeurs distinctives sont les séquences ordonnées de zéro ou d'un nombre pair positif de chiffres hexadécimaux (chaque paire de chiffres correspondant à une séquence ordonnée de huit éléments binaires).

Les valeurs de type **octetstring** doivent être indiquées par un nombre pair arbitraire (pouvant être zéro) de chiffres hexadécimaux

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

précédés d'une apostrophe (') et suivis par la paire de caractères 'O. Chaque chiffre hexadécimal sert à indiquer la valeur d'un demi-octet en représentation hexadécimale. Par exemple:

```
'FF96'O
```

d) **charstring**: type dont les valeurs distinctives sont zéro, un ou plusieurs caractères extraits de la Rec. UIT-T T.50 et l'ISO/CEI 646 [5]. Le type chaîne de caractères qui est précédé du mot clé **universal** désigne les types dont les valeurs distinctives sont zéro, un ou plusieurs caractères extraits de l'ISO/CEI 10646-1 [6].

Les valeurs de type **charstring** et **universal charstring** doivent être indiquées par un nombre arbitraire (pouvant être zéro) de caractères extraits du jeu de caractères approprié et suivis d'un guillemet droit (").

S'il est nécessaire de définir des chaînes contenant le caractère de guillemet droit ("), ce caractère est représenté par une paire de guillemets droits sur la même ligne sans caractères d'espacement intermédiaires. Par exemple, la chaîne "abcd" représente la chaîne de littéraux "abcd".

6.1.2 Accès à des éléments individuels de chaîne

Les éléments individuels d'un type chaîne peuvent être manipulés au moyen d'une syntaxe de type séquence tabulaire. Seuls les éléments isolés de la chaîne peuvent être manipulés.

Les unités de longueur des différents éléments de type chaîne sont indiquées dans le Tableau 4.

Tableau 4/Z.140 – Unités de longueur utilisées dans les spécifications de longueur de champ

Type	Unités de longueur
bitstring	bits
hexstring	chiffres hexadécimaux
octetstring	octets
character strings	caractères

L'indexation doit commencer par la valeur zéro (0). Par exemple:

```
// Si l'on a
MyBitString := '11110111'B;
// Alors l'opération
MyBitString[4] := '1'B;
// Donne la chaîne binaire '11111111'B
```

6.2 Sous-types et valeurs définis par l'utilisateur

Les types définis par l'utilisateur doivent être indiqués par le mot clé **type**. Avec des types définis par l'utilisateur, il est possible de construire des sous-types (comme des listes, des étendues et des restrictions de longueur) à partir du type **integer** et des divers types chaîne.

6.2.1 Liste de valeurs

La notation TTCN-3 permet de spécifier une liste de valeurs distinctives de type quelconque, comme énuméré dans le Tableau 3. Les valeurs contenues dans la liste doivent être du type de base et doivent être un sous-ensemble vrai des valeurs définies par le type de base. Le sous-type défini par cette liste limite les valeurs autorisées du sous-type à celles qui figurent dans la liste. Par exemple:

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

6.2.2 Etendues

La notation TTCN-3 permet de spécifier une étendue de valeurs de type **integer**, **char** et **universal char** (ou des dérivés de ces types). Le sous-type défini par le contenu de cette étendue limite les valeurs autorisées du sous-type à celles qui se trouvent dans l'étendue située entre la limite inférieure et la limite supérieure incluses. Par exemple:

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

6.2.2.1 Etendues infinies

Afin de spécifier une étendue d'entiers infinie, le mot clé **infinity** peut être utilisé à la place d'une valeur indiquant qu'il n'y a ni limite inférieure ni limite supérieure. Par exemple:

```
type integer MyIntegerRange (-infinity .. -1); // Ne contenant que des
                                               // nombres entiers négatifs
```

NOTE – La "valeur" de l'infinité dépend de la réalisation. L'utilisation de cette caractéristique peut conduire à des problèmes de portabilité.

6.2.2.2 Mélange de listes et d'étendues

Pour les valeurs de type **integer**, **char** et **universal char** (ou des dérivés de ces types), il est possible de mélanger listes et étendues. Par exemple:

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

6.2.3 Restrictions de longueur de chaîne

La notation TTCN-3 permet de spécifier des restrictions de longueur applicables aux types chaîne. Les limites de longueur sont de complexité différente selon le type de chaîne auquel elles sont appliquées. Dans tous les cas, ces limites doivent donner des valeurs non négatives de type **integer** (ou des valeurs dérivées de type **integer**). Par exemple:

```
type bitstring MyByte length(8);                    // Longueur exacte 8
type bitstring MyByte length(8 .. 8);               // Longueur exacte 8
type bitstring MyNibbleOrByte length(4 .. 8);       // Longueur minimale 4,
                                                    // Longueur maximale 8
```

Le Tableau 4 spécifie les unités de longueur pour différents types de chaîne.

Pour la limite supérieure, le mot clé **infinity** peut également être utilisé pour indiquer qu'il n'y a pas de limite supérieure de longueur. La limite supérieure doit être plus grande ou égale à la limite inférieure.

6.3 Types et valeurs structurés

Le mot clé **type** est également utilisé pour spécifier des types structurés comme **record**, **record of**, **set**, **set of**, **enumerated**, **union**.

Les valeurs de ces types peuvent être indiquées au moyen d'une notation d'affectation explicite ou d'un initialiseur de notation abrégée. Par exemple:

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

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

Il n'est pas permis de mélanger les deux notations de valeur dans le même contexte (immédiat). Par exemple:

```
// Cela est interdit
const MyRecordType MyRecordValue:= {MyIntegerValue, field2 := true, "une
//chaîne"}
```

6.3.1 Types et valeurs d'enregistrement

La notation TTCN-3 prend en charge le type structuré et ordonné qui est désigné par le mot clé **record**. Les éléments d'un type enregistrement peuvent être de l'un quelconque des types de base ou des types définis par l'utilisateur comme d'autres enregistrements, des ensembles ou des séquences tabulaires. Les valeurs d'un enregistrement doivent être compatibles avec les types des champs de cet enregistrement. Les identificateurs d'élément sont localisés dans l'enregistrement et doivent y être uniques. Une constante qui est de type enregistrement ne doit contenir aucune variable (y compris les paramètres de module) en tant que valeur de champ, directement ou indirectement.

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

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

Les enregistrements peuvent être définis sans champs (c'est-à-dire en tant qu'enregistrements vides). Par exemple:

```
type record MyEmptyRecord {  }
```

Une valeur de type **record** est attribuée au titre d'un élément individuel. Par exemple:

```
var integer MyIntegerValue:= 1;

var MyRecordType MyRecordValue:=
{
     field1 := MyIntegerValue,
```

```
     field2 := MyOtherRecordValue,
     field3 := "A string"
}

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

ou au moyen d'un initialiseur. Par exemple:

```
MyRecordValue:= {MyIntegerValue, {'11001'B, true}, "une chaîne"};
```

Dans le cas de champs facultatifs, il est permis d'omettre la valeur au moyen du symbole d'omission de paramètre. Par exemple:

```
MyRecordValue:= {MyIntegerValue, - , "une chaîne"};

// Noter que cela revient à écrire ce qui suit, c'est-à-dire que la valeur du
// champ field2 est indéfinie:
MyRecordValue.field1 := MyIntegerValue;
MyRecordValue.field3 := "une chaîne"
```

6.3.1.1 Référencement de champs d'enregistrement imbriqués

Les éléments d'enregistrements imbriqués sont référencés par des paires d'identificateurs *RecordId.Element.Id*. Par exemple:

```
MyVar1 := MyRecord1.MyElement1;
// Si un enregistrement est imbriqué alors la référence peut ressembler à ceci:
MyVar2 := MyRecord1.MyElement1.MyRecord2.MyElement2;
```

6.3.1.2 Eléments facultatifs d'un enregistrement

Les éléments facultatifs d'un enregistrement (**record**) doivent être spécifiés au moyen du mot clé **optional**. Par exemple:

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

6.3.2 Types et valeurs d'ensemble

La notation TTCN-3 prend en charge les types structurés non ordonnés qui sont désignés par le mot clé **set**. Les types et valeurs d'un ensemble sont similaires à ceux d'un enregistrement sauf que l'ordre des champs d'un ensemble n'est pas significatif. Par exemple:

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

La notation d'initialiseur pour régler les valeurs ne doit pas être utilisée pour les valeurs des types **set**.

6.3.2.1 Eléments facultatifs d'un ensemble

Les éléments facultatifs d'un ensemble **set** doivent être spécifiés au moyen du mot clé **optional**.

6.3.3 Enregistrements et ensembles de types particuliers

La notation TTCN-3 prend en charge la spécification d'enregistrements et d'ensembles dont les éléments sont tous du même type et spécifiés au moyen du mot clé **of**. Ces enregistrements et ensembles ne possèdent pas d'identificateurs d'élément et peuvent être considérés comme étant respectivement analogues à une séquence tabulaire ordonnée et à une séquence tabulaire non ordonnée.

Le mot clé **length** sert à limiter la longueur des types **record** et **set of**. Par exemple:

```
type record of length(10) integer MyRecordOfType;      // est un enregistrement
                                                        // d'un maximum de
                                                        // 10 entiers
type set of boolean MySetOfType;                        // est un ensemble
                                                        // illimité de valeurs
                                                        // booléennes
type record of length(10) charstring StringArray length(10);
                                  // est un enregistrement d'un maximum de
                                  // 10 chaînes ayant chacune une longueur
                                  // maximale de 10 caractères
```

La notation de valeur pour **record** et **set of** est identique à la notation pour les séquences tabulaires (voir § 6.4).

6.3.4 Types et valeurs d'énumération

La notation TTCN-3 prend en charge les types énumérés, qui servent à modéliser des types ne prenant qu'un ensemble nommé distinct de valeurs. Les opérations effectuées sur les types énumérés ne doivent utiliser que les identificateurs nommés et sont limitées aux opérateurs d'affectation, d'équivalence et de séquencement.

Chaque valeur nommée peut (facultativement) avoir une valeur d'entier associée qui est définie après le nom entre parenthèses. Ces valeurs ne sont utilisées par le système que pour permettre d'utiliser des opérateurs relationnels. Si aucun entier explicite n'est indiqué, l'on suppose que la séquence commence par zéro. Par exemple:

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

// Une instanciation valide de MyEnumType serait:
var MyEnumType Today    := Monday;
var MyEnumType Tomorrow := Tuesday;
// et la déclaration Today < Tomorrow est vraie
```

6.3.5 Réunion logique d'ensembles

La notation TTCN-3 prend en charge les types **union**, qui sont similaires aux types **record**, sauf qu'un seul des champs spécifiés sera jamais présent dans une valeur réelle de réunion logique. Les types **union** sont utiles pour modéliser une structure qui peut prendre un nombre fini de types connus. Par exemple:

```
type union MyUnionType
{
     integer        number,
     charstring     string
}
// Une instanciation valide de MyUnionType serait:
var MyUnionType age;
age.number := 34;
```

La notation d'initialiseur pour régler les valeurs ne doit pas être utilisée pour les valeurs des types **union**.

Le mot clé **optional** ne doit pas être utilisé avec des types **union**.

6.4 Séquences tabulaires

En commun avec de nombreux langages de programmation, les séquences tabulaires ne sont pas considérées comme étant des types en notation TTCN-3. Elles peuvent en revanche être spécifiées au point d'une déclaration de variable. Par exemple:

```
var integer MyArray[3];      // Instancie une séquence tabulaire d'entiers de
                             // 3 éléments avec l'index 0 à 2
```

Les valeurs des éléments d'une séquence tabulaire doivent être compatibles avec la déclaration de variable correspondante. Les valeurs peuvent être attribuées individuellement ou collectivement. Par exemple:

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

// ou au moyen d'un initialiseur:
MyArray:= {10, 20, 30};
```

Les index de séquence tabulaire sont des expressions qui doivent prendre des valeurs de type **integer** positives, y compris la valeur zéro. Par défaut, l'indexation des séquences tabulaires TTCN-3 doit commencer par le chiffre 0 (zéro).

Les dimensions d'une séquence tabulaire doivent être spécifiées au moyen d'expressions constantes qui doivent prendre une valeur positive de type **integer**. Les dimensions d'une séquence tabulaire peuvent également être spécifiées au moyen d'étendues. Dans ce cas, les valeurs inférieures et supérieures des étendues définissent les valeurs inférieures et supérieures des index. Par exemple:

```
var integer MyArray[1 .. 5]; // Instancie une séquence tabulaire d'entiers de
                             // 5 éléments avec l'index 1 à 5
MyArray[1] := 10; // Index le plus bas
MyArray[5] := 50; // Index le plus haut
```

Les séquences tabulaires enregistrant des types **record** permettent de spécifier des séquences tabulaires à plusieurs dimensions. Par exemple:

```
// Si l'on a
type record MyRecordType
{
     integer            field1,
     MyOtherStruct      field2,
     charstring         field3
}
// Une séquence tabulaire du type MyRecordType pourrait être:
var MyRecordType MyRecordArray[10];
// Une référence à un élément particulier aurait l'allure suivante:
     MyRecordArray[1].field1 := 1;
```

6.5 Types récursifs

Si applicable, les définitions de type TTCN-3 peuvent être récursives. L'utilisateur doit cependant veiller à ce que la récurrence des types soit résoluble et à ce qu'il n'y ait pas de récurrence en boucle.

6.6 Paramétrisation des types

La paramétrisation des types permet d'insérer des identificateurs de type fictifs qui jouent le rôle de remplaçant pour tout type. En d'autres termes, un type peut être laissé ouvert par le spécificateur TTCN-3 à condition qu'il puisse être résolu lors de la compilation.

NOTE – Il s'agit d'une généralisation du concept de métatype d'unité PDU dans la notation TTCN-2.

Le type réel n'est connu que lorsque le paramètre de type est effectivement utilisé. Par exemple:

```
type record MyRecordType(MyMetaType)
{
     boolean  field1,
     MyMetaType  field2 // MyMetaType n'est pas d'un type particulier
}

var MyRecordType(integer) MyRecordValue :=
{
     field1 := true,
     field2 := 123           // MyMetaType est maintenant de type entier
}
```

6.7 Compatibilité des types

La notation TTCN-3 n'est pas fortement typée mais ce langage exige tout de même la compatibilité des types. Les variables, constantes, modèles, etc. sont compatibles en termes de types s'ils aboutissent au même type de base et, en cas d'affectations, d'appariement, etc., si aucun sous-typage n'est violé (par exemple, étendues, restrictions de longueur).

Par exemple:

```
// Si l'on a
type integer MyInteger(1 .. 10);
:
var integer x;
var MyInteger y;

// Then
x := 20;  // est une affectation valide
y := 20;  // n'est PAS une affectation valide parce que 20 n'est pas dans
          // l'étendue de y

y := 5;   // est une affectation valide

x := y;   // est une affectation valide, parce que la valeur de y est dans
          // l'étendue de x
y := x;   // n'est PAS une affectation valide, parce que la valeur de x n'est
          // pas dans l'étendue de y

x := 5;   // est une affectation valide
y := x;   // est une affectation valide, parce que la valeur de x est
          // maintenant dans l'étendue de y
```

6.7.1 Conversion de type

S'il est nécessaire de convertir des valeurs d'un certain type en valeurs d'un autre type et si ces types ne sont pas issus du même type de base, il faut utiliser l'une des fonctions de conversion prédéfinies dans l'Annexe D ou une fonction définie par l'utilisateur. Par exemple:

```
// Pour convertir une valeur d'entier en valeur de chaîne hexadécimale,
// utiliser la fonction prédéfinie int2hex
     MyHstring := int2hex(123, 4);
```

7 Modules

Les principaux blocs de construction de la notation TTCN-3 sont les modules. Par exemple, un module peut définir une suite de tests exécutables proprement dite ou simplement une bibliothèque. Un module se compose d'une partie définitions (facultative) et d'une partie commande de module (facultative).

NOTE – Le terme "suite de tests" est synonyme d'un module TTCN-3 complet, contenant des tests élémentaires et une partie commande.

7.1 Nommage des modules

Les noms des modules sont exprimés par un identificateur TTCN-3 suivi d'un identificateur d'objet facultatif.

NOTE – L'identificateur de module est le nom alphanumérique informel de ce module.

7.2 Paramétrisation des modules

La liste paramétrique **module** définit un ensemble de valeurs qui sont fournies par l'environnement de test au moment de l'exécution et qui doivent à partir de ce moment être traitées comme des constantes. Par exemple:

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

NOTE – Cette paramétrisation offre une fonctionnalité similaire aux paramètres de suite de tests TTCN-2 qui fournissent des valeurs PICS et PIXIT à la suite de tests.

7.2.1 Valeurs par défaut des paramètres de module

Lorsque les valeurs des paramètres réels d'un module ne sont pas fournies par l'environnement de test au moment de l'exécution, il est permis de spécifier des valeurs par défaut pour les paramètres de module. Cela doit être effectué par une affectation dans la liste paramétrique du module. Par exemple:

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

7.3 Partie définitions d'un module

La partie définitions d'un module spécifie les définitions de niveau sommital du module. Ces définitions peuvent être utilisées partout ailleurs dans le module, y compris dans la partie commande. Les éléments linguistiques qui peuvent être définis dans un module TTCN-3 sont énumérés dans le Tableau 1. Les définitions de module peuvent être importées par d'autres modules.

Exemple:

```
module MyModule
{    // Ce module ne contient que des définitions
     :
     const integer MyConstant := 1;
     type record MyMessageType { ... }
     :
```

```
     function TestStep(){ ... }
     :
}
```

Les déclarations d'éléments linguistiques dynamiques comme **var** ou **timer** ne doivent être faites que dans la partie commande, dans les tests élémentaires ou dans les fonctions.

NOTE – La notation TTCN-3 ne prend pas en charge la déclaration de variables dans la partie définitions d'un module. Elle ne les prend en charge que dans la partie commande. En d'autres termes, des variables globales ne peuvent pas être définies en notation TTCN-3.

7.3.1 Groupes de définitions

Dans la partie définitions d'un module, les définitions peuvent être recueillies dans des groupes nommés. Un groupe de déclarations peut être spécifié chaque fois qu'une déclaration isolée est autorisée. Les groupes peuvent être imbriqués, c'est-à-dire qu'ils peuvent contenir d'autres groupes. Cela permet au spécificateur de suite de tests de structurer, en particulier, des recueils de données de test ou des fonctions décrivant le comportement de test.

Les groupements sont effectués afin de faciliter la lisibilité et d'ajouter une structure logique à la suite de tests, si nécessaire. Cela signifie que tous les identificateurs des déclarations contenues dans l'ensemble de groupes (y compris les éventuels groupes imbriqués) doivent être uniques à tout niveau de groupement donné. En d'autres termes, les groupes (imbriqués ou non) n'ont pas de portée *sauf* dans le contexte d'éventuels attributs conférés au groupe par une instruction **with** associée. Dans de tels cas, une instruction **with** concernant un groupe extérieur est neutralisée par une instruction **with** concernant un groupe intérieur.

Exemple:

```
// Un recueil de définitions
group MyGroup
{
     const integer MyConst:= 1;
     :
     type record MyMessageType { ... }
     }
     // Un groupe d'étapes de test
     group MyTestStepLibrary
     {
       group MyGroup1
       {
          function MyTestStep11() { ... }
          function MyTestStep12() { ... }
          :
          function MyTestStep1n() { ... }
       }
       group MyGroup2
       {
          function MyTestStep21() { ... }
          function MyTestStep22() { ... }
          :
          function MyTestStep2n() { ... }
       }
     }
```

7.4 Partie commande d'un module

La partie commande d'un module décrit l'ordre d'exécution (avec d'éventuelles répétitions) des tests élémentaires proprement dits. Un test élémentaire doit être défini dans la partie définitions du module et être appelé dans la partie commande.

Exemple:

```
module MyTestSuite
{    // Ce module contient des définitions ...
     :
     const integer MyConstant := 1;
     type record MyMessageType { ... }
     template MyMessageType MyMessage := { ... }
     :
     function MyFunction1() { ... }
     function MyFunction2() { ... }
     :
     testcase MyTestcase1() runs on MyMTCType { ... }
     testcase MyTestcase2() runs on MyMTCType { ... }
     :
     // ... et une partie commande de sorte qu'il est exécutable
     control
     {
       var boolean MyVariable; // Variable de commande locale
       :
       MyTestCase1(); // exécution séquentielle de test élémentaires
       MyTestCase2();
       :
     }
}
```

7.5 Importation à partir de modules

Au moyen de l'instruction **import**, il est possible de réutiliser des définitions spécifiées dans différents modules. La notation TTCN-3 ne possède pas de création syntaxique explicite pour l'exportation, de sorte que, par défaut, toutes les définitions de module contenues dans la partie définitions d'un module peuvent être importées. Une instruction **import** peut être utilisée à un endroit quelconque de la partie définitions d'un module. Elle ne doit pas être utilisée dans la partie commande.

Si une définition importée possède des attributs (définis au moyen d'une instruction **with**), ces attributs doivent également être importés.

NOTE – Si le module possède des attributs globaux, ceux-ci sont associés aux définitions qui ne possèdent pas ces attributs.

Exemple:

```
module MyModuleA
{    // Ce module contient des définitions et des définitions importées
     :
     const integer MyConstant := 1;
     import all from MyModuleB;  // La portée des définitions importées est
                                 // globale pour MyModuleA
     type record MyMessageType { ... }
     :
     function MyBehaviourC()
     {
       const integer MyConstant := 2;
       // l'importation ne peut pas être effectuée ici
       :
     }
     :
     control
     { // l'importation ne peut pas être effectuée ici
       :
     }
}
```

7.5.1 Règles d'utilisation de l'importation

Les règles suivantes doivent être appliquées lors de l'utilisation de l'importation:

a) seules les définitions de niveau sommital dans le module peuvent être explicitement importées. Les définitions qui apparaissent à un niveau de portée inférieur (par exemple, les constantes locales définies dans une fonction) ne doivent pas être importées;

b) par défaut, toutes les définitions qui dépendent d'autres définitions, par exemple, les types **`record`**, sont importées en même temps que toutes les définitions dont elles dépendent. Si l'on ne souhaite pas importer ces éléments dépendants, l'instruction **`nonrecursive`** peut être utilisée;

c) des groupes de définitions peuvent également être importés. Les groupes ne sont cependant utilisés qu'à des fins de structuration car ils ne possèdent pas d'unités de portée. Il est donc permis d'importer des sous-groupes, c'est-à-dire des groupes définis dans un autre groupe.

7.5.2 Importation de définitions particulières

Des définitions particulières peuvent être importées. Par exemple:

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

7.5.3 Importation de toutes les définitions d'un module

Le contenu entier d'une partie définitions de module (mais non le module réel proprement dit) peut être importé. Par exemple:

```
import all from MyModule;
```

7.5.4 Importation de groupes

Des groupes peuvent être importés. Par exemple:

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

Des sous-groupes, c'est-à-dire des groupes définis à l'intérieur d'un autre groupe, sont également importés par cette instruction.

7.5.5 Importation de définitions de la même sorte

Des blocs de définitions de même sorte peuvent être importés. Par exemple:

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

7.5.6 Importation récursive de définitions complexes

Par défaut, les définitions récursives, c'est-à-dire celles qui se rapportent à d'autres définitions, sont implicitement importées par l'instruction **`import`**. Exemples de définitions récursives: les types **`record`** ainsi que leurs types ou fonctions de composant qui appellent d'autres fonctions. Par exemple:

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

Toutes les définitions implicitement importées sont visibles au niveau sommital du module visible et peuvent être utilisées à la suite de l'instruction d'importation.

NOTE – Les définitions localisées à l'intérieur de définitions environnantes, par exemple, les déclarations de constante localisées dans une fonction, ne seront jamais visibles.

Exemple:

```
// Si l'on a
module MyModuleA
{    :
     function MyBehaviourB() { ... }
     function MyBehaviourA()
     { :
       MyBehaviourB();
       :
       const integer LocalConst:= 1000;
       :
     }
}

// Alors
module MyModuleB
{    :
     import function MyBehaviourA from MyModuleA;
     :
}
// importera également et rendra visible MyBehaviourB. La constante
// LocalConst sera encore imbriquée dans MyBehaviourA et ne sera pas visible
// (à l'extérieur de MyBehaviourA).
```

Si des définitions importées d'un module donné dépendent de définitions contenues dans un autre module, ces dernières sont importées également, c'est-à-dire que l'importation doit implicitement s'étendre aux définitions dépendant du module tiers. Cela est conforme à la règle stipulant qu'une définition importée est traitée de la même façon qu'une définition développée dans le module proprement dit.

Si l'on souhaite empêcher des importations récursives, l'instruction **nonrecursive** doit être utilisée. Par exemple:

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

7.5.7 Traitement des collisions de noms lors d'une importation

Tous les modules TTCN-3 doivent posséder leur propre espace nominatif, dans lequel toutes les définitions doivent être identifiées de manière univoque. Des collisions de noms peuvent par exemple se produire à cause d'importations à partir de modules différents, d'importations de groupes ou d'importations de définitions récursives. Les collisions de noms doivent être résolues par préfixation de la définition importée (provoquant la collision de noms) avec l'identificateur du module duquel elle est importée. Le préfixe et l'identificateur doivent être séparés par un point (**.**).

S'il n'y a pas d'ambiguïtés, la préfixation n'a pas toujours besoin d'être présente lorsque les définitions importées sont utilisées.

Exemple:

```
module MyModuleA
{    :
     type bitstring MyTypeA;
     import type MyTypeA from SomeModuleC;      // Où MyTypeA est du type chaîne
                                                // de caractères
     import type MyTypeB from SomeModuleC;      // Où MyTypeB est du type chaîne
                                                // de caractères
     :
     control
     { :
       var SomeModuleC.MyTypeA MyVar1 := "Test String"; // Un prefixe doit
                                                        // être utilisé
```

```
        var MyTypeA MyVar2 := '10110011'B;  // C'est le module original MyTypeA
        :
        var MyTypeB MyVar3 := "Test String";     // Le préfixe n'a pas besoin
                                                 // d'être utilisé...
        var SomeModuleC.MyTypeB MyVar3 := "Test String";
                                                 // ... mais il peut l'être au
                                                 // besoin
        :
      }
}
```

NOTE – Les définitions portant un nom identique qui a été défini dans des modules différents sont toujours supposées être différentes, même si les définitions réelles sont identiques dans les différents modules. Par exemple, l'importation d'un type qui est déjà défini localement, même avec le même nom, conduirait à la disponibilité de deux types différents dans le module.

7.5.8 Traitement des références multiples à la même définition

L'utilisation de l'importation (**import**) concernant des définitions isolées, des groupes de définitions, des définitions de la même sorte, etc., peut conduire à des situations dans lesquelles la même définition est citée plusieurs fois en référence. Dans ces cas, la définition ne doit être importée qu'une seule fois.

NOTE – Les mécanismes permettant de résoudre de telles ambiguïtés, par exemple, par effacement et envoi d'avertissements à l'utilisateur, sont hors du domaine d'application de la présente Recommandation. Ils devraient être fournis par des outils TTCN-3.

7.5.9 Paramètres d'importation et de module

Si une définition importée utilise un paramètre de module, ce paramètre doit également être inclus.

7.5.10 Définitions importées de modules non TTCN

Le mot clé **language** sert à indiquer les cas où des définitions de type sont importées de modules non TTCN. Par exemple:

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

Par défaut, le langage est la notation TTCN-3. Par exemple:

```
import type MyType from MyModule;
// est identique à
import type MyType from MyModule language "TTCN-3";
```

8 Configurations d'essai

La notation TTCN-3 permet la spécification (dynamique) de configurations de test concurrentes (appelées configuration pour abréger). Une configuration se compose d'un ensemble de composants de test interconnectés avec des accès de communication bien définis et une interface explicite de système de test qui définit les frontières du système de test. (Voir Figure 2.)

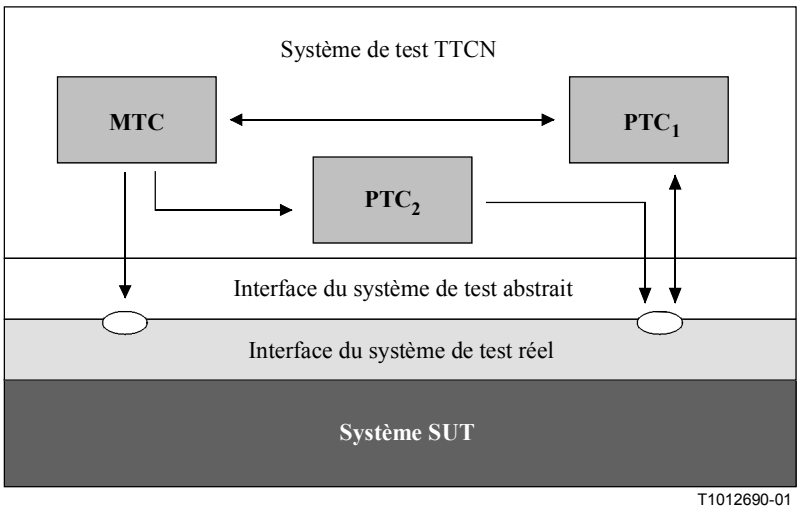


Figure 2/Z.140 – Vue théorique d'une configuration de test TTCN-3 typique

A l'intérieur de chaque configuration, il doit y avoir un (et un seul) composant de test principal (MTC). Les composants de test qui ne sont pas de type MTC sont dénommés composants de test parallèle ou PTC. Le composant MTC doit être créé automatiquement au début de l'exécution de chaque test élémentaire. Le comportement défini dans le corps du test élémentaire doit s'exécuter sur ce composant. Au cours de l'exécution d'un test élémentaire, d'autres composants peuvent être créés dynamiquement par utilisation explicite de l'opération **`create`**.

L'exécution du test élémentaire doit prendre fin lorsque le composant MTC se termine. Tous les autres composants PTC sont traités de manière égale, c'est-à-dire qu'il n'y a pas de relation hiérarchique entre eux. La terminaison d'un composant PTC donné ne met fin à aucun autre composant ni au MTC.

La communication est assurée entre les composants à l'intérieur du système de test et entre les composants et l'interface du système de test, via les accès de communication.

Les types de composant de test et d'accès de test, indiqués par les mots clés **`composant`** et **`port`**, doivent être définis dans la partie définitions du module. La configuration réelle des composants et des connexions établies entre eux est réalisée par l'exécution des opérations **`create`** et **`connect`** dans le comportement de test élémentaire. Les accès de composant sont connectés aux accès de l'interface du système de test au moyen de l'opération **`map`** (voir § 21.2).

8.1 Modèle de communication entre accès

Les composants d'accès peuvent être connectés à d'autres composants et avec l'interface du système de test. Il n'y a pas de restrictions quant au nombre de connexions qu'un composant peut disposer, mais un composant ne doit pas se connecter à lui-même. Des connexions en étoile sont autorisées.

Les composants de test sont connectés par l'intermédiaire de leurs accès, c'est-à-dire que les connexions entre composants, et entre chaque composant et l'interface du système de test, sont orientées vers les accès. Chaque accès est modélisé sous la forme d'une file d'attente qui mémorise les messages ou les appels de procédure jusqu'à ce qu'ils soient traités par le composant détenant cet accès.

NOTE – En principe infinis, les accès TTCN-3 peuvent subir des débordements dans un système de test réel. Ce cas sera traité comme une erreur de test élémentaire (voir § 24.2.1).

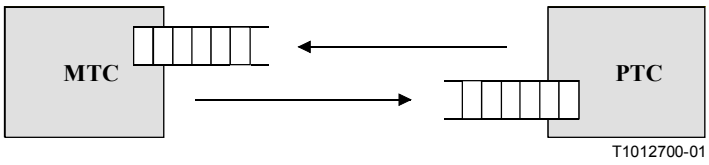


Figure 3/Z.140 – Modèle d'accès de communication TTCN-3

8.2 Interface avec un système de test abstrait

La notation TTCN-3 sert à contrôler des implémentations. L'objet soumis au test est dénommé implémentation sous test (IUT, *implementation under test*). L'implémentation IUT peut offrir des interfaces directes pour les tests ou peut faire partie d'un système, auquel cas l'objet testé est dénommé système sous test (SUT, *system under test*). Dans le cas d'une configuration minimale, la réalisation IUT et le système SUT sont équivalents. Dans la présente Recommandation, le terme système SUT est utilisé de manière générale pour désigner, soit un système SUT, soit une implémentation IUT.

Dans un environnement de tests réels, les tests élémentaires ont besoin de communiquer avec le système SUT. Cependant, la spécification de la connexion physique réelle est hors du domaine d'application de la notation TTCN-3. En revanche, une interface bien définie (mais abstraite) avec un système de test est associée à chaque test élémentaire. Une définition d'interface de système de test est identique à une définition de composant, c'est-à-dire qu'il s'agit d'une liste de tous les accès de communication possibles permettant de connecter le test élémentaire au système SUT.

8.3 Définition des types d'accès de communication

Les accès facilitent la communication entre composants de test et entre ceux-ci et l'interface avec le système de test.

La notation TTCN-3 prend en charge les accès en mode message et en mode procédure. Chaque accès doit être défini comme étant, soit en mode message, soit en mode procédure ou en mode mixte. Le mode doit être indiqué par le mot clé **`message`** ou **`procedure`** dans la définition de type d'accès associée.

Les accès sont directionnels. Leur sens est spécifié par les mots clés **`in`** (dans le sens entrant), **`out`** (dans le sens sortant) et **`inout`** (dans les deux sens). Chaque définition de type d'accès doit comporter une ou plusieurs listes indiquant le recueil autorisé de types (de messages) et/ou de procédures ainsi que le sens de communication autorisé. Par exemple:

```
// Accès en mode message qui permet de recevoir les types MsgType1 et
// MsgType2 à cet accès, d'envoyer le type MsgType3 par l'intermédiaire de
// cet accès et d'envoyer ou de recevoir toute valeur d'entier par cet accès
type port MyMessagePortType message
{
     in       MsgType1, MsgType2;
     out  MsgType3;
     inout    integer
     }

// Accès en mode procédure qui permet l'appel distant des procédures Proc1,
// Proc2 et Proc3. Noter que les procédures Proc1, Proc2 et Proc3 sont définies
// en tant que signatures
type port MyProcedurePortType procedure
{
     out  Proc1, Proc2, Proc3
}
```

NOTE – Le terme "message" signifie ici aussi bien les messages définis par des modèles que les messages définis par des valeurs réelles résultant d'expressions. La liste limitant les utilisations possibles d'un accès en mode message n'est donc qu'une simple liste des noms de type.

L'utilisation du mot clé **all** dans une des listes associées à un type d'accès permet de transmettre par cet accès de communication tous les types et toutes les signatures de procédure définis dans le module. Par exemple:

```
// Accès en mode message qui permet de transférer dans les deux sens, par cet
// accès, toute valeur des types intégrés et définis par l'utilisateur
type port MyProcedurePortType message
{
     inout all
}
```

8.3.1 Accès mixtes

Il est possible de définir un accès comme permettant deux sortes de communication, ce qui est indiqué par le mot clé **mixed**. En d'autres termes, les listes relatives aux accès mixtes seront également mixtes et comporteront à la fois des signatures et des types. Aucune séparation n'est faite dans la définition.

```
// Accès en mode mixte, définissant un accès en mode message et un accès en
// mode procédure avec le même nom. Les listes in, out et inout sont
// également mixtes: les types MsgType1, MsgType2, MsgType3 et integer se
// rapportent à la partie en mode message de l'accès en mode mixte, tandis
// que Proc1, Proc2, Proc3, Proc4 et Proc5 se rapportent à l'accès en mode
// procédure.
type port MyMixedPortType mixed
{
     in       MsgType1, MsgType2, Proc1, Proc2;
     out      MsgType3, Proc3, Proc4;
     inout    integer, Proc5;
}

// Accès en mode mixte permettant d'utiliser tous les types et toutes les
// signatures définis dans le module afin de communiquer avec le système SUT
// ou avec d'autres composants de tests

type port MyAllMixedPortType mixed
{
     inout    all
}
```

Un accès mixte est défini en TTCN-3 comme étant une notation abrégée pour deux accès, soit un accès en mode message et un accès en mode procédure, sous le même nom. Lors du fonctionnement du système, la distinction entre les deux accès est effectuée par les opérations de communication.

Les opérations utilisées pour commander les accès (voir § 21), c'est-à-dire **start, stop** et **clear** doivent s'appliquer aux deux files d'attente (dans un ordre arbitraire) si elles sont appelées avec un identificateur d'accès mixte.

8.4 Définition des types de composant

Le type **component** définit les accès qui sont associés à un composant. Ces définitions doivent être faites dans la partie définitions du module. Les noms d'accès contenus dans une définition de composant sont locaux par rapport à ce composant, c'est-à-dire qu'un autre composant peut avoir des accès portant les mêmes noms. Les accès du même composant doivent avoir des noms uniques, ce qui cependant ne doit pas être interprété comme signifiant que ces accès établissent une connexion quelconque entre les composants.

Exemple:

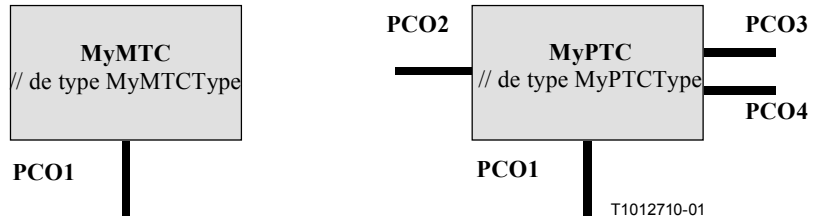


Figure 4/Z.140 – Composants typiques

```
type component MyMTCType
{
     port MyMessagePortType           PCO1
}

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

8.4.1 Déclaration de variables et de temporisations locales dans un composant

Il est possible de déclarer des variables et des temporisations locales par rapport à un composant particulier. Par exemple:

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

Ces déclarations sont visibles par toutes les fonctions qui agissent sur le composant. Cela doit être indiqué explicitement au moyen du mot clé **runs on** (voir § 16).

Les variables et temporisations d'un composant sont associées à l'instance de celui-ci et suivent les règles de portée qui sont définies dans le § 5.1. Chaque nouvelle instance d'un composant possédera donc son propre ensemble de variables et de temporisations comme spécifié dans la définition du composant (y compris d'éventuelles valeurs initiales, si déclarées).

8.4.2 Définition de composants avec séquences tabulaires d'accès

Il est possible de définir des séquences tabulaires d'accès dans les définitions des types de composant (voir également § 21.9). Par exemple:

```
type component My3pcoCompType
{
     port MyMessageInterfaceType PCO[3]
     // Définit un type de composant qui possède une séquence tabulaire de 3 accès.
}
```

8.5 Adressage d'entités à l'intérieur du système SUT

Un système SUT peut se composer de plusieurs entités qui doivent être adressées individuellement. Les données d'adressage sont d'un type utilisé avec les opérations d'accès pour adresser des entités SUT. La représentation réelle des données de type **address** est résolue soit par une définition de type explicite à l'intérieur de la suite de tests soit par le système de test externe (c'est-à-dire que le type **address** reste dans la spécification TTCN-3 sous la forme d'un type

ouvert). Cela permet de spécifier des tests élémentaires abstraits indépendamment de tout mécanisme réel d'adressage propre au système SUT.

Les adresses SUT explicites ne doivent être produites qu'à l'intérieur d'un module TTCN-3 si le type est défini dans ce module; sinon, les adresses SUT explicites dans le module ne doivent être transmises que sous la forme de paramètres ou ne doivent être reçues que dans des champs de message ou en tant que paramètres d'appels de procédure distants.

Par ailleurs, la valeur spéciale **null** permet d'indiquer une adresse indéfinie, par exemple, pour l'initialisation de variables du type adresse.

Exemple:

```
// Associe le type integer à l'adresse du type ouvert
type integer address;
 :
// Nouvelle variable d'adresse initialisée par la valeur null
var address MySUTentity := null;
 :
// Réception d'une valeur d'adresse et son affectation à la variable MySUTentity
PCO.receive(address:*) -> value MySUTentity;
 :
// Utilisation de l'adresse reçue pour l'envoi du modèle MyResult
PCO.send(MyResult) to MySUTentity;
 :
// Utilisation de l'adresse reçue pour réception d'un modèle de confirmation
PCO.receive(MyConfirmation) from MySUTentity;
```

8.6 Références de composant

Les références de composant sont uniques par rapport aux composants de test créés au cours de l'exécution d'un test élémentaire. Cette unique référence de composant est produite par le système de test au moment de la création d'un composant, c'est-à-dire qu'une référence de composant est le résultat d'une opération de type **create** (voir § 21.1). Par ailleurs, les références de composant sont renvoyées par les fonctions prédéfinies de type **system** (renvoi de la référence de composant afin d'identifier les accès de l'interface avec le système de test), de type **mtc** (renvoi de la référence du composant MTC) et de type **self** (renvoi de la référence du composant dans lequel la fonction **self** est appelée).

Les références de composant sont utilisées dans les opérations de configuration **connect, map** et **start** (voir § 21) afin d'établir des configurations de test. Elles sont également utilisées dans les parties **from**, **to** et **sender** des opérations de communication aux fins de l'adressage (voir § 22).

Par ailleurs, la valeur spéciale **null** permet d'indiquer une adresse indéfinie, par exemple, pour l'initialisation de variables manipulant des références de composant.

La représentation réelle des données de référence de composant doit être résolue par le système de test externe. Cela permet de spécifier des tests élémentaires abstraits indépendamment de tout environnement d'exécution réelle de la notation TTCN-3. En d'autres termes, la notation TTCN-3 ne limite pas l'implémentation d'un système de test en ce qui concerne le traitement et l'identification des composants de test.

NOTE – Une référence de composant comporte des informations de type. C'est-à-dire que, par exemple, une variable de traitement de références de composant doit toujours utiliser dans sa déclaration le nom correspondant du type de composant.

Exemple:

```
// Une définition de type de composant
type component MyCompType {
     port PortTypeOne PCO1;
```

```
     port PortTypeTwo PCO2
}
// Déclaration de deux variables pour le traitement de références à des
// composants de type MyCompType et création d'un composant de ce type
var MyCompType MyCompInst := MyCompType.create;

// Utilisation de références de composant dans des opérations de
// configuration se rapportant toujours au composant créé ci-dessus
connect(self:MyPCO1, MyCompInst:PCO1);
map(MyCompInst:PCO2, system:ExtPCO1);
MyCompInst.start(MyBehavior(self));   // le mot clé "self" est transmis comme
                                      // paramètre à MyBehavior

// Utilisation des références de composant dans les clauses "from" et "to"
MyPCO1.receive from MyCompInst;
 :
MyPCO2.receive(integer:*) -> sender MyCompInst;
 :
MyPCO1.receive(MyTemplate) from MyCompInst;
 :
MPCO2.send(integer:5) to MyCompInst;

// L'exemple suivant explique le cas d'une connexion point-multipoint à un
// accès PCO1 où les valeurs de type M1 peuvent être reçues de plusieurs
// composants des différents types CompType1, CompType2 and CompType3 et où
// l'expéditeur doit être extrait. Dans ce cas, le système suivant peut être
// utilisé:
 :
var M1 MyMessage, MyResult;
var MyCompType1 MyInst1 := null;
var MyCompType2 MyInst2 := null;
var MyCompType3 MyInst3 := null;
 :
alt {
     [] PCO1.receive(M1:*) from MyCompType1 -> value MyMessage sender MyInst1 {}
     [] PCO1.receive(M1:*) from MyCompType2 -> value MyMessage sender MyInst2 {}
     [] PCO1.receive(M1:*) from MyCompType3 -> value MyMessage sender MyInst3 {}
}
 :
MyResult := MyMessageHandling(MyMessage);      // Un certain résultat est
                                               // extrait d'une fonction
 :
if (MyInst1 != null) {PCO1.send(MyResult) to MyInst1};
if (MyInst2 != null) {PCO1.send(MyResult) to MyInst2};
if (MyInst3 != null) {PCO1.send(MyResult) to MyInst3};
 :
```

8.7 Définition de l'interface avec le système de test

Une définition de type de composant est utilisée afin de définir l'interface avec le système de test parce que, théoriquement, les définitions de type de composant et d'interface avec le système de test ont la même forme (car ce sont toutes les deux des ensembles d'accès définissant des points de connexion possibles).

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

Généralement, une référence de type de composant, définissant l'interface avec le système de test, est associée à chaque test élémentaire. Les accès de l'interface avec le système de test sont automatiquement instanciés avec le composant MTC au début de l'exécution du test élémentaire, c'est-à-dire lorsque le test élémentaire est appelé par la partie commande du module.

L'opération qui renvoie l'adresse de l'interface avec le système de test est nommée **system**. Elle peut servir à adresser les accès du système de test. Par exemple:

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

Si le composant MTC est le seul composant qui soit instancié au cours de l'exécution du test, une interface avec le système de test n'a pas besoin d'être associée au test élémentaire. Dans ce cas, la définition de type de composant associée au MTC définit implicitement l'interface correspondante avec le système de test.

9 Déclaration des constantes

Les constantes peuvent être déclarées et utilisées dans un en-tête de module, dans la partie commande d'un module, dans des tests élémentaires et dans des fonctions. Les définitions des constantes sont désignées par le mot clé **const**. La valeur de la constante doit être attribuée au point de déclaration. Par exemple:

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

L'affectation de la valeur à la constante peut être effectuée à l'intérieur du module ou de façon externe. Dans ce dernier cas, il s'agit d'une déclaration de constante externe qui est désignée par le mot clé **external**. Les constantes externes doivent être analysées comme des valeurs au moment de la compilation. Par exemple:

```
external const integer MyExternalConst;    // external constant declaration
```

Une constante externe peut être d'un type quelconque mais ce type doit être connu dans le module, c'est-à-dire qu'il s'agira d'un type de base défini dans le module ou importé d'un autre module. Le mappage du type sur la représentation externe d'une constante externe est également hors du domaine d'application de la présente Recommandation. Le mécanisme d'importation d'une constante externe dans un module est hors du domaine d'application de la présente Recommandation.

10 Déclaration des variables

Les variables sont indiquées par le mot clé **var**. Elles peuvent être déclarées et utilisées dans la partie commande d'un module, dans les tests élémentaires et dans les fonctions. Elles ne doivent pas être déclarées ou utilisées dans un en-tête de module (c'est-à-dire que les variables globales ne sont pas prises en charge par la notation TTCN-3). Une déclaration de variable peut avoir une valeur initiale qui lui est attribuée à titre facultatif. Par exemple:

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

L'utilisation de variables non initialisées au moment de l'exécution doit provoquer une erreur de test élémentaire.

11 Déclaration des temporisations

Des temporisations peuvent être déclarées et utilisées dans la partie commande d'un module, dans les tests élémentaires et dans les fonctions. Elles ne doivent être ni déclarées ni utilisées dans la partie définitions d'un module. Une déclaration de temporisation peut avoir une valeur de durée par défaut qui lui est attribuée à titre facultatif. Le temporisateur doit être armé avec cette valeur si aucune autre valeur n'est spécifiée. Cette valeur doit être du type **`float`** si l'unité de base est la seconde. Par exemple:

```
timer MyTimer1 := 5E-3; // déclaration de la temporisation MyTimer1 avec la
                        // valeur par défaut de 5ms

timer MyTimer2;  // déclaration de MyTimer2 sans valeur par défaut de
                 // temporisation c'est-à-dire qu'une valeur doit être
                 // attribuée lorsque le temporisateur est armé
```

Les opérations de temporisation **`start`**, **`stop`**, **`read`** et **`timeout`** peuvent être utilisées afin de commander les temporisations (voir § 23). Par exemple:

```
// Les utilisations de MyTimer2 pourraient être:
MyTimer2.start(10); // 10 s
MyTimer2.start(180); // 3 min
```

11.1 Temporisations utilisées comme paramètres

Les temporisations ne peuvent être transmises par référence qu'à des fonctions et qu'à des variantes nommées. Les temporisations transmises à une fonction ou à une variante nommée sont connues à l'intérieur de la définition comportementale de cette fonction ou de cette variante nommée.

Une temporisation transmise en tant que paramètre de référence peut être utilisée comme toute autre temporisation, c'est-à-dire qu'elle n'a pas besoin d'être déclarée. Une temporisation armée peut également être transmise à une fonction ou à une variante nommée. La temporisation continue son exécution, c'est-à-dire qu'elle n'est pas arrêtée implicitement. D'éventuels événements de fin de temporisation peuvent donc être traités à l'intérieur de la fonction ou de la variante nommée à laquelle la temporisation est transmise.

Exemple:

```
// Définition de fonction avec temporisation dans la liste des paramètres
// formels
function MyBehaviour (timer MyTimer)
{    :
     MyTimer.start;
     :
}
```

12 Déclaration des messages

Un des éléments clés de la notation TTCN-3 est la capacité d'émettre et de recevoir des messages complexes par les accès de communication définis par la configuration de test. Ces messages peuvent être ceux qui sont explicitement orientés vers les essais du système SUT ou vers les messages internes de coordination et de commande spécifiques de la configuration de test choisie.

NOTE – En notation TTCN-2, ces messages sont les primitives de service abstrait (ASP), les unités de données de protocole (PDU) et les messages de coordination. Le langage noyau de la notation TTCN-3 est générique en ce sens qu'il ne formule pas de distinctions syntaxiques ou sémantiques de cette sorte.

Les messages complexes peuvent être définis en tant que types d'enregistrement (voir § 6.3.1). Par exemple:

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

Les messages peuvent, naturellement, être sous-structurés. Par exemple:

```
// Elément d'information de type 1 (IEType1). Déclarations similaires
// pour IEType2 jusqu'à IETypeN
type record IEType1
{
     IEFieldType1  iefield1,
     IEFieldType2  iefield2,
     :
     IEFieldTypeN  iefieldN
}

// Un message contenant des éléments d'information
type record MyMessageType
{
     IEType1  field1,
     IEType2  field2,
     :
     IETypeN field3
}
```

12.1 Champs de message facultatifs

Par défaut, tous les champs d'un message doivent être obligatoires. Les champs de message facultatifs doivent être spécifiés au moyen du mot clé **optional**. Par exemple:

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

13 Déclaration des signatures de procédure

Les signatures de procédures (abrégées en signatures) sont nécessaires pour la communication synchrone. Une procédure peut être soit invoquée dans le système SUT (c'est-à-dire que le système de test effectue l'appel) soit invoquée dans le système de test (c'est-à-dire que le système SUT effectue l'appel).

Pour les procédures appelées par le système SUT comme pour celles qui sont appelées par le système de test, la signature **signature** complète d'une procédure doit être définie dans le module TTCN-3.

A l'intérieur de la définition de signature **signature**, la liste de paramètres peut inclure des identificateurs, des types de paramètre et leur sens (c'est-à-dire **in**, **out** ou **inout**). Noter que le sens de paramètres est celui qui est vu par le correspondant *appelé* plutôt que par le correspondant *appelant*. Par exemple:

```
signature MyRemoteProc (in integer Par1, out float Par2, inout integer Par3)
     return integer;
// Cela définit la procédure distante MyRemoteProc. MyRemoteProc renvoie une
// valeur d'entier et possède trois paramètres: un paramètre in de type
// entier, un paramètre out de type nombre à virgule flottante, et un
// paramètre inout de type entier.
```

Un appel **call** de procédure produira chez l'appelé, soit l'exécution d'une réponse (cas normal) soit le déclenchement d'une exception. Les actions résultant d'un appel de procédure accepté sont définies par le correspondant récepteur (voir § 22).

13.1 Omission de paramètres réels

Il est permis d'omettre des paramètres réels dans une liste paramétrique de signature. On indique cette omission en remplaçant le paramètre réel, à sa position correcte, par le mot clé **omit**. Par exemple:

```
ParameterList(Par1, omit, Par3) // Le paramètre 2 est omis
```

NOTE – Ces omissions sont souvent nécessaires lors de l'utilisation de signatures de procédure en communication synchrone.

13.2 Spécification des exceptions

Les exceptions sont représentées en notation TTCN-3 sous la forme de valeurs d'un type spécifique. Même des modèles et des mécanismes d'appariement peuvent être utilisés.

NOTE – La conversion des exceptions produites par le système SUT vers le type correspondant dépend de l'outil et du système. Elle est donc hors du domaine d'application de la présente Recommandation.

Les exceptions sont définies par l'insertion d'une liste d'exceptions dans la définition de signature. Cette liste définit tous les types pouvant être associés à l'ensemble des exceptions possibles (dont la signification ne sera habituellement distinguée que par sa représentation au moyen de valeurs spécifiques de ces types).

Exemple:

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

// Un appel de la procédure MyRemoteProc peut déclencher des exceptions de
// type ExceptionType1 ou des exceptions de type ExceptionType2
```

14 Déclaration des modèles

Les modèles sont utilisés soit pour transmettre un ensemble de valeurs distinctes soit pour vérifier si un ensemble de valeurs reçues correspond à la spécification du modèle.

Les modèles offrent les possibilités suivantes:

a) ils permettent d'organiser et de réutiliser des données de test, y compris une forme simple d'héritage;

b) ils peuvent être paramétrés;

c) ils permettent de mettre en œuvre des mécanismes d'appariement;

d) ils peuvent être utilisés avec des communications soit en mode message soit en mode procédure.

A l'intérieur des valeurs d'un modèle, des étendues et des attributs d'appariement peuvent être spécifiés puis utilisés aussi bien dans les communications en mode message qu'en mode procédure. Les modèles peuvent être spécifiés pour tout type ou signature de procédure TTCN-3. Les modèles de type servent aux communications en mode message et les modèles de signature servent aux communications en mode procédure.

14.1 Déclaration des modèles de message

Les instances de message comportant des valeurs réelles peuvent être spécifiées au moyen de modèles. Un modèle peut être considéré comme étant un ensemble d'instructions permettant de construire un message d'envoi ou d'apparier un message reçu.

Les modèles peuvent être spécifiés pour tout type TTCN-3 défini dans le Tableau 3, sauf pour les types spéciaux (**port**, **component**, **address**).

```
// Lorsqu'il est utilisé dans une opération de réception ce modèle apparie
// toute valeur d'entier
template integer Mytemplate := *;
// Ce modèle apparie les seules valeurs d'entier 1, 2 ou 3
template integer Mytemplate := (1, 2, 3);
```

Il est cependant prévu qu'ils seront le plus souvent appliqués à des enregistrements, comme indiqué par les exemples des paragraphes suivants.

14.1.1 Modèles d'envoi de messages

Un modèle utilisé dans une opération **send** définit un ensemble complet de valeurs de champ formant le message à transmettre par un accès de test. Au moment de l'opération **send**, le modèle doit être entièrement défini, c'est-à-dire que tous les champs doivent se réduire à des valeurs réelles et qu'aucun mécanisme d'appariement ne doit être utilisé dans les champs du modèle, que ce soit directement ou indirectement.

Exemple:

```
// Si l'on a la définition de message
type record MyMessageType
{
     integer     field1,
     charstring  field2,
     boolean     field3
}
// un modèle de message pourrait être:
template MyMessageType MyTemplate:=
{
     field1 := 1,
     field2 := "My string",
     field3 := true
}
// et une opération d'envoi correspondante pourrait être:
MyPCO.send(MyTemplate);
```

NOTE – Les modèles peuvent également être utilisés pour des exceptions si le type correspondant a été défini.

14.1.2 Modèles de réception de messages

Un modèle utilisé dans une opération de réception **receive** définit un modèle de données auquel un message entrant doit être apparié. Les mécanismes d'appariement, définis dans l'Annexe C, peuvent être utilisés dans les modèles de réception. Aucune association des valeurs entrantes avec le modèle ne doit se produire.

Exemple:

```
// Si l'on a la définition de message
type record MyMessageType
{
     integer     field1,
     charstring  field2,
     boolean     field3
}

// un modèle de message pourrait être
template MyMessageType MyTemplate:=
{
     field1 := 1,
     field2 := pattern "abc*xyz",
     field3 := true
}

// et une opération de réception correspondante pourrait être:
MyPCO.receive(MyTemplate);
```

14.2 Déclaration des modèles de signature

Les instances des listes de paramètres de procédure contenant des valeurs réelles peuvent être spécifiées au moyen de modèles. Ceux-ci peuvent être définis pour toute procédure au moyen d'un référencement de la définition de signature associée.

Exemple:

```
// définition de signature pour une procédure distante
signature RemoteProc(in integer Par1, out integer Par2, inout integer Par3)
     return integer;

// exemple de modèles associés à une signature de procédure définie
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 Modèles d'appel de procédure

Un modèle utilisé dans une opération d'appel **call** ou de réponse **reply** définit un ensemble complet de valeurs de champ pour tous les paramètres de type **in** et **inout**. Au moment de l'opération d'appel call, tous les paramètres **in** et **inout** contenus dans le modèle doivent se réduire à des valeurs réelles et aucun mécanisme d'appariement ne doit être utilisé dans ces champs, directement ou indirectement. Toute spécification de modèle pour paramètres **out** est simplement

négligée. Il est donc permis de spécifier des mécanismes d'appariement pour ces champs ou de les omettre (voir Annexe C).

Exemple:

```
// Appel valide car tous les paramètres in et inout ont une valeur distincte
MyPCO.call(RemoteProc:Template1);

// Appel valide car tous les paramètres in et inout ont une valeur distincte
MyPCO.call(RemoteProc:Template2);

// Appel invalide car les paramètres Par3 ont un attribut d'appariement et
// non une valeur
MyPCO.call(RemoteProc:Template3);

// Les modèles ne renvoient jamais de valeurs. Dans le cas des paramètres
// Par2 et Par3 les valeurs renvoyées par l'appel doivent être extraites au
// moyen d'une clause d'affectation située à la fin de l'instruction d'appel.
```

14.2.2 Modèles d'acceptation des appels de procédure

Un modèle utilisé dans une opération de type **getcall** définit un modèle de données auquel les champs paramétriques entrants sont appariés. Les mécanismes d'appariement, définis dans l'Annexe C, peuvent être utilisés dans les modèles de réception. Aucune association des valeurs entrantes avec le modèle ne doit se produire. Les éventuels paramètres de type **in** doivent être négligés dans le processus d'appariement.

Exemple:

```
// Obtention d'appel valide, qui apparie toute valeur d'entier si Par2 == 2
// et si Par3 == 3
MyPCO.getcall(RemoteProc:Template1);

// Obtention d'appel valide, qui apparie toute valeur d'entier si Par3 == 3
// et toute valeur de Par2
MyPCO.getcall(RemoteProc:Template2);

// Obtention d'appel valide, qui apparie toute valeur d'entier avec toute
// valeur de Par3 et Par2
MyPCO.getcall(RemoteProc:Template3);
```

14.3 Mécanismes d'appariement de modèles

Généralement, les mécanismes d'appariement seront utilisés pour remplacer des valeurs de champs individuels dans un modèle ou même pour remplacer tout le contenu d'un modèle. Certains de ces mécanismes peuvent être utilisés en combinaison.

Les mécanismes d'appariement et les structures génériques peuvent également être utilisés en ligne, mais seulement dans les événements reçus (c'est-à-dire au cours des opérations **receive**, **getcall**, **getreply** et **catch**). Ils peuvent apparaître dans des valeurs explicites. Par exemple:

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

L'identificateur de type est facultatif, par exemple:

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

Le type du modèle en ligne doit cependant être dans la liste des accès par lesquels le modèle est reçu. En cas d'ambiguïté (par exemple, lors d'un sous-typage), le nom du type doit être inclus dans l'instruction de réception.

Les mécanismes d'appariement sont répartis en quatre groupes comme suit:

a) valeurs spécifiques (c'est-à-dire une expression qui s'évalue comme une valeur spécifique);

b) symboles spéciaux qui peuvent être utilisés *à la place* de valeurs:
   - **`(...)`**: une liste de valeurs;
   - **`complement (...)`**: complément d'une liste de valeurs;
   - **`omit`**: la valeur est omise;
   - **`?`**: structure générique représentant une valeur quelconque;
   - **`*`**: structure générique représentant une valeur quelconque ou aucune valeur (c'est-à-dire, une valeur omise);
   - **(**de bas en haut**)**: étendue de valeurs d'entier entre et y compris les limites inférieure et supérieure.

c) symboles spéciaux qui peuvent être utilisés *à l'intérieur de* valeurs:
   - **`?`**: structure générique pour tout élément particulier dans une chaîne, dans une séquence tabulaire, dans un enregistrement (**`record of`**) ou dans un ensemble (**`set of`**);
   - **`*`**: structure générique représentant un nombre quelconque d'éléments consécutifs dans une chaîne, dans une séquence tabulaire, dans un enregistrement (**`record of`**) ou dans un ensemble (**`set of`**) ou ne représentant aucun élément que ce soit (c'est-à-dire qu'un élément est omis).

d) symboles spéciaux qui décrivent des *attributs* de valeurs:
   - **`length`**: restrictions pour chaînes et séquences tabulaires;
   - ifpresent: pour l'appariement de valeurs facultatives de champ (s'il n'y a pas d'omission).

Les mécanismes d'appariement pris en charge et leurs symboles associés (s'ils existent) sont représentés dans le Tableau 5 avec leur portée. Dans ce tableau, la colonne de gauche énumère tous les types équivalents en notation TTCN-3 et ASN.1 comme défini dans la série de Recommandations UIT-T X.680 [7], [8], [9] et [10], auxquels ces mécanismes d'appariement s'appliquent. L'on trouvera en Annexe C une description complète de chaque mécanisme d'appariement.

Tableau 5/Z.140 – Mécanismes TTCN-3 d'appariement

Mécanisme utilisé avec valeurs de	**Valeur**	**Au lieu de valeurs**								**Attributs**	
	Valeur spécifique	**Liste de valeurs**	**Liste complémentée**	**Valeur omise**	**Toute valeur (?)**	**Toute valeur ou aucune valeur (*)**	**Etendue**	**Tout élément (?)**	**Tout élément ou aucun élément (*)**	**Restriction de longueur**	**Si présent**
`boolean`	Oui	Oui	Oui	Oui	Oui	Oui					Oui
`integer`	Oui	Oui	Oui	Oui	Oui	Oui	Oui				Oui
`char`	Oui	Oui	Oui	Oui	Oui	Oui	Oui				Oui
`universal char`	Oui	Oui	Oui	Oui	Oui	Oui	Oui				Oui
`float`	Oui	Oui	Oui	Oui	Oui	Oui					Oui
`bitstring`	Oui	Oui	Oui	Oui	Oui	Oui		Oui	Oui	Oui	Oui
`octetstring`	Oui	Oui	Oui	Oui	Oui	Oui		Oui	Oui	Oui	Oui

Tableau 5/Z.140 – Mécanismes TTCN-3 d'appariement

Mécanisme utilisé avec valeurs de	Valeur	Au lieu de valeurs								Attributs	
	Valeur spécifique	Liste de valeurs	Liste complémentée	Valeur omise	Toute valeur (?)	Toute valeur ou aucune valeur (*)	Etendue	Tout élément (?)	Tout élément ou aucun élément (*)	Restriction de longueur	Si présent
hexstring	Oui	Oui	Oui	Oui	Oui	Oui		Oui	Oui	Oui	Oui
characterstring	Oui	Oui	Oui	Oui	Oui	Oui		Oui	Oui	Oui	Oui
record	Oui	Oui	Oui	Oui	Oui	Oui					Oui
record of	Oui	Oui	Oui	Oui	Oui	Oui		Oui	Oui	Oui	Oui
array	Oui	Oui	Oui	Oui	Oui	Oui		Oui	Oui	Oui	Oui
set	Oui	Oui	Oui	Oui	Oui	Oui					Oui
set of	Oui	Oui	Oui	Oui	Oui	Oui		Oui	Oui	Oui	Oui
enumerated	Oui	Oui	Oui	Oui	Oui	Oui					Oui
union	Oui	Oui	Oui	Oui	Oui	Oui					Oui

14.4 Paramétrisation des modèles

Les modèles relatifs aux opérations d'expédition comme de réception peuvent être paramétrés. Les paramètres formels d'un modèle peuvent comprendre des modèles, des fonctions et les symboles spéciaux d'appariement. Les règles applicables aux listes de paramètres formels et réels doivent être suivies comme défini au § 5.3.

Exemple:

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

// pourrait être utilisé comme suit
pco1.send(MyTemplate(123));
```

14.4.1 Paramétrisation avec attributs d'appariement

Afin de permettre la transmission d'attributs d'appariement en tant que paramètres, le mot clé supplémentaire **template** doit être ajouté avant le champ de type, ce qui transforme le paramètre en modèle et élargit pratiquement les paramètres permis pour le type associé de façon qu'ils incluent l'ensemble approprié d'attributs d'appariement (voir Annexe C) ainsi que l'ensemble normal de valeurs. Les champs des paramètres de modèle ne doivent pas être appelés par référence.

Exemple:

```
// Le modèle
template MyMessageType MyTemplate (template integer MyFormalParam):=
{    field1 := MyFormalParam,
     field2 := pattern "abc*xyz",
```

```
    field3 := true
}
// pourrait être utilisé comme suit:
pco1.receive(MyTemplate(?));
// Ou, si le champ field1 a été défini comme étant facultatif:
pco1.receive(MyTemplate(omit));
```

14.5 Paramétrisation des modèles

Seules les définitions des types **function**, **testcase**, **named alt** et **template** peuvent avoir des modèles en tant que paramètres formels.

Exemple:

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

14.6 Modèles modifiés

Normalement, un modèle spécifiera un ensemble de valeurs ou de symboles d'appariement de base ou par défaut pour chacun des champs définis dans la définition appropriée. Si de légères modifications sont nécessaires pour spécifier un nouveau modèle, il est possible de spécifier un modèle modifié. Celui-ci spécifie des modifications relatives à des champs particuliers du modèle originel, soit directement soit indirectement.

Le mot clé **modifies** indique le modèle supérieur dont le modèle nouveau ou modifié doit être issu. Ce modèle supérieur peut être soit le modèle original soit un modèle modifié.

Les modifications se produisent de façon liée avec retour obligé au modèle original. Si un champ de modèle est spécifié dans le modèle modifié avec sa valeur correspondante ou son symbole d'appariement correspondant, cette valeur ou ce symbole remplace celle ou celui qui est spécifié(e) dans le modèle supérieur. Si un champ de modèle n'est pas spécifié dans le modèle modifié avec sa valeur correspondante ou son symbole d'appariement correspondant, l'on doit utiliser la valeur ou le symbole d'appariement figurant dans le modèle supérieur.

Un modèle modifié ne doit pas faire référence à lui-même, soit directement soit indirectement. Autrement dit, les dérivations récursives ne sont pas autorisées.

Exemple:

```
// Si l'on a
template MyRecordType MyTemplate1 :=
{
     field1 := 123,
     field2 := "une chaîne",
     field3 := true
}

// alors le fait d'écrire
template MyRecordType MyTemplate2 modifies MyTemplate1 :=
{
     field2 := "Une chaîne modifiée",
     field3 := omit     // le champ 3 doit être spécifié comme étant facultatif
                        // dans le type d'enregistrement correspondant
}
```

```
// est équivalent au fait d'écrire
template MyRecordType MyTemplate2 :=
{
     field1 := 123,
     field2 := "Une chaîne modifiée",
     field3 := omit
}
```

14.6.1 Paramétrisation des modèles modifiés

Si un modèle de base possède une liste de paramètres formels, les règles suivantes s'appliquent à tous les modèles modifiés issus de ce modèle de base, qu'ils en dérivent ou non par une ou par plusieurs étapes de modification:

a) le modèle dérivé ne doit pas omettre de paramètres mais peut avoir des paramètres supplémentaires (ajoutés), le cas échéant;

b) la liste de paramètres formels doit suivre le nom du modèle dans chaque modèle modifié;

c) les modèles paramétrés contenus dans les champs de modèle ne doivent pas être modifiés ni être explicitement omis dans un modèle modifié.

Exemple:

```
// Si l'on a
template MyRecordType MyTemplate1(integer Mypar):=
{
     field1 := MyPar,
     field2 := "une chaîne",
     field3 := true
}

// alors une modification pourrait être
     template MyRecordType MyTemplate2(integer MyPar) modifies MyTemplate1 :=
{    // le champ 1 est paramétré dans le modèle 1
     field2 := "Une chaîne modifiée",
     field3 := omit     // le champ 3 doit être spécifié comme étant facultatif
                        // dans le type correspondant d'enregistrement
}
```

14.6.2 Modèles modifiés en ligne

La notation TTCN-3 permet, en plus de la création de contraintes modifiées explicitement nommées, la définition de contraintes modifiées en ligne.

Exemple:

```
// Si l'on a
template MyMessageType Setup :=
{    field1 := 75,
     field2 := "abc",
     field3 := true
}

// Pourrait être utuilisé pour définir un modèle modifié en ligne d'opération
// Setup
pco1.send (modifies Setup := {field1 76});
```

14.7 Modification des champs de modèle

Toutes les modifications apportées aux champs de modèle ne doivent être effectuées que par paramétrisation ou par modèles dérivés en ligne au moment de l'exécution d'une opération de communication (par exemple, **`send`**, **`receive`**, **`call`**, **`getcall`**, etc.). Les effets de ces modifications

sur la valeur du champ de modèle ne persistent pas dans le modèle qui fait suite à l'événement de communication correspondant.

La notation de la sorte *MyTemplateId.Fieldid* ne doit pas être utilisée pour insérer ou extraire des valeurs dans des modèles lors d'événements de communication. Le symbole "**->**" doit être utilisé à cette fin (voir § 22).

14.8 Opération d'appariement

L'opération **match** permet de comparer la valeur d'une variable à un modèle. Cette opération renvoie une valeur booléenne. Si le type du modèle et la variable ne sont pas compatibles, l'opération renvoie l'opérateur *faux*. Si les types sont compatibles, la valeur de retour de l'opération indique si la valeur de la variable est conforme au modèle spécifié.

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

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

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

14.9 Valeur d'opération

L'opération **valueof** permet d'affecter la valeur spécifiée dans un modèle aux champs d'une variable. Celle-ci et le modèle doivent être de types compatibles (voir § 6.7) et chaque champ du modèle doit se réduire à une seule valeur.

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

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

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

15 Opérateurs

La notation TTCN-3 prend en charge un certain nombre d'opérateurs prédéfinis qui peuvent être utilisés dans les termes d'expressions TTCN-3. Ces opérateurs prédéfinis se répartissent en sept catégories comme suit:

a) opérateurs arithmétiques;

b) opérateurs de chaîne;

c) opérateurs relationnels;

d) opérateurs logiques;

e) opérateurs binaires;

f) opérateurs de décalage;

g) opérateurs de rotation.

Ces opérateurs sont énumérés dans le Tableau 6.

Tableau 6/Z.140 – Liste des opérateurs TTCN-3

Catégorie	Opérateur	Symbole ou mot clé
Opérateurs arithmétiques	addition	`+`
	soustraction	`-`
	multiplication	`*`
	division	`/`
	modulo	`mod`
	reste	`rem`
Opérateurs de chaîne	concaténation	`&`
Opérateurs relationnels	égal à	`==`
	inférieur à	`<`
	supérieur à	`>`
	inégal à	`!=`
	supérieur à ou égal à	`>=`
	inférieur à ou égal à	`<=`
Opérateurs logiques	non logique	`not`
	et logique	`and`
	ou logique	`or`
	oux logique	`xor`
Opérateurs binaires	non binaire	`not4b`
	et binaire	`and4b`
	ou binaire	`or4b`
	oux binaire	`xor4b`
Opérateurs de décalage	décalage à gauche	`<<`
	décalage à droite	`>>`
Opérateurs de rotation	rotation à gauche	`<@`
	rotation à droite	`@>`

Les priorités de ces opérateurs sont représentées dans le Tableau 7. Dans chaque rangée de ce tableau, les opérateurs énumérés ont une priorité égale. Si plusieurs opérateurs de priorité égale apparaissent dans une expression, les opérations sont évaluées de gauche à droite. Les parenthèses peuvent être utilisées pour grouper des opérandes dans des expressions: dans ce cas, une expression entre parenthèses a la priorité la plus élevée lors de l'évaluation.

Tableau 7/Z.140 – Priorité des opérateurs

Priorité	Type d'opérateur	Opérateur
Supérieure		`( ... )`
	Unaire	`+`, `–`, **`not`**, **`not4b`**
	Binaire	`*`, `/`, **`mod`**, **`rem`**
		`+`, `–`
		`<<`, `>>`, `<@`, `@>`
		`<`, `>`, `<=`, `>=`
		`==`, `!=`
		`and4b`
		`xor4b`
		`or4b`
		`and`
		`xor`
		`or`
Inférieure		`&`

15.1 Opérateurs arithmétiques

Les opérateurs arithmétiques représentent les opérations d'addition, de soustraction, de multiplication, de division et modulo. Les opérandes de ces opérateurs doivent être du type **`integer`** (y compris les dérivés de ce type) ou **`float`** (y compris les dérivés de ce type), sauf pour l'opérateur **`mod`**, qui ne doit être utilisé qu'avec des types **`integer`** (y compris ses dérivés).

Avec les types **`integer`**, les résultats des opérations arithmétiques sont du type **`integer`**. Avec les types **`float`**, les résultats des opérations arithmétiques sont du type **`float`**.

Si le signe plus (+) ou moins (–) est utilisé comme opérateur unaire, les règles des opérandes s'appliquent également. Le résultat de l'utilisation de l'opérateur moins est la valeur négative de l'opérande si celui-ci était positif et inversement.

Le résultat de l'exécution de l'opération de division (/) sur deux entiers est le suivant:

a) les valeurs de type **`integer`** donnent la valeur **`integer`** entière qui résulte de la division du premier entier par le second (c'est-à-dire que les valeurs fractionnaires sont rejetées);

b) les valeurs de type **`float`** donnent la valeur **`float`** qui résulte de la division du premier nombre en virgule flottante par le second (c'est-à-dire que les valeurs fractionnaires ne sont pas rejetées).

Les opérateurs **`rem`** et **`mod`** s'appliquent à des opérandes de type **`integer`** et donnent un résultat de type **`integer`**. Les opérations `x` **`rem`** `y` et `x` **`mod`** `y` calculent le reste qui est issu d'une division de l'entier `x` par l'entier `y`. Ces opérateurs ne sont donc définis que pour les opérandes `y` différents de zéro. Pour les entiers `x` et `y` positifs, les deux opérations `x` **`rem`** `y` et `x` **`mod`** `y` donnent le même résultat, mais si ces arguments sont négatifs, les résultats sont différents.

Formellement, les opérateurs **`mod`** et **`rem`** sont définis comme suit:

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

Le Tableau 8 illustre la différence entre les opérateurs **`mod`** et **`rem`**.

Tableau 8/Z.140 – Effet des opérateurs mod et 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 Opérateurs de chaîne

Ces opérateurs relationnels prédéfinis effectuent la concaténation des types chaîne. Les opérandes peuvent être de quelconques valeurs de type chaîne qui sont compatibles. L'opération est une simple concaténation de gauche à droite. Aucune forme d'addition arithmétique n'est impliquée. Le résultat est de type chaîne compatible. Par exemple:

```
'1111'B & '0000'B & '1111'B donne '111100001111'B
```

15.3 Opérateurs relationnels

Les opérateurs relationnels prédéfinis représentent les relations d'égalité, d'infériorité, de supériorité, d'inégalité, de supériorité ou égalité, et d'infériorité ou égalité. Les opérandes d'égalité (**==**) et d'inégalité (**!=**) peuvent être d'un type arbitraire. Tous les autres opérateurs relationnels doivent avoir des opérandes du seul type **integer** (y compris ses dérivés) ou **float** (y compris ses dérivés). Dans tous les cas, les deux opérandes doivent être de types compatibles. Le résultat de ces opération est de type **boolean**.

15.4 Opérateurs logiques

Les opérateurs booléens **boolean** prédéfinis effectuent les opérations de négation, de ET **and** logique, de OU **or** logique et de **xor** OUX logique. Leurs opérandes doivent être de type **boolean**. Le résultat des opérateurs logiques est de type booléen.

Le NON **net** logique est l'opérateur unaire qui renvoie la valeur **true** si son opérande est faux et qui renvoie la valeur **false** si son opérande est vrai.

Le ET **and** logique renvoie la valeur **true** si ses deux opérandes sont vrais; sinon, il renvoie la valeur **false**.

Le OU **or** logique renvoie la valeur **true** si au moins un de ses opérandes est vrai; il renvoie la valeur **false** seulement si ses opérandes sont tous les deux faux ou vrais.

Le OUX **xor** logique renvoie la valeur **true** si un de ses opérandes est vrai; il renvoie la valeur **false** si ses opérandes sont tous les deux faux ou vrais.

15.5 Opérateurs binaires

Les opérateurs binaires prédéfinis effectuent les opérations NON, ET, OU et OUX au niveau binaire. Ces opérateurs sont respectivement dénotés par **not4b, and4b, or4b** et **xor4b**.

NOTE – Ces notations représentent les mots anglais "not for bit" (non pour bit), "and for bit" (et pour bit), etc.

Leurs opérandes doivent être de type **bitstring, hexstring, octetstring**. Le résultat des opérateurs binaires doit être du même type que celui des opérandes.

L'opérateur unaire **not4b** inverse, au niveau des bits, les valeurs binaires individuelles de son opérande comme suit: chaque bit 1 de l'opérande est mis à 0 et chaque bit 0 est mis à 1. Autrement dit:

```
not4b  '1'B  donne  '0'B
not4b  '0'B  donne  '1'B
```

Exemple:

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

L'opérateur binaire **and4b** accepte deux opérandes. Aux positions binaires correspondantes, la valeur résultante est 1 si les deux bits sont mis à 1; sinon, la valeur du bit résultant est 0. Autrement dit:

```
'1'B and4b '1'B donne  '1'B
'1'B and4b '0'B donne  '0'B
'0'B and4b '1'B donne  '0'B
'0'B and4b '0'B donne  '0'B
```

Exemple:

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

L'opérateur binaire **or4b** accepte deux opérandes. Aux positions binaires correspondantes, la valeur résultante est 0 si les deux bits sont mis à 0; sinon, la valeur du bit résultant est 1. Autrement dit:

```
'1'B or4b '1'B donne  '1'B
'1'B or4b '0'B donne  '1'B
'0'B or4b '1'B donne  '1'B
'0'B or4b '0'B donne  '0'B
```

Exemple:

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

L'opérateur binaire **xor4b** accepte deux opérandes. Aux positions binaires correspondantes, la valeur résultante est 0 si les deux bits sont mis à 0 ou si les deux bits sont mis à 1; sinon, la valeur du bit résultant est 0. Autrement dit:

```
'1'B xor4b '1'B donne  '0'B
'0'B xor4b '0'B donne  '0'B
'0'B xor4b '1'B donne  '1'B
'1'B xor4b '0'B donne  '1'B
```

Exemple:

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

15.6 Opérateurs de décalage

Les opérateurs de décalage prédéfinis effectuent l'opération de décalage à gauche (**<<**) et de décalage à droite (**>>**). Leur opérande de gauche doit être de type **bitstring, hexstring, octetstring** ou **integer**. Leur opérande de droite doit être du type **integer**. Le résultat de ces opérateurs doit être du même type que celui de l'opérande de gauche.

Les opérateurs de décalage ont un comportement différent selon le type de leur opérande de gauche, comme suit:

a) **bitstring** ou **integer** : l'unité de décalage appliquée est 1 bit;

b) **hexstring**: l'unité de décalage appliquée est 1 chiffre hexadécimal;

c) **octetstring**: l'unité de décalage appliquée est 1 octet.

L'opérateur de décalage à gauche (**<<**) accepte deux opérandes. Il décale l'opérande de gauche du nombre d'unités de décalage vers la gauche qui est spécifié par l'opérande de droite. Les unités de décalage en excès (bits, chiffres hexadécimaux ou octets) sont rejetées. Pour chaque unité de décalage effectué vers la droite, un zéro est inséré à droite de l'opérande de gauche (`'0'B`, `'0'H` ou `'00'O` selon le type de l'opérande de gauche).

NOTE 1 – Si l'opérande de gauche est du type **integer**, chaque décalage de bit vers la gauche équivaut à une multiplication par deux de l'opérande de gauche.

NOTE 2 – Un verdict d'erreur doit être rendu si un débordement dû au système se produit lors de l'application de l'opération de décalage à gauche à l'opérande de gauche.

Exemple:

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

L'opérateur de décalage à droite (**>>**) accepte deux opérandes. Il décale l'opérande de gauche du nombre d'unités de décalage vers la droite qui est spécifié par l'opérande de droite. Les unités de décalage en excès (bits, chiffres hexadécimaux ou octets) sont rejetées. Pour chaque unité de décalage effectué vers la droite, un zéro est inséré à gauche de l'opérande de gauche (`'0'B`, `'0'H` ou `'00'O` selon le type de l'opérande de gauche).

NOTE 3 – Si l'opérande de gauche est du type **integer**, chaque décalage de bit vers la droite équivaut à une division d'entier par deux (2) de l'opérande de gauche.

NOTE 4 – Si l'opérande de gauche est de type **integer** et que sa valeur soit négative, le bit de signe doit être propagé lors d'un décalage vers la droite.

Exemple:

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

15.7 Opérateurs de rotation

Les opérateurs de rotation prédéfinis effectuent l'opération de décalage à gauche (**<@**) et de décalage à droite (**@>**). Leur opérande de gauche doit être de type **bitstring, hexstring, octetstring, charstring** ou **universal charstring**. Leur opérande de droite doit être du type **integer**. Le résultat de ces opérateurs doit être du même type que celui de l'opérande de gauche.

Les opérateurs de rotation ont un comportement différent selon le type de leur opérande de gauche, comme suit:

a) **bitstring** : l'unité de rotation appliquée est 1 bit;

b) **hexstring**: l'unité de rotation appliquée est 1 chiffre hexadécimal;

c) **octetstring**: l'unité de rotation appliquée est 1 octet;

d) **charstring** ou **universal charstring**: l'unité de rotation appliquée est 1 caractère.

L'opérateur de rotation à gauche (**<@**) accepte deux opérandes. Il fait tourner l'opérande de gauche du nombre d'unités de rotation vers la gauche qui est spécifié par l'opérande de droite. Les unités de rotation en excès (bits, chiffres hexadécimaux, octets ou caractères) sont réinsérées dans l'opérande de gauche à partir de sa droite.

Exemple:

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

L'opérateur de rotation à droite (**@>**) accepte deux opérandes. Il fait tourner l'opérande de gauche du nombre d'unités de rotation vers la droite qui est spécifié par l'opérande de droite. Les unités de rotation en excès (bits, chiffres hexadécimaux, octets ou caractères) sont réinsérées dans l'opérande de gauche à partir de sa gauche.

Exemple:

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

16 Fonctions

Les fonctions sont utilisées en notation TTCN-3 pour exprimer un comportement de test ou pour structurer des calculs dans un module, par exemple afin de calculer une certaine valeur, pour initialiser un ensemble de variables ou pour vérifier une condition. Les fonctions peuvent renvoyer une valeur, ce qui est indiqué par le mot clé **return** suivi d'un identificateur de type. Si aucun mot clé **return** n'est spécifié, la fonction est nulle. En notation TTCN-3, il n'existe pas de mot clé explicite pour désigner la nullité. Le mot clé **return**, lorsqu'il est utilisé dans le corps d'une fonction, provoque la terminaison de celle-ci et le retour d'une valeur compatible avec le type de retour. Par exemple:

```
// Définition de la fonction MyFunction qui ne possède aucun paramètre
function MyFunction() return integer
{

     return 7;   // Renvoie la valeur d'entier 7 lorsque la fonction se
                 // termine
}
```

NOTE – Les fonctions TTCN-3 remplacent les opérations de suite de tests et les définitions de procédure de suite de tests contenues dans la notation TTCN-2. Des fonctions informelles peuvent être déclarées en tant que fonctions externes avec des commentaires explicatifs ou au moyen d'une fonction formelle vide assortie de commentaires.

Une fonction peut être définie à l'intérieur d'un module ou être déclarée comme étant définie de manière externe (c'est-à-dire de type **external**). Pour une fonction externe, seule l'interface fonctionnelle doit être fournie dans le module TTCN-3. La réalisation de la fonction externe est hors du domaine d'application de la présente Recommandation. Les fonctions externes ne sont pas autorisées à contenir des opérations sur accès.

```
external function MyFunction4() return integer;// Fonction externe sans
                                               // paramètres qui renvoie une
                                               // valeur d'entier

external function InitTestDevices();  // Fonction externe qui n'a d'effet
                                      // qu'à l'extérieur du module TTCN-3
```

Dans un module, le comportement d'une fonction peut être défini au moyen des instructions de programmation et des opérations définies dans le § 18. Si une fonction comporte des opérations sur accès, le type de composant associé doit être référencé au moyen des mots clés **runs on** dans l'en-tête de fonction afin de définir le nombre, le type et les identificateurs des accès disponibles. La

seule exception à cette règle est le cas où tous les accès utilisés dans la fonction sont transmis en paramètres.

Si une fonction comporte des opérations sur accès, soit tous les accès utilisés dans cette fonction doivent être transmis en paramètres soit un type de composant associé doit être référencé au moyen des mots clés **runs on** dans l'en-tête de la fonction afin de définir le nombre, le type et les identificateurs des accès disponibles. Par exemple:

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

Les instances de différents types de composant peuvent utiliser la même fonction si elles répondent à la règle de cohérence suivante:

"Soit C1 et C2 deux types de composant et FUNC une fonction qui fait référence à C1 dans sa clause ***runs on****. Une instance du type de composant C2 peut utiliser le composant FUNC si la définition du type C2 contient l'ensemble de la définition du type C1. En d'autres termes, le composant C2 contient les mêmes noms que C1 pour adresser les accès du même type."*

16.1 Paramétrisation des fonctions

Les fonctions peuvent être paramétrées. Les règles de la liste de paramètres formels doivent être suivies comme défini dans le § 5.3. Par exemple:

```
function MyFunction2(inout integer MyPar1)
{
                              // MyFunction2 ne renvoie pas de valeur
     MyPar1 := 10 * MyPar1;   // mais change la valeur de MyPar1 qui est
                              // transmise par référence
}

function MyFunction3() runs on MyPTCType
{
                                    // MyFunction3 ne renvoie pas de valeur, mais
     var integer MyVar := 5;        // fait effectivement usage de l'opération sur
     PCO1.send(MyVar);              // accès d'envoi et nécessite donc une clause
                                    // d'exécution pour résoudre les
                                    // identificateurs d'accès par référencement
                                    // d'un type de composant
)
```

16.2 Invocation des fonctions

Une fonction est invoquée par une référence à son nom et par la liste paramétrique réelle. Les fonctions qui ne renvoient pas de valeurs peuvent être invoquées directement. Les fonctions qui renvoient des valeurs peuvent être invoquées à l'intérieur d'expressions. Les règles applicables aux listes paramétriques réelles doivent être observées comme défini au § 5.3.

```
MyVar := MyFunction4(); // La valeur renvoyée par MyFunction4 est attribuée à
                        // MyVar. Les types de la valeur renvoyée et MyVar
                        // doivent être identiques

MyFunction2(MyVar2);    // MyFunction2 ne renvoie pas de valeur et est appelée
                        // avec le paramètre réel MyVar2, qui peut être
                        // transmis par référence

MyVar3 := MyFunction6(4)+ MyFunction7(MyVar3); // Fonctions utilisées dans
                                               // expressions
```

Des restrictions particulières s'appliquent aux fonctions liées à des composants utilisant l'opération **start**. Ces restrictions sont décrites au § 21.5.

16.3 Fonctions prédéfinies

La notation TTCN-3 contient un certain nombre de fonctions prédéfinies (intégrées) (voir Tableau 9) qu'il n'est pas nécessaire de déclarer avant emploi.

Tableau 9/Z.140 – Liste des fonctions TTCN-3 prédéfinies

Catégorie	Fonction	Mot clé
Fonctions de conversion	Convertit une valeur **integer** en valeur **char**	**int2char**
	Convertit une valeur **char** en valeur **int** valeur	**char2int**
	Convertit une valeur **integer** en valeur **universal char**	**int2unichar**
	Convertit une valeur **universal char** en valeur **int**	**unichar2int**
	Convertit une valeur **bitstring** en valeur **integer**	**bit2int**
	Convertit une valeur **hexstring** en valeur **integer**	**hex2int**
	Convertit une valeur **octetstring** en valeur **Integer**	**oct2int**
	Convertit une valeur **charstring** en valeur **integer**	**str2int**
	Convertit une valeur **integer** en valeur **bitstring**	**int2bit**
	Convertit une valeur **integer** en valeur **hexstring**	**int2hex**
	Convertit une valeur **integer** en valeur **octetstring**	**int2oct**
	Convertit une valeur **integer** en valeur **charstring**	**int2str**
Fonctions de longueur/taille	Renvoie la longueur d'une valeur de tout type chaîne	**lengthof**
	Renvoie le nombre d'éléments contenus dans un type **record**, **record of**, **template**, **set**, **set of** ou **array**	**sizeof**
Fonctions de présence/choix	Détermine si un champ facultatif est présent dans un type **record**, **record of**, **template**, **set** ou **set of**	**ispresent**
	Détermine quel choix a été fait dans un type **union**	**ischosen**

Lorsqu'une fonction prédéfinie est invoquée:

1) le nombre de paramètres réels doit être égal au nombre de paramètres formels;
2) chaque paramètre réel doit se réduire à un élément de son type de paramètre formel correspondant;
3) toutes les variables apparaissant dans la liste paramétrique doivent être liées.

La description complète des fonctions prédéfinies est donnée dans l'Annexe D.

17 Tests élémentaires

Les tests élémentaires sont une sorte spéciale de fonction. Leur exécution dans la partie commande d'un module est associée à l'instruction **execute** (voir § 26.1). Le résultat de l'exécution d'un test élémentaire est toujours une valeur de type **verdicttype**. Chaque test élémentaire doit contenir un composant MTC et un seul du type qui est cité en référence dans l'en-tête de la définition du test élémentaire. Le comportement défini dans le corps d'un test élémentaire est celui du composant MTC.

Lorsqu'un test élémentaire est invoqué, les accès de l'interface avec le système de test sont instanciées, le composant MTC est créé et le comportement spécifié dans la définition du test élémentaire est lancé dans le MTC. Toutes ces actions doivent être accomplies implicitement, c'est-à-dire sans les opérations explicites **create** et **start**.

Une définition de test élémentaire comporte les deux parties suivantes afin de fournir les informations permettant que ces opérations implicites soient accomplies:

a) partie interface (obligatoire): indiquée par les mots clés **runs on** qui font référence au type de composant requis pour le MTC et qui rend les noms d'accès associés visibles dans le comportement du MTC;

b) et la partie système de test (facultative): indiquée par le mot clé **system** qui fait référence au type de composant définissant les accès nécessaires à l'interface avec le système de test. La partie système de test ne doit être omise que si, au cours de l'exécution du test, seul le composant MTC est instancié. Dans ce cas, le type de composant MTC définit explicitement les accès d'interface avec le système de test.

Exemple:

```
testcase MyTestCaseOne()
runs on MyMtcType1           // Définit le type du MTC
system  MyTestSystemType     // Rend les noms d'accès de l'interface TSI
                             // visibles par le MTC
{
     : // Le comportement ici défini s'exécute sur le composant MTC lorsque
       // le test élémentaire est invoqué
}

// ou lors d'un test élémentaire où seul le composant MTC est instancié
testcase MyTestCaseTwo() runs on MyMtcType2
{
     : // Le comportement ici défini s'exécute sur le composant MTC lorsque
       // le test élémentaire est invoqué
}
```

18 Instructions et opérations de programmation

Les éléments fondamentaux de programmation dans la partie commande des modules et fonctions TTCN-3 sont des instructions de programmation de base telles que des expressions, des affectations, des structures d'itération et analogues, des instructions comportementales comme le comportement séquentiel, le comportement à variantes, l'entrelacement, les valeurs par défaut, etc., ainsi que des opérations comme **send, receive, create**, etc.

Les instructions peuvent être soit simples (sans inclure d'autres instructions de programmation) soit composites (pouvant inclure d'autres instructions).

Les instructions peuvent être groupées en blocs. Les blocs d'instructions peuvent être utilisés dans différentes unités de portée, c'est-à-dire la partie commande d'un module, les fonctions et les comportements de test. La sorte des instructions pouvant être utilisées dans un bloc dépendra de l'unité de portée dans laquelle ce bloc sera utilisé. Par exemple, un bloc d'instructions apparaissant dans une fonction ne doit utiliser que les instructions de programmation qui peuvent être utilisées dans des fonctions.

Les règles générales de portée sont décrites dans le § 5.4.

Un bloc d'instructions est syntaxiquement équivalent à une instruction simple. Un bloc peut donc apparaître chaque fois qu'une instruction est autorisée dans une fonction, ce qui implique que les blocs peuvent être imbriqués. Les déclarations éventuelles doivent être formulées au début du bloc. Ces déclarations ne sont visibles qu'à l'intérieur du bloc et pour les sous-blocs imbriqués.

Les déclarations d'un bloc doivent être exécutées dans l'ordre de leur apparition. La spécification d'un bloc d'instructions vide (soit {}) est autorisée. Un bloc d'instructions vide implique qu'aucune action n'est entreprise.

Tableau 10 /Z.140 – Aperçu général des instructions et opérations TTCN-3

Instruction	Mot clé ou symbole associé	Peut être utilisée dans commande de module	Peut être utilisée dans fonctions, tests élémentaires et variantes nommées
Instructions de programmation de base			
Expressions	`(...)`	Oui	Oui
Affectations	`:=`	Oui	Oui
Journalisation	`log`	Oui	Oui
Etiqueter et aller à	`label` / `goto`	Oui	Oui
Si-sinon	`if (...) { ... } else { ... }`	Oui	Oui
Boucle pour	`for (...) { ... }`	Oui	Oui
Boucle tant que	`while (...) { ... }`	Oui	Oui
Boucle d'exécution tant que	`do { ... } while (...)`	Oui	Oui
Exécution d'un arrêt	`stop`	Oui	Oui
Instructions de programmation comportementales			
Comportement à variantes	`alt { ... }`	Oui (Note 1)	Oui
Variante nommée	`named alt { ... }`	Oui (Note 1)	Oui
Comportement entrelacé	`interleave { ... }`	Oui (Note 1)	Oui
Activation d'un défaut	`activate`	Oui (Note 1)	Oui
Désactivation d'un défaut	`deactivate`	Oui (Note 1)	Oui
Commande de retour	`return`		Oui
Opérations de configuration			
Création d'un composant de test parallèle	`create`		Oui
Connexion d'un composant à un composant	`connect`		Oui
Déconnexion de deux composants	`disconnect`		Oui
Mappage d'un accès vers interface de test	`map`		Oui
Démappage d'un accès à partir de l'interface du système de test	`unmap`		Oui
Obtention d'adresse de composant MTC	`mtc`		Oui
Obtention d'adresse d'interface de système de test	`system`		Oui
Obtention d'adresse propre	`self`		Oui

Tableau 10 /Z.140 – Aperçu général des instructions et opérations TTCN-3

Instruction	Mot clé ou symbole associé	Peut être utilisée dans commande de module	Peut être utilisée dans fonctions, tests élémentaires et variantes nommées
Début d'exécution d'un composant de test	`start`		Oui
Exécution d'un arrêt d'un composant de test	`stop`		Oui
Contrôle de terminaison d'un composant PTC	`running`		Oui
Attente de terminaison d'un composant PTC	`done`		Oui
Opérations de communication			
Envoi de message	`send`		Oui
Invocation d'appel de procédure	`call`		Oui
Réponse à appel de procédure issu d'entité distante	`reply`		Oui
Déclenchement d'exception (à un appel accepté)	`raise`		Oui
Réception de message	`receive`		Oui
Déclenchement sur message	`trigger`		Oui
Acceptation d'appel de procédure issu d'entité distante	`getcall`		Oui
Obtention de réponse issue d'un appel précédent	`getreply`		Oui
Acquisition d'exception (issue d'entité appelée)	`catch`		Oui
Contrôle de message (actuel)/appel reçu	`check`		Oui
Libérer accès	`clear`		Oui
Libérer et utiliser l'accès	`start`		Oui
Arrêt d'utilisation d'un accès (réception & émission)	`stop`		Oui
Opérations de temporisation			
Armement de temporisateur	`start`	Oui	Oui
Arrêt de temporisateur	`stop`	Oui	Oui
Lecture de la durée écoulée	`read`	Oui	Oui
Vérification d'armement de temporisateur	`running`	Oui	Oui
Evénement de temporisation	`timeout`	Oui	Oui

Tableau 10 /Z.140 – Aperçu général des instructions et opérations TTCN-3

Instruction	Mot clé ou symbole associé	Peut être utilisée dans commande de module	Peut être utilisée dans fonctions, tests élémentaires et variantes nommées
Opérations de verdict			
Définition du verdict local	`verdict.set`		Oui
Obtention du verdict local	`verdict.get`		Oui
Opérations de système SUT			
Action distante que doit effectuer le SUT	`sut.action`		Oui
Exécution de tests élémentaires			
Exécution d'un test élémentaire	`execute`	Oui	Oui (Note 2)
NOTE 1 – Ne peut être utilisée que dans une commande avec opérations de temporisation. NOTE 2 – Ne peut être utilisée que dans des fonctions et des variantes nommées qui sont utilisées dans une commande de module.			

19 Instructions de programmation de base

Les instructions de programmation de base sont des expressions, des affectations, des opérations, des structures en boucle, etc. Toutes les instructions de programmation de base peuvent être utilisées dans la partie commande d'un module et dans les fonctions TTCN-3.

Tableau 11/Z.140 – Aperçu général des instructions de programmation de base TTCN-3

Instructions de programmation de base	
Instruction	**Mot clé ou symbole associé**
Expressions	`( ... )`
Affectations	`:=`
Journalisation	`log`
Etiqueter et aller à	`label / goto`
Si-sinon	`if ( ... ) { ... } else { ... }`
Boucle pour	`for ( ... ) { ... }`
Boucle tant que	`while ( ... ) { ... }`
Boucle d'exécution tant que	`do { ... } while ( ... )`
Exécution d'un arrêt	`stop`

19.1 Expressions

La notation TTCN-3 permet de spécifier des expressions au moyen des opérateurs définis au § 15. Les expressions sont construites à partir d'autres expressions (simples). Elles peuvent contenir des fonctions. Leur résultat doit être la valeur d'un type spécifique et les opérateurs utilisés doivent être compatibles avec le type des opérandes. Par exemple:

```
(x + y – increment(z))*3;
```

19.1.1 Expressions booléennes

Une expression booléenne ne doit contenir que des valeurs booléennes et/ou des opérateurs booléens et/ou des opérateurs relationnels. Elle doit donner à l'évaluation la valeur booléenne **`true`** ou **`false`**. Par exemple:

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

19.2 Affectations

Des valeurs peuvent être affectées à des variables. Cela est indiqué par le symbole **`":="`**. Au cours de l'exécution d'une affectation, le côté droit de celle-ci doit donner à l'évaluation un élément du même type que son côté gauche. L'effet d'une affectation est d'associer la variable (qui peut également être l'élément d'un type **`record`** ou **`set`**) à la valeur de l'expression. Celle-ci ne doit pas contenir de variables illimitées. Toutes les affectations se suivent dans l'ordre de leur apparition, c'est-à-dire dans un traitement de gauche à droite. Par exemple:

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

19.3 L'instruction Log

L'instruction **`log`** permet d'écrire une chaîne de caractères dans un dispositif de journalisation associé à la commande de test ou au composant de test où l'instruction est utilisée. Par exemple:

```
log("Line 248 in PTC_A");
// La chaîne "Line 248 in PTC_A" est écrite dans un dispositif de
// journalisation du système de test
```

NOTE – La définition des capacités complexes de journalisation et de suivi du cheminement, qui peuvent dépendre d'un utilitaire, sont hors du domaine de la présente Recommandation.

19.4 L'instruction Label

L'instruction **`label`** permet de spécifier des étiquettes dans des tests élémentaires, dans des fonctions, dans des variantes nommées et dans la partie commande d'un module. Une instruction **`label`** peut être utilisée librement comme d'autres instructions de programmation comportementale TTCN-3 selon les règles de syntaxe définies en Annexe A. Elle peut être utilisée avant ou après une instruction TTCN-3 mais ne doit pas être utilisée, par exemple, en tant que première instruction d'alternative contenue dans une instruction **`alt`** ou **`interleave`** (voir § 20.2.7).

19.5 L'instruction Goto

L'instruction **`goto`** peut être utilisée dans les fonctions, les tests élémentaires, les variantes nommées et dans la partie commande d'un module TTCN. Elle effectue un saut vers une étiquette ou vers le début d'une instruction **`alt`** afin de forcer un comportement répété (voir § 20.2.8).

19.6 L'instruction If-else

L'instruction **`if-else`**, également dite *instruction conditionnelle*, sert à indiquer un branchement logique dans le flux de commande dû à des expressions booléennes. Schématiquement, l'instruction conditionnelle se présente comme suit:

```
if (expression1)
     statementblock1
else
     statementblock2
```

où statementblock$_x$ se rapporte à un bloc d'instructions.

Exemple:

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

if (MyVar < 10) {
       MyVar := MyVar * 10;
       log ("MyVar < 10");
}
else {
     MyVar := MyVar/5;
}
```

Une proposition plus complexe pourrait être:

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

Dans de tels cas, la lisibilité dépend fortement du formatage mais celui-ci ne doit pas avoir d'incidence syntaxique ou sémantique.

19.7 L'instruction For

L'instruction **for** définit une boucle de compteur. La valeur de la variable d'index est augmentée, diminuée ou manipulée de façon qu'un critère de terminaison soit atteint à l'issue d'un certain nombre de boucles d'exécution..

L'instruction **for** contient deux affectations et une expression booléenne. La première affectation est nécessaire pour initialiser la variable d'index (ou de compteur) de la boucle. L'expression booléenne termine la boucle. La deuxième affectation sert à manipuler la variable d'index. Par exemple:

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

Le critère de terminaison de la boucle doit être exprimé par l'expression booléenne. Il est vérifié au début de chaque nouvelle itération de la boucle. Si sa valeur est trouvée **true**, l'exécution continue par l'instruction qui suit immédiatement la boucle **for**.

La variable d'index d'une boucle **for** peut être déclarée avant d'être utilisée dans l'instruction **for** ou être déclarée et initialisée dans l'en-tête de cette instruction. Si la variable d'index est déclarée et initialisée dans l'en-tête d'instruction **for**, la portée de la variable d'index est limitée au corps de boucle, c'est-à-dire n'est visible qu'à l'intérieur du corps de boucle. Par exemple:

```
var integer j;                          // Déclaration de la variable d'entier j
for (j:=1; j<=10; j:= j+1) { ... }      // Utilisation de la variable j comme
                                        // variable d'index de boucle
```

```
for (var float i:=1.0; i<7.9; i:= i*1.35) { ... }     // La variable d'index i
                                  // est déclarée et initialisée dans
                                  // l'en-tête de boucle. La variable I n'est
                                  // visible que dans le corps de boucle
```

19.8 L'instruction While

Une boucle **while** est exécutée tant que la condition de la boucle est vraie. La condition de boucle doit être vérifiée au début de chaque nouvelle itération de boucle. Si la condition de boucle n'est pas vraie, la boucle est terminée et l'exécution doit se poursuivre par l'instruction qui suit immédiatement la boucle **while**. Par exemple:

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

19.9 L'instruction Do-while

La boucle **do-while** est identique à une boucle **while** sauf que la condition de boucle doit être vérifiée à la *fin* de chaque itération de boucle. En d'autres termes, lors de l'utilisation d'une boucle **do-while**, le comportement est exécuté au moins une fois avant que la condition de boucle soit évaluée pour la première fois. Par exemple:

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

19.10 L'instruction d'arrêt d'exécution (Stop)

L'instruction **stop** donne à une exécution une fin différente selon le contexte dans lequel elle est employée: si elle est utilisée dans la partie commande d'un module, elle met fin à l'exécution de l'ensemble du module. Si elle est utilisée dans une fonction qui exécute un comportement, elle met fin au composant de test correspondant.

20 Instructions de programmation comportementales

Les instructions de programmation comportementales peuvent être utilisées dans des tests élémentaires, dans des fonctions et dans la partie commande d'un module, sauf pour l'instruction de retour, qui ne doit être utilisée que dans des tests élémentaires et des fonctions. Les instructions de programmation comportementales spécifient le comportement dynamique des composants de test connectés aux accès de communication. Le comportement de test peut être exprimé séquentiellement sous la forme d'un ensemble de variantes ou d'une combinaison de séquences et de variantes. Un opérateur d'entrelacement permet de spécifier des séquences entrelacées ou des variantes.

Tableau 12/Z.140 – Aperçu général des instructions de programmation comportementales TTCN-3

Instructions de programmation comportementales	
Instruction	**Mot clé ou symbole associé**
Comportement à variantes	**alt { ... }**
Variante nommée	**named alt { ... }**
Comportement entrelacé	**interleave { ... }**
Activation d'un défaut	**activate**
Désactivation d'un défaut	**deactivate**
Commande de retour	**return**

20.1 Comportement séquentiel

La forme la plus simple d'un comportement est un ensemble d'instructions exécutées séquentiellement, comme illustré à la Figure 5.

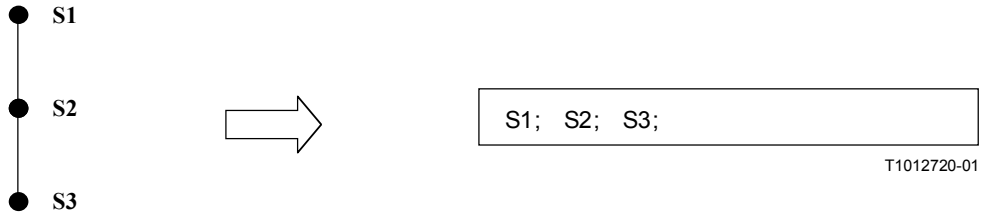


Figure 5/Z.140 – Illustration d'un comportement séquentiel

Les instructions individuelles de la séquence doivent être séparées par le délimiteur "**;**". Par exemple:

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

20.2 Comportement à variantes

Une forme de comportement plus complexe exprime des séquences d'instructions par des ensembles de variantes possibles afin de former une arborescence de chemins d'exécution, comme illustré à la Figure 6.

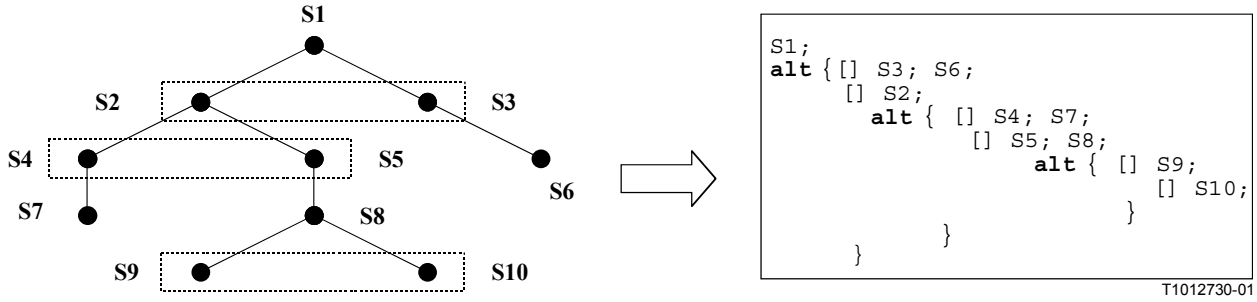


Figure 6/Z.140 – Illustration d'un comportement à variantes

L'instruction **alt** indique un embranchement logique de comportement de test dû à la réception et au traitement d'événements de communication et/ou de temporisation et/ou à la terminaison de composants de test parallèles. En d'autre termes, cette instruction est associée à l'emploi des opérations TTCN-3 **receive**, **trigger**, **getcall**, **getreply**, **catch**, **check**, **timeout** et **done**. L'instruction **alt** indique un ensemble d'événements possibles qui doivent être appariés en fonction d'une analyse sélective particulière (voir § 20.2.1).

NOTE – L'instruction **alt** correspond aux variantes situées au même niveau de décalage en notation TTCN-2. Il y a cependant trois différences notables:

a) les expressions booléennes permettant de désactiver des variantes ne peuvent être formulées que dans une instruction d'alternative;

b) il n'est pas possible d'examiner la file d'attente aux accès au moyen d'une expression booléenne puis de désactiver une variante;

c) il n'est pas possible d'appeler une fonction en tant que variante dans l'instruction **alt**, sauf si une sauvegarde conditionnelle (c'est-à-dire un opérateur [**else**]) est le dernier choix dans l'instruction d'alternative (voir § 20.2.3).

Exemple:

```
// Utilisation d'instructions d'alternative imbriquées
:
alt
{
[]   L1.receive(DL_REL_CO:*)                    // Message UA ou DM reçu;
     { verdict.set(pass);                        // couche 2 libérée
       TAC.stop;
       TNOAC.start;
       alt {
       [] L1.receive(DL_EST_IN)                 // Message SABME reçu
          {   TNOAC.stop;
              verdict.set(pass);
          }
       [] TNOAC.timeout
          {   L1.send(DEL_EST_RQ:*);
              TAC.start;
              alt {
              [] L1.receive(DL_EST_CO:*)    // Message UA reçu; couche Liaison de
                                            // données établie
                 { TAC.stop;
                   verdict.set(pass)
                 }
              [] TAC.timeout                // Pas de réponse
                 {verdict.set(inconc)}
              [] L1.receive                 // Comme OTHERWISE en TTCN-2
                 {verdict.set(inconc)}
                  }
          }
       [] L1.receive                        // Comme OTHERWISE en TTCN-2
          {verdict.set(inconc)}
       }
     }
[]   TAC.timeout                            // Pas de réponse
     {verdict.set(inconc)}
[]   L1.receive                             // Comme OTHERWISE en TTCN-2
     {verdict.set(inconc)}
}
:

// Utilisation d'instruction d'alternative avec des expressions (ou sauvegardes)
// booléennes
:
alt {
[]      L1.receive(MyMessage1)
        {verdict.set(fail)}
[x>1]   L2.receive(MyMessage2)              // Sauvegarde/expression booléenne
        {verdict.set(pass)}
[x<=1]  L2.receive(MyMessage3)              // Sauvegarde/expression booléenne
        {verdict.set(inconc)}
}
:

// Utilisation de done dans des instructions d'alternative
:
alt {
     []   MyPTC.done {
          verdict.set(pass)
       }
```

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

20.2.1 Exécution d'un comportement à variantes

Les instructions d'alternative contenues dans une instruction **alt** sont traitées dans l'ordre de leur apparition. La sémantique opérationnelle de la notation TTCN-3 (voir Annexe B) part du principe que le statut d'un quelconque des événements ne peut pas changer au cours du processus d'essai d'appariement d'une variante contenue dans un ensemble de variantes. Cela implique qu'une sémantique d'analyse sélective est utilisée pour les événements reçus et pour les temporisations, c'est-à-dire qu'une vue sélective des événements qui ont été reçus et des temporisations qui ont expiré est prise à chaque fois autour d'un ensemble de variantes. Seules les variantes identifiées dans la vue sélective pourront être appariées lors du prochain cycle d'exploration des variantes.

NOTE 1 – Cette sémantique est exactement la même que pour la notation TTCN-2.

NOTE 2 – Les événements synchrones (par exemple, **call**) bloquent la boucle jusqu'à ce qu'une communication ait eu lieu.

20.2.2 Sélection/désélection d'une variante

Il est possible, au besoin, d'activer ou de désactiver une variante au moyen d'une expression booléenne placée entre les crochets "[ ]" de la variante. Par exemple:

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

Les crochets d'ouverture et de fermeture ("[" et "]") doivent être présents au début de chaque variante, même s'ils sont vides. Cela non seulement facilite la lecture mais est également nécessaire pour distinguer syntaxiquement une variante d'une autre.

20.2.3 Branche "else" d'une alternative

Il est possible, au besoin, de définir dans l'instruction d'alternative une branche qui sera toujours suivie si aucune autre variante préalablement définie ne peut être choisie. Si une branche **else** est définie, toutes les variantes définies par la suite sont redondantes (c'est-à-dire qu'elles ne pourront jamais être atteintes). Par exemple:

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

Il convient de noter que des options par défaut sont toujours ajoutées à la fin de toutes les alternatives. Si une branche **else** est définie, une option **default** activée ne sera jamais choisie.

NOTE – Il est également possible d'utiliser l'option **else** dans des variantes nommées.

20.2.4 Déclaration de variantes nommées

Les variantes utilisées à un certain nombre d'emplacements peuvent être définies dans une variante nommée qui est désignée par la paire de mots clés **named alt**. Les variantes nommées doivent être définies globalement dans les définitions de module. Une fois invoquée, une variante nommée est identique au comportement de la structure **alt**, sauf qu'elle possède un identificateur et permet une paramétrisation.

Lorsqu'elle est citée en référence, une variante nommée a le même effet qu'une substitution de macro. Une variante nommée peut être citée en référence à tout emplacement d'une définition de comportement où il est autorisé d'inclure une structure **alt** normale.

Exemple:

```
// Définition de la macro de variantes nommées
named alt HandlePCO2()
{
     []   PCO2.receive(DL_EST_IN)
          {PCO2.send(DL_EST_CO)}

     []   PCO2.receive(DL_EST_CO) {}
       // ne rien faire
}

// Utilisation d'une variante nommée en ligne
testcase TC001() runs on MyPTCtype
{
     :
       HandlePCO2();              // Appel de la structure named alt
     :
}

// Qui se développe en:
testcase TC001() runs on MyPTCtype
{
     :
     alt  {
          []  PCO2.receive(DL_EST_IN)
              {PCO2.send(DL_EST_CO)}
          []  PCO2.receive(DL_EST_CO) {}
          // ne rien faire
     }
     :
}
```

20.2.5 Expansion de variantes au moyen de variantes nommées

En plus du référencement direct en ligne, il est possible d'effectuer une expansion explicite des variantes spécifiées dans la structure **named alt** en utilisant l'instruction **expand**. Celle-ci peut être placée à tout endroit dans une instruction **alt**. Elle y insérera les sauvegardes associées qui sont contenues dans la structure **named alt**.

Exemple:

```
// Utilisation d'une structure named alt par expansion
testcase TC002()runs on MyPTCtype
{
  :
```

```
     alt  {
       [] PCO1.receive(DL_EST_IN)
          {PCO1.send(DL_EST_CO)}
       [] PCO1.receive(DL_EST_CO) {}
          // ne rien faire
       [expand] HandlePCO2() // Expansion de variantes nommées en fonction
                             // d'une instruction d'alternative spécifiée
       }
}

// Qui s'étend comme suit:
testcase TC002()runs on MyPTCtype
{
     :
     alt  {
          []  PCO1.receive(DL_EST_IN)
              {PCO1.send(DL_EST_CO)}
          []  PCO1.receive(DL_EST_CO) {}
          // ne rien faire
          []  PCO2.receive(DL_EST_IN)
              {PCO2.send(DL_EST_CO)}
          []  PCO2.receive(DL_EST_CO) {}
          // ne rien faire
     }
}
```

20.2.6 Paramétrisation de variantes nommées

Les variantes nommées peuvent être paramétrées avec des types, des valeurs, des fonctions et des modèles. Comme les variantes nommées ne sont pas des unités de portée, les paramètres formels à définir sont simplement remplacés par les paramètres réels correspondants au moment de l'exécution de la macro d'expansion.

Exemple:

```
named alt HandleAnyPCO(MyPortT PCO)
{
     []   PCO.receive(DL_EST_IN)
          {PCO.send(DL_EST_CO)}
     []   PCO.receive(DL_EST_CO) {}
       // ne rien faire
}

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

20.2.7 L'instruction "Label" dans un comportement

L'instruction **label** permet de spécifier des étiquettes dans des tests élémentaires, dans des fonctions, dans des variantes nommées et dans la partie commande d'un module. Elle peut être utilisée avant ou après une quelconque instruction TTCN-3 mais ne doit pas être la première instruction d'une variante dans une instruction **alt** ou **interleave**.

Exemple:

```
label MyLabel;
// Définit l'étiquette MyLabel

// Les étiquettes L1, L2 et L3 sont définies dans le fragment de code TTCN-3
// suivant:
 :
label L1;                              // Définition de l'étiquette L1
alt{
[]   PCO1.receive(MySig1)
     { label L2;                       // Définition de l'étiquette L2
       PCO1.send(MySig2);
       PCO1.receive(MySig3)
     }
[]   PCO2.receive(MySig4)
     { PCO2.send(MySig5);
       PCO2.send(MySig6);
       label L3;                       // Définition de l'étiquette L3
       PCO2.receive(MySig7);
       goto L1;                        // Sauter jusqu'à l'étiquette L1
     }
}
:
```

20.2.8 L'instruction "Goto" dans un comportement

L'instruction **goto** peut être utilisée dans des fonctions, dans des tests élémentaires, dans des variantes nommées et dans la partie commande d'un module TTCN. L'instruction **goto** effectue un saut jusqu'à une étiquette ou jusqu'au début d'une instruction **alt** afin de forcer la répétition d'un comportement.

La réévaluation d'une instruction **alt** peut être effectuée:

a) soit au moyen de l'expression **goto** *<LabelId>* où l'instruction d'étiquetage correspondante doit être placée immédiatement avant le mot clé **alt** de l'alternative réelle vers laquelle le saut est effectué;

b) soit au moyen de l'expression **goto alt** à l'intérieur de l'instruction **alt** à réévaluer. Dans ce cas, le mot clé **alt** peut être considéré comme une étiquette implicite pour l'instruction **alt** à l'intérieur de laquelle l'instruction **goto** est utilisée.

20.2.8.1 Restriction d'utilisation de l'instruction "Goto"

L'instruction **goto** permet de sauter librement, en avant ou en arrière, à l'intérieur d'une séquence d'instructions, de sortir d'une instruction composite donnée (par exemple une boucle **while**) et de sauter plusieurs niveaux afin de sortir d'instructions composites imbriquées (par exemple, des options imbriquées). Mais l'utilisation de l'instruction **goto** doit être limitée par les règles suivantes:

a) il n'est pas permis de sauter à l'intérieur ou à l'extérieur de fonctions, de tests élémentaires, de variantes nommées et de la partie commande d'un module TTCN;

b) il n'est pas permis de sauter à l'intérieur d'une séquence d'instructions définie dans une instruction composite (c'est-à-dire une instruction de type **alt**, une boucle **while**, une boucle **for**, une instruction **if-else**, une boucle **do-while** et l'instruction **interleave**);

c) à titre d'exception à la règle a) concernant les variantes nommées, il est permis d'utiliser l'instruction **goto alt** à l'intérieur d'une variante nommée afin de forcer la réévaluation d'une instruction **alt** à l'intérieur de laquelle cette variante nommée peut faire l'objet d'une expansion;

NOTE – Cette règle permet de sauter hors d'une variante nommée de manière restreinte afin d'offrir la capacité de décrire des valeurs par défaut.

d) il n'est pas permis d'utiliser l'instruction goto à l'intérieur d'une instruction **interleave**.

Exemple:

```
// Le fragment de code TTCN-3 suivant contient
 :
label L1;
MyVar := 2 * MyVar;
if (MyVar < 2000) { goto L1; }             // ... saut en arrière L1 et
MyVar2 := Myfunction(MyVar);
if (MyVar2 > MyVar) { goto L2; }           // ... saut en avant vers L2,
PCO1.send(MyVar);
PCO1.receive -> value MyVar2;
label L2;
PCO2.send(integer: 21);
alt {
     []   PCO1.receive
       { goto alt; }              // ... saut qui force la réévaluation
                                  // de l'instruction alt précédente
     []   PCO2.receive(integer: 67)
       {  label L3;
          PCO2.send(MyVar);
          alt {
           [] PCO1.receive
              { goto alt; }  // ... nouveau saut qui force la réévaluation
                             // de l'instruction alt précédente (ce qui
                             // est différent de l'instruction goto
                             // précédemment),
           [] PCO2.receive(integer: 90)
              {   PCO2.send(integer: 33);
                  PCO2.receive(integer: 13);
                  goto L4;  // ... saut en avant hors de deux instructions
                            // alt imbriquées,
              }
           [] PCO2.receive(MyError)
              { goto L3; }    // ... saut en arrière hors de l'instruction alt
                             // actuelle,
           [] any port.receive
              { goto L2; }    // ... saut en arrière hors de deux instructions
                             // alt imbriquées,
          }
       }
     []   any port.receive
       { goto L2; }// ... et un grand saut en arrière hors d'une instruction alt
}
label L4;
:
```

20.3 Comportement entrelacé

Les instructions de transfert de commande **for**, **while**, **do-while**, **goto**, **activate**, **deactivate**, **stop**, **return** et les appels (directs et indirects) de fonctions définies par l'utilisateur, qui comportent des opérations de communication, ne doivent pas être utilisées dans les instructions de type **interleave**. Par ailleurs, il n'est pas permis de sauvegarder des branches d'une instruction **interleave** avec des expressions booléennes (c'est-à-dire que les crochets [] doivent toujours être vides). Il n'est pas permis non plus d'effectuer une expansion d'instructions **interleave** au moyen de variantes nommées ou de spécifier des branches conditionnelles **else** dans un comportement entrelacé.

Un comportement entrelacé peut toujours être remplacé par un ensemble équivalent de variantes imbriquées. Les procédures de ce remplacement sont décrites dans l'Annexe B.

La règle d'évaluation d'une instruction d'entrelacement est la suivante:

a) chaque fois qu'une instruction de réception est exécutée, les instructions de non-réception suivantes sont exécutées par la suite jusqu'à ce que la prochaine instruction de réception soit atteinte ou que la séquence entrelacée soit terminée;

   NOTE – Les instructions de réception sont des instructions TTCN-3 qui peuvent apparaître dans des ensembles de variantes, c'est-à-dire **receive**, **check**, **trigger**, **getcall**, **getreply**, **catch** et **timeout**. Les instructions de non-réception indiquent toutes les instructions autres que de transfert de commande qui peuvent être utilisées dans l'instruction d'entrelacement.

b) l'évaluation se poursuit alors par la prise de la prochaine vue sélective.

La sémantique opérationnelle de l'entrelacement est entièrement définie dans l'Annexe B.

Exemple:

```
// Le fragment de code TTCN-3 suivant
 :
interleave {
[]   PCO1.receive(MySig1)
     { PCO1.send(MySig2);
       PCO1.receive(MySig3);
     }
[]   PCO2.receive(MySig4)
     { PCO2.send(MySig5);
       PCO2.send(MySig6);
       PCO2.receive(MySig7);
     }
}
:

// Peut être interprété comme une notation abrégée pour
:
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 Comportement par défaut

Le comportement par défaut peut être considéré comme une extension d'une instruction **alt** ou comme une simple opération de réception qui est définie de manière spéciale. Un comportement par défaut doit être défini par la spécification d'une variante **named alt** et doit être activé avant de pouvoir être invoqué et exécuté.

L'activation d'un comportement par défaut signifie que les variantes définies dans la variante **named alt** correspondante sont ajoutées au niveau sommital de toutes les variantes subséquentes.

Le comportement par défaut est également ajouté à toute opération de réception simple (c'est-à-dire ailleurs que dans une instruction **alt**), aux temporisations et aux instructions d'achèvement **done**, cela parce que ces opérations se réduisent théoriquement à une seule alternative. Par exemple:

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

// Est identique à
:
alt {
     [] MyPort.receive(MyMsg) {}
}
:
```

20.4.1 Les opérations Activate et Deactivate

Un comportement par défaut est activé par l'utilisation de l'opération **activate** et est désactivé par l'utilisation de l'opération **deactivate**. Une opération **deactivate** vide désactive tous les comportements par défaut qui sont actifs.

Dans le cas de l'activation multiple de plusieurs variantes nommées, les éléments **alt** doivent faire l'objet d'une expansion dans l'ordre d'activation.

Si l'argument d'une opération d'activation est une liste de variantes nommées, les éléments **alt** doivent faire l'objet d'une expansion dans l'ordre indiqué par la liste.

Exemple:

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

```
:

// définition de comportement interne
activate( Default1() );

CL2.receive(MySetup);

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

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

// Cette instruction désactive le comportement par défaut Default1
deactivate(Default1);
// Cette instruction désactive tout comportement par défaut déjà activé
deactivate;

// Théoriquement, après définition et activation la variante par défaut subit
// une expansion jusqu'à la fin de toute instruction alt ou receive subséquente

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

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

// ce qui équivaut à
:
alt {
       [] CL2.receive(MySetup);   // L'instruction de réception simple devient
                                  // donc une instruction d'alternative à part
                                  // entière

       [] MyPort.check
          {MyBehaviour1()}
}

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

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

20.5 L'instruction Return

L'instruction **return** met fin à l'exécution d'une fonction et renvoie la commande au point à partir duquel la fonction a été appelée. Une instruction **return** peut (à titre d'option) être associée à une valeur de retour. L'utilisation du retour dans un test élémentaire ou dans une commande est équivalent à l'instruction **stop**.

Exemple:

```
function MyFunction() return boolean
{
     :
     if (date == "1.1.2000") { return false; }
// l'exécution s'arrête le 1.1.2000 et renvoie la valeur "false" en tant
// qu'indication de défaillance
     :
     return true;  // la valeur "true" est renvoyée
}

function MyBehaviour() return verdicttype
{
     :
     if (MyFunction()) { verdict.set(pass); }   // Utilisation de MyFunction dans
                                                // une instruction "if"
     else { verdict.set(inconc); }
     :
     return verdict.get; // Retour explicite du verdict
}
```

21 Opérations de configuration

Les opérations de configuration (voir Tableau 13) servent à installer et à commander des composants de test. Ces opérations ne doivent être utilisées que dans les tests élémentaires et fonctions de la notation TTCN-3 (c'est-à-dire hors partie commande du module).

Tableau 13/Z.140 – Aperçu général des opérations de configuration TTCN-3

Opérations de configuration	
Instruction	**Nom de l'opération**
Création de composant de test parallèle	**create**
Connexion d'un composant à un autre composant	**connect**
Déconnexion de deux composants	**disconnect**
Mappage d'un accès de composant vers accès d'interface de test	**map**
Démappage d'un accès à partir de l'interface du système de test	**unmap**
Obtention d'adresse de composant MTC	**mtc**
Obtention d'adresse d'interface de système de test	**system**
Obtention d'adresse propre	**self**
Début d'exécution d'un composant de test	**start**
Arrêt d'exécution d'un composant de test	**stop**
Contrôle de terminaison d'un composant PTC	**running**
Attente de terminaison d'un composant PTC	**done**

21.1 L'opération de création

Le composant MTC est le seul composant de test qui est automatiquement créé au début d'un test élémentaire. Tous les autres composants de test sont créés explicitement au cours de l'exécution du test par l'opération **create**. Un composant est créé avec son ensemble complet d'accès, auxquels les files d'entrée sont vides. Par ailleurs, si un accès est défini comme étant du type **in** ou **inout**, il doit se trouver en état d'écoute, prêt à recevoir du trafic par la connexion.

Etant donné que tous les composants et tous les accès sont implicitement détruits lors de la terminaison de chaque test élémentaire, celui-ci doit créer complètement sa configuration requise de composants et de connexions lors de son invocation.

```
// Cet exemple déclare une variable de type adresse, qui est utilisée pour
// mémoriser la référence d'un composant nouvellement créé de type
// MyComponentType qui est le résultat de la fonction de création.
:
var MyComponenttype MyNewComponent;
 :
MyNewComponent := MyComponentType.create;
 :
```

L'opération de création **`create`** doit renvoyer l'unique référence de composant de l'instance nouvellement créée. L'unique référence du composant sera normalement mémorisée dans une variable (voir § 8.6) et pourra être utilisée pour connecter des instances et à des fins de communication comme l'émission et la réception.

Les composants peuvent être créés à tout point d'une définition de comportement offrant une flexibilité totale en termes de configurations dynamiques (c'est-à-dire que tout composant peut créer tout autre composant). La visibilité des références de composant doit suivre les mêmes règles de portée que celles des variables. Afin de faire référence à des composants situés à l'extérieur de leur portée de création, la référence de composant doit être transmise sous forme de paramètre ou sous forme de champ dans un message.

21.2 Les opérations de connexion et de mappage

Les accès d'un composant de test peuvent être connectés à d'autres composants ou aux accès de l'interface avec le système de test. Dans le cas de connexions entre deux composants de test, l'opération **`connect`** doit être utilisée. L'opération **`map`** doit être utilisée lors de la connexion d'un composant de test à une interface avec le système de test. L'opération **`connect`** relie directement un accès à un autre avec le côté **`in`** relié au côté **`out`** et inversement. L'opération **`map`** peut par ailleurs être considérée comme étant simplement une conversion de nom définissant la façon dont les flux de communication doivent être référencés. (Voir Figure 7.)

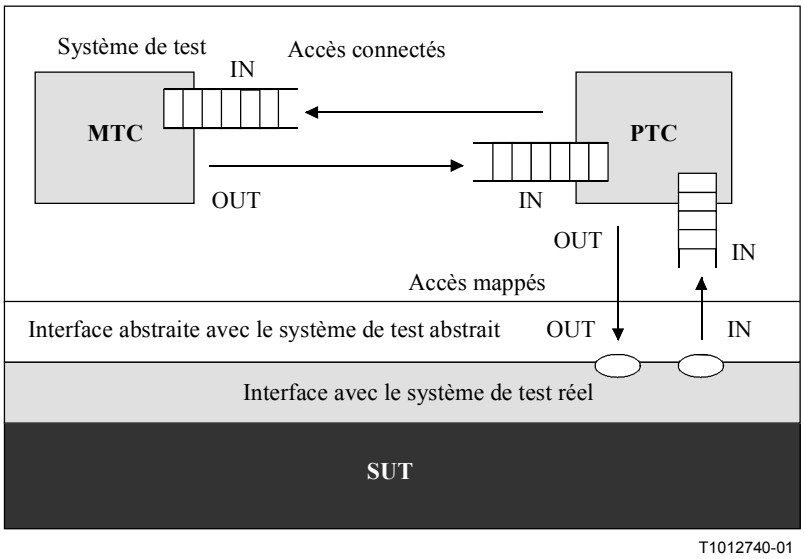


Figure 7/Z.140 – Illustration des opérations de connexion et de mappage

Avec les deux opérations de connexion **`connect`** et de mappage **`map`**, les accès à connecter sont identifiés par les références des composants à connecter et par les noms des accès à connecter.

Il existe deux opérations permettant d'identifier le composant MTC (ou **mtc**) et l'interface avec le système de test (ou **system**). Ces deux opérations (voir § 8.6) peuvent être utilisées pour identifier et connecter des accès.

Les deux opérations **connect** et **map** peuvent être appelées à partir de toute définition (fonction) de comportement. Avant qu'une ou l'autre opération soit appelée, les composants à connecter doivent cependant avoir été créés et leurs références de composant doivent être connues en même temps que les noms des accès correspondants.

Les deux opérations **connect** et **map** permettent la connexion d'un accès à plusieurs autres accès. Il n'est pas permis de connecter vers un accès mappé ou de mapper vers un accès connecté.

Exemple:

```
// L'on part de l'hypothèse que les accès Port1, Port2, Port3 et PCO1 sont
// correctement définis et déclarés dans les définitions correspondantes
// de type d'accès et de type de composant
:
var MyComponentType  MyNewComponent;
 :
MyNewComponent := MyComponentType.create;
 :
 :
connect(MyNewComponent:Port1, mtc:Port3);
map(MyNewComponent:Port2, system:PCO1);
 :
 :
// Dans cet exemple un nouveau composant de type MyComponentType est créé et
// sa référence est mémorisée dans la variable MyNewComponent. Ensuite, dans
// l'opération de connexion, l'accès Port1 de ce nouveau composant est
// connecté à l'accès Port3 du MTC. Au moyen de l'opération de mappage,
// l'accès Port2 du nouveau composant est alors connecté à l'accès PCO1 de
// l'interface avec le système de test
```

21.2.1 Connexions cohérentes

Pour les deux opérations **connect** et **map**, seules des connexions cohérentes sont permises.

Soit les hypothèses suivantes:

a) les accès PORT1 et PORT2 sont les accès à connecter;

b) inlist-PORT1 définit les messages ou procédures dans le sens entrant de PORT1;

c) outlist-PORT1 définit les messages ou procédures dans le sens sortant de PORT1;

d) inlist-PORT2 définit les messages ou procédures dans le sens entrant de PORT2;

e) outlist-PORT2 définit les messages ou procédures dans le sens sortant de PORT2.

L'opération **connect** est permise si et seulement si:

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

L'opération **map** (en supposant que PORT2 est l'accès de l'interface avec le système de test) est permise si et seulement si:

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

Dans tous les autres cas, ces opérations ne sont pas permises.

Etant donné que la notation TTCN-3 permet des configurations et adresses dynamiques, ces vérifications de cohérences ne peuvent pas être toutes effectuées statiquement au moment de la compilation. Toutes les vérifications qui n'ont pas pu être faites au moment de la compilation doivent l'être à celui de l'exécution du programme et doivent conduire, en cas de défaillance, à une erreur de test élémentaire.

21.3 Les opérations de déconnexion et de démappage

Les opérations **disconnect** et **unmap** sont les contraires des précédentes. Elles effectuent la déconnexion d'accès (déjà connectés) de composants de test et le démappage d'accès (déjà mappés) de composants de test et d'accès à l'interface avec le système de test.

Les deux opérations **disconnect** et **unmap** peuvent être appelées à partir de tout composant si l'on connaît ses références ainsi que le nom des accès correspondants. Une opération de déconnexion **disconnect** ou de démappage **unmap** n'a d'effet que si la connexion ou le mappage à supprimer a déjà fait l'objet d'une création.

Exemple:

```
 :
 :
connect(MyNewComponent:Port1, mtc:Port3);
map(MyNewComponent:Port2, system:PCO1);
 :
 :
disconnect(MyNewComponent:Port1, mtc:Port3);    // Déconnexion d'une connexion
                                                // déjà établie
unmap(MyNewComponent:Port2, system:PCO1);       // Démappage d'un mappage
                                                // existant
```

21.4 Les opérations MTC, System et Self

La référence de composant (voir § 8.6) possède deux opérations: **mtc** et **system** qui renvoient respectivement la référence du composant de test principal et celle de l'interface avec le système de test. Par ailleurs, l'opération **self** peut être utilisée pour renvoyer la référence du composant dans lequel elle est appelée. Par exemple:

```
var MyComponentType MyAddress;
MyAddress := self; // Mémoriser la référence de composant actuelle
```

Les seules opérations permises sur les références de composant sont l'affectation et l'équivalence.

21.5 L'opération de lancement de composant de test

Une fois qu'un composant a été créé et connecté, il faut associer un comportement à ce composant et en lancer l'exécution. L'on utilise à cette fin l'opération **start** (la création de composant ne lance pas l'exécution du comportement du composant). La raison de la distinction entre les opérations **create** et **start** est de permettre d'effectuer les opérations de connexion avant d'exécuter réellement le composant de test.

L'opération de lancement **start** doit associer le comportement requis au composant de test. Ce comportement est défini par référence à une fonction déjà définie. Par exemple:

```
// L'on part de l'hypothèse que les accès Port1, Port2, Port3 et PCO1 sont
// correctement définis et déclarés dans les définitions correspondantes de
// type d'accès et de type de composant
:
var MyComponentType MyNewComponent;
 :
MyNewComponent := MyComponentType.create;
 :
connect(MyNewComponent:Port1, mtc:Port3);
connect(MyNewComponent:Port2, system:PCO1);
 :
 :
MyNewComponent.start(MyComponentBehaviour());
 :
```

```
// Dans cet exemple, un nouveau composant est d'abord créé, puis connecté à son
// environnement et finalement est lancé au moyen de l'opération "start". Pour
// identifier le composant à exécuter sa référence est utilisée
```

Les restrictions suivantes s'appliquent à une fonction invoquée dans une opération de lancement **start** de composant de test:

– si cette fonction possède des paramètres, ceux-ci ne doivent être que du type **in**, c'est-à-dire des paramètres de valeur;

– cette fonction doit avoir une définition de type **runs on** faisant référence au même type de composant que le composant nouvellement créé ou doit transmettre sous forme de paramètres toutes les informations à extraire de la définition du type de composant;

– les accès et les temporisateurs ne peuvent être transmis à cette fonction que s'ils se rapportent à des accès et à des temporisateurs contenus dans la définition du type du composant nouvellement créé, c'est-à-dire que les accès et les temporisateurs sont locaux par rapport aux instances de composant et ne doivent pas être transmis à d'autres composants.

NOTE – La capacité de transmettre des accès comme paramètres permet de spécifier des fonctions génériques qui ne sont pas associées à un seul type de composant spécifique.

21.6 L'opération d'arrêt de composant de test

L'instruction d'arrêt **stop** de composant de test met fin explicitement à l'exécution du composant de test dans lequel l'arrêt est appelé. Cette opération ne contient pas d'arguments. Par exemple:

```
if (date == "1.1.2000") { stop; }// L'exécution s'arrête le 1.1.2000
```

Si le composant de test qui est arrêté est le MTC, tous les composants PTC restant en cours doivent également être arrêtés et le test élémentaire se termine.

NOTE – Le mécanisme concret d'arrêt de tous les composants restant en cours est hors du domaine d'application de la présente Recommandation.

Les ressources doivent être toutes libérées lorsqu'un composant de test se termine, soit explicitement au moyen de l'opération d'arrêt **stop** ou lorsque le traitement parvient à une instruction **return** dans la fonction qui a lancé à l'origine la composante de test, soit implicitement lorsque le composant arrive à la fin de son arborescence comportementale. Toute variable mémorisant une référence de composant arrêté doit être vide.

Les règles de terminaison de tests élémentaires et de calcul du verdict de test final sont décrites au § 24.

21.7 L'opération d'exécution

L'opération **running** permet l'exécution d'un comportement dans un composant de test afin de s'assurer que le comportement exécuté dans un autre composant de test est achevé. L'opération d'exécution **running** est considérée comme étant une expression booléenne et, en tant que telle, renvoie une valeur booléenne indiquant si le composant de test spécifié (ou tous les composants de test spécifiés) s'est terminé. Contrairement à l'opération **done**, l'opération **running** peut être utilisée librement dans des expressions booléennes. Par exemple:

```
if (PTC1.running)                          // Utilisation de l'opération "running"
                                           // dans une instruction "if"
{
     // Faire quelque chose!
}
```

```
while (all component.running != true) { // Utilisation de l'opération "running"
                                        // dans une itération
       MySpecialFunction()
}
```

21.8 L'opération de fin d'exécution

L'opération **done** permet l'exécution d'un comportement dans un composant de test afin de s'assurer que le comportement exécuté dans un autre composant de test est achevé.

L'opération **done** doit être utilisée de la même façon qu'une opération de réception ou de temporisation **timeout**. En d'autres termes, elle ne doit pas être utilisée dans une expression booléenne mais peut l'être afin de déterminer une variante dans une instruction **alt** ou comme instruction autonome dans une description de comportement. Dans ce dernier cas, une opération **done** est considérée comme une notation abrégée pour une instruction **alt** avec une seule variante, c'est-à-dire qu'elle a une sémantique bloquante et offre donc la possibilité d'une attente passive de la terminaison de composants de test.

NOTE – L'opération **done** possède une sémantique identique en notation TTCN-3 et en notation TTCN-2.

Exemple:

```
// Utilisation de l'opération "done" dans une alternative
:
alt {
   [] MyPTC.done {
         verdict.set(pass)
      }

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


// l'opération "done" ci-après est une instruction autonome:
 :
all component.done;
 :

// a la signification suivante:
 :
alt {
   [] all component.done {}
}
 :

// et donc, bloque l'exécution jusqu'à ce que tous les composants de test
// parallèles se soient terminés
```

21.9 Utilisation de séquences tabulaires de composants

Les opérations **create**, **connect**, **start** et **stop** ne s'appliquent pas directement à des séquences tabulaires de composants. En revanche, un élément spécifique de la séquence tabulaire doit être fourni en tant que paramètre. Pour les composants, l'effet d'une séquence tabulaire est obtenu par l'utilisation de références de composant et par l'affectation de l'élément approprié de la séquence tabulaire au résultat de l'opération de création **create**.

```
// Cet exemple montre comment modéliser l'effet de la création, de la
// connexion et de l'exécution de séquences tabulaires de composants au moyen
// d'une boucle et par la mémorisation de la référence du composant créé dans
// une séquence tabulaire de références de composant.

testcase MyTestCase() runs on MyMtcType system MyTestSystemInterface
{
      :
     var integer i;
     var MyPTCType1MyPtcType[11];
      :
     for (i:= 1; i<=10; i:=i+1)
     {
       MyPtcAddresses[i] := MyPtcType1.create;
       connect(self:PtcCoordination, MyPtcAddresses[i]:MtcCordination);
       MyPtcAddresses[i].start(MyPtcBehaviour());
     }
      :
}
```

21.10 Utilisation des mots clés Any et All avec des composants

Les mots clés **any** et **all** peuvent être utilisés avec des opérations de configuration comme indiqué dans le Tableau 14.

Tableau 14/Z.140 – Utilisation des mots clés Any et All avec des composants

Opération	Permise		Exemple
	any	**all**	
create			
start			
running	Oui mais à partir d'un MTC seulement	Oui mais à partir d'un MTC seulement	**any component.running** **all component.running**
done	Oui mais à partir d'un MTC seulement	Oui mais à partir d'un MTC seulement	**any component.done** **all component.done**
stop			

22 Opérations de communication

La notation TTCN-3 prend en charge les communications en mode message (asynchrones) et en mode procédure (synchrones) (voir § 8.1). Les communications asynchrones ne bloquent pas l'opération **send**, comme décrit à la Figure 8 où le traitement dans le composant MTC continue immédiatement après l'exécution de l'opération **send**. Le système SUT est bloqué à l'opération **receive** jusqu'à ce qu'il reçoive un message d'envoi exécuté.

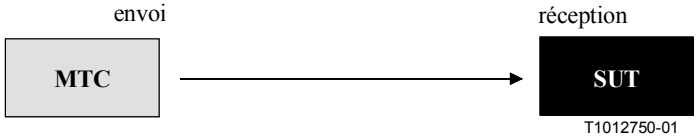


Figure 8/Z.140 – Illustration de l'opération synchrone d'envoi et de réception

Les communications synchrones bloquent l'opération appel **`call`**, comme illustré par la Figure 9, dans laquelle l'opération **`call`** bloque le traitement dans le composant MTC jusqu'à ce qu'une réponse **`reply`** ou une exception soit reçue par le système SUT. Comme pour l'opération de réception **`receive`**, l'opération obtention d'appel **`getcall`** bloque le SUT jusqu'à la réception de l'appel.

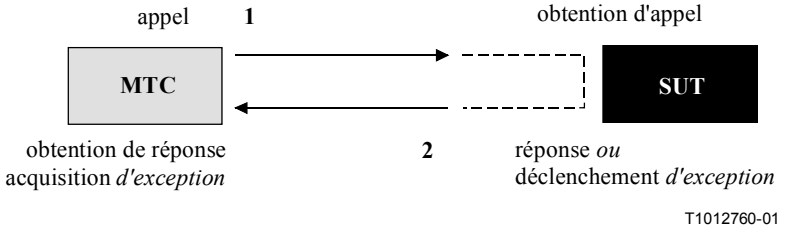


Figure 9/Z.140 – Illustration d'un appel synchrone complet

Des opérations comme **`send`** et **`call`** sont désignées collectivement par le terme *opérations de communication*. Ces opérations ne doivent être utilisées que dans les tests élémentaires et fonctions de la notation TTCN-3 (c'est-à-dire non directement dans la partie commande du module). Les opérations de communication sont divisées en trois groupes:

a) un composant envoie un message, appelle une procédure, ou répond à un appel accepté ou déclenche une exception. Ces actions sont collectivement désignées par l'expression *opérations d'envoi*;

b) un composant reçoit un message, accepte un appel de procédure, reçoit une réponse pour une procédure déjà appelée ou acquiert une exception. Ces actions sont collectivement désignées par l'expression *opérations de réception*;

c) une commande d'accès à un accès est effectuée par une opération **clear**, **start** ou **stop**. Ces actions sont collectivement désignées par l'expression *opérations de commande*.

Ces opérations peuvent être utilisées aux accès de communication d'un composant de test comme résumé dans le Tableau 15. En cas d'accès mixtes, toutes les opérations sont applicables.

Tableau 15/Z.140 – Aperçu général des opérations de communication TTCN-3

Opérations de communication			
Opération de communication	**Mot clé**	**Peut être utilisé à un accès en mode message**	**Peut être utilisé à un accès en mode procédure**
Opérations d'envoi			
Envoi de message	**`send`**	Oui	
Invocation d'appel de procédure	**`call`**		Oui
Réponse à appel de procédure issu d'entité distante	**`reply`**		Oui
Déclenchement d'exception (à un appel accepté)	**`raise`**		Oui
Opérations de réception			
Réception de message	**`receive`**	Oui	
Déclenchement sur message	**`trigger`**	Oui	

Tableau 15/Z.140 – Aperçu général des opérations de communication TTCN-3

Opérations de communication			
Opération de communication	**Mot clé**	**Peut être utilisé à un accès en mode message**	**Peut être utilisé à un accès en mode procédure**
Acceptation d'appel de procédure issu d'entité distante	`getcall`		Oui
Obtention de réponse issue d'un appel précédent	`getreply`		Oui
Acquisition d'exception (issue d'entité appelée)	`catch`		Oui
Vérifier message/appel/exception/réponse reçu	`check`	Oui	Oui
Opérations de commande			
Libérer accès	`clear`	Oui	Oui
Libérer et utiliser l'accès	`start`	Oui	Oui
Arrêt d'utilisation d'un accès (réception & émission)	`stop`	Oui	Oui

22.1 Opérations d'envoi

Les opérations d'envoi sont les suivantes:

a) `send`: envoi d'un message en mode asynchrone;

b) `call`: appel d'une procédure;

c) `reply`: réponse à un appel de procédure accepté issu du système SUT;

d) `raise`: déclenchement d'une exception si un appel de procédure est reçu.

22.1.1 Format général des opérations d'envoi

Les opérations d'envoi se composent d'une partie *envoi* et, dans le cas de l'opération `call` en mode procédure, d'une partie *réponse* et d'une partie *traitement des exceptions*.

La partie *envoi*:

- spécifie l'accès auquel l'opération spécifiée doit être effectuée;
- définit la valeur des informations à transmettre;
- fournit une expression d'adresse (facultative) qui désigne de manière univoque la correspondance de communication en cas de connexion point-multipoint.

Le nom de l'accès, le nom de l'opération et la valeur doivent être présents dans toutes les opérations d'envoi. L'identification du correspondant de communication (indiqué par le mot clé `to`) est facultative et n'a besoin d'être spécifiée qu'en cas de connexions point-multipoint où l'entité réceptrice doit être définie explicitement.

22.1.1.1 Traitement des réponses et des exceptions

Le traitement des réponses et des exceptions n'est nécessaire que lors d'une communication synchrone. La partie traitement des réponses et des exceptions de l'opération `call` est facultative. Elle n'est requise que lorsque la procédure appelée renvoie une valeur ou possède des paramètres `out` ou `inout` dont les valeurs sont nécessaires dans le composant appelant ainsi que lorsque la procédure appelée peut déclencher des exceptions devant être traitées par le composant appelant.

La partie traitement des réponses et exceptions de l'opération d'appel fait appel aux opérations **getreply** et **catch** pour offrir la capacité requise.

22.1.2 L'opération d'envoi

L'opération **send** sert à placer une valeur dans la file d'attente d'un accès de messagerie sortante. Cette valeur peut être spécifiée par référencement d'un modèle, d'une variable ou d'une constante. Elle peut également être définie en ligne à partir d'une expression (qui peut évidemment être une valeur explicite). Lors de la définition en ligne de la valeur, le champ facultatif de type doit être utilisé s'il y a ambiguïté concernant le type de la valeur à envoyer.

L'opération **send** ne doit être utilisée qu'aux accès en mode message (ou mixtes) et le type de la valeur à envoyer doit figurer dans la liste des types sortants de la définition de type de l'accès. Par exemple:

```
MyPort.send(MyTemplate(5,MyVar));
// Envoie le modèle MyTemplate avec les paramètres réels 5 et MyVar via
// MyPort.

MyPort.send(integer:5);
// Envoie la valeur d'entier 5
```

En cas de connexions point à multipoint, le correspondant de communication doit être spécifié de façon univoque. Cela doit être indiqué par le mot clé **to**. Par exemple:

```
MyPort.send("My string") to MyPartner;
// Envoie la chaîne "My string" vers un composant avec une référence de
// composant mémorisée dans la variable MyPartner.

MyPCO.send(MyVariable + YourVariable - 2) to MyPartner;
// Envoie le résultat de l'expression arithmétique à MyPartner.
```

22.1.3 L'opération d'appel

L'opération **call** sert à spécifier le fait qu'un composant de test appelle une procédure dans le système SUT ou dans un autre composant de test. L'appel est une opération bloquante en ce sens qu'elle doit attendre une réponse (c'est-à-dire un message **reply**) ou une exception en provenance de l'entité appelée. En d'autres termes, l'opération d'appel fonctionne de façon synchrone.

NOTE – Cela est comparable à l'essai d'une capacité de serveur, c'est-à-dire que le système SUT est le serveur et que le composant joue le rôle d'un client.

L'opération **call** ne doit être utilisée qu'aux accès en mode procédure (ou mixtes). La définition de type de l'accès auquel l'opération d'appel intervient doit inclure le nom de la procédure dans sa liste **in** ou **inout**, c'est-à-dire qu'il doit être permis d'appeler cette procédure à cet accès.

La valeur de l'opération **call** est une signature qui peut soit être définie sous la forme d'un modèle de signature soit être définie en ligne. Par exemple:

```
signature MyProc (out integer MyPar1, inout boolean MyPar2);
:
MyPort.call(MyProc:{MyVar1,MyVar2});
// Appelle la procédure distante MyProc at MyCL avec les paramètres "in" et
// "inout": 5 et MyVar. Ni une valeur de retour ni une exception n'est attendue
// en provenance de cet appel. Si un des deux ou les deux paramètres sont
// définis comme étant de type "inout", leur valeur ne sera pas prise en
// considération c'est-à-dire qu'elle ne sera pas affectée à une variable.

// L'exemple suivant explique les possibilités d'affecter des valeurs aux
// paramètres "in" et "inout" dans l'argument d'appel. La signature suivante
// est supposée présente pour la procédure à appeler.
// A noter que MyProc2 n'a pas de valeur de retour et pas d'exceptions
```

```
signature MyProc2 (in integer A, out integer B, inout integer C);
:
MyPort.call(MyProc2:{1, - , 3});
// Seules les valeurs de paramètres "in" et "inout" sont spécifiées. Les
// valeurs renvoyées des paramètres out et inout sont inutilisées après
// la communication et ne sont donc pas affectées à des variables.
```

Tous les paramètres **in** et **inout** de la signature doivent avoir une valeur spécifique, c'est-à-dire que l'emploi de mécanismes d'appariement comme *AnyValue* n'est pas autorisé.

Les arguments de signature de l'opération **call** ne sont pas utilisés afin d'extraire des noms de variable pour les paramètres **out** et **inout**. L'affectation réelle à des variables de la valeur de retour de procédure et des valeurs des paramètres **out** et **inout** doit être faite explicitement dans la partie obtention de réponse (**getreply**) et d'exception (**catch**) de l'opération **call**. C'est ce qui est indiqué respectivement par les mots clés **value** et **param**. Cela permet d'utiliser des modèles de signature dans les opérations d'appel **call** comme l'on peut utiliser des modèles pour des types.

En général, une opération d'appel **call** est censée avoir une sémantique bloquante. La notation TTCN-3 prend cependant en charge les appels non bloquants. Un appel exempt de valeurs de retour est censé être de type non bloquant. Les exceptions (si spécifiées) déclenchées par un appel sans valeurs de retour doivent être traitées à l'intérieur d'une déclaration **alt** subséquente. Il est également possible de forcer la sémantique non bloquante par le mot clé **nowait** (voir § 22.1.3.3).

En cas de connexions point à multipoint, le correspondant de communication doit être spécifié de façon univoque. Cela doit être indiqué au moyen du mot clé **to**. Par exemple:

```
MyPort.call(MyProc:{MyVar1, MyVar2}) to MyPartner;
// Dans cet exemple l'appelé est explicitement identifié par la référence de
// composant mémorisée dans la variable MyPartner.
```

22.1.3.1 Traitement des réponses à un appel

Le traitement de la réponse à un appel est effectué au moyen de l'opération **getreply** (voir § 22.2.5). Cette opération définit le comportement en variante en fonction de la réponse obtenue à la suite de l'opération d'appel. Par exemple:

```
MyPort.call(MyProc:{MyVar1, MyVar2}) to MyPartner
                              // Où { ... } est un modèle en ligne
{
     [] MyCl.getreply(MyProc:{MyVar1, MyVar2}) {}
}
```

Si nécessaire, la valeur de retour de la procédure appelée doit être insérée explicitement dans l'opération **getreply**, ce qui est exprimé au moyen des caractères **'->'** et du mot clé (facultatif) **value.** Par exemple:

```
MyPort.call(MyProc:{MyVar1, MyVar2}) to MyPartner
{
     [] MyCl.getreply(MyProc:{MyVar1, MyVar2}) -> value MyResult {}
}
// Une valeur doit être renvoyée par MyProc qui sera mémorisée dans la
// variable MyResult.
```

Les arguments de signature de l'opération **call** ne sont pas utilisés pour extraire des noms de variable pour les paramètres **out** et **inout**. L'affectation réelle à des variables de la valeur de retour de procédure et des valeurs des paramètres **out** et **inout** doit être faite explicitement dans la partie obtention de réponse (**getreply**) et acquisition d'exception de l'opération **call**. C'est ce qui est indiqué respectivement par les mots clés **value** et **param**. Cela permet d'utiliser des modèles de signature dans les opérations d'appel comme l'on peut utiliser des modèles pour des types. Par exemple:

```
MyPort.call(MyProc:{5,MyVar}) to MyPartner
{
     []MyCl.getreply(MyProc:{MyVar1, MyVar2}) -> value MyResult param
       (MyPar1Var,MyPar2Var) {}
}
// Dans cet exemple les deux paramètres de MyProc sont spécifiés comme
// paramètres "inout" et leurs valeurs après la fin de MyProc sont affectées
// à MyPar1Var et MyPar2Var.
```

22.1.3.2 Traitement des exceptions à un appel

Le traitement des exceptions à un appel est effectué au moyen de l'opération **catch** (voir § 22.2.6). Cette opération définit le comportement en variante en fonction de l'exception qui a (éventuellement) été déclenchée par l'opération **call**. Par exemple:

```
signature MyProc3 (out integer MyPar1, inout boolean MyPar2) return MyResultType
     exception (ExceptionTypeOne, ExceptionTypeTwo, ExceptionTypeThree);
:
// Ci-après, l'opération "call" montre le mécanisme d'obtention de réponse
// ("getreply") et d'acquisition d'exception de l'opération "call"
MyPort.call(MyProc3:{5,MyVar}, 30E-3) to MyPartner
{
     []   MyCl.getreply(MyProc3:{MyVar1, MyVar2}) ->  value MyResult param
                                                       (MyPar1Var,MyPar2Var) {}
     []   MyPort.catch(MyProc3, MyExceptionOne)
       {                                  // Traite une exception
          verdict.set(fail);              // Fixe le verdict et
          stop                            // s'arrête à la suite de l'exception
       }
     []   MyPort.catch(MyProc3, MyExceptionTwo) // Traite une seconde exception
       {verdict.set(inconc)}                    // Fixe le verdict et continue
                                                // après l'appel à la suite de
                                                // la seconde exception

     [MyCondition] MyPort.catch(MyProc3, MyExceptionThree) {}
                                 // Traite une troisième exception qui peut se
                                 // produire si MyCondition donne la valeur
                                 // "true".

     []   MyPort.catch(timeout) {}     // Exception de temporisation c'est-à-dire
                                       // que l'appelé ne réagit pas à temps et
                                       //que rien n'est effectué
}
```

22.1.3.3 Traitement des exceptions de temporisation d'un appel

L'opération **call** peut (en option) comporter une temporisation. Cela est défini par une valeur ou constante explicite de type **float**. L'on définit également la durée pendant laquelle une exception **timeout** doit être produite par le système de test une fois que l'opération **call** a commencé. Si aucune partie contenant une valeur de temporisation n'est présente dans l'opération **call**, aucune exception de type **timeout** ne doit être produite. Par exemple:

```
MyPort.call(MyProc:{5,MyVar}, 20E-3)
{
     []   MyPort.catch(timeout)
       {
          verdict.set(fail);
          stop
       }
}
// Cet exemple montre un appel avec une valeur de temporisation de 20 ms.
// Cela signifie que si l'appelé ne renvoie pas une réponse ou une exception
// dans cet intervalle de temps, le système de test produira automatiquement
```

```
// une exception de temporisation. Le traitement de la temporisation est
// effectué au moyen d'une opération de traitement. Si la procédure s'achève
// sans exception de temporisation, l'exécution continuera avec les
// instructions qui suivent l'opération "call".
```

L'utilisation du mot clé **nowait** dans la partie contenant la valeur de temporisation d'une opération **call** permet d'appeler une procédure sans attendre une terminaison ou une réponse ou le déclenchement d'une exception par la procédure appelée ou le déclenchement d'une exception de temporisation. Par exemple:

```
MyPort.call(MyProc:{5, MyVar}, nowait);
// Dans cet exemple le composant de test continuera l'exécution sans attendre la
// terminaison de MyProc.
```

Dans de tels cas, une réponse ou exception possible doit être extraite de la file d'attente au moyen d'une opération **getreply** ou **catch** contenue dans une instruction **alt** subséquente.

22.1.4 L'opération de réponse

L'opération **reply** sert à répondre à un appel déjà accepté conformément à la signature de procédure. Cette opération ne doit être utilisée qu'à un accès en mode procédure (ou mixte). La définition du type d'accès doit comporter le nom de la procédure à laquelle l'opération de réponse **reply** appartient.

La partie valeur de l'opération de réponse se compose d'une référence de signature avec une liste associée des paramètres réels et une valeur de retour (facultative). La signature peut être définie sous la forme d'un modèle de signature ou être définie en ligne. Tous les paramètres **out** et **inout** de la signature doivent avoir une valeur spécifique, c'est-à-dire que l'utilisation de mécanismes d'appariement tels que *AnyValue* n'est pas autorisée. Par exemple:

```
MyPort.reply(MyProc2:{ - ,5});
// Réponses à un appel accepté de MyProc2. La procédure MyProc2 n'a pas de
// valeur de retour mais deux paramètres.
// Le premier paramètre est un paramètre de type "in" c'est-à-dire que sa
// valeur ne fera pas l'objet d'une réponse et donc n'a pas besoin d'être
// spécifiée. Le second paramètre est de type "out" ou "inout". Sa valeur
// est 5.
```

En cas de connexions point à multipoint, le correspondant de communication doit être spécifié explicitement et doit être unique. Cela doit être indiqué par le mot clé **to**. Par exemple:

```
MyPort.reply(MyProc3:{ - ,5}) to MyPartner;
// Cet exemple est identique au précédent, mais la réponse est dirigée vers
// un composant avec une référence de composant mémorisée dans la variable
// MyPartner
```

Si une valeur doit être renvoyée à l'appelant, cela doit être explicitement indiqué au moyen du mot clé **value**.

```
MyPort.reply(MyProc:{5,MyVar} value 20);
// Réponses à un appel accepté de MyProc. La valeur de retour de MyProc
// est 20 et possède deux paramètres qui sont des paramètres "out" ou
// "inout". Leurs valeurs sont fournies par 5 et MyVar.
```

22.1.5 L'opération de déclenchement d'une exception

L'opération **raise** sert à déclencher une exception. Une exception ne doit être déclenchée qu'à un accès en mode procédure (ou mixte). Une exception est une réaction à un appel de procédure accepté, dont le résultat conduit à un événement exceptionnel. Le type de l'exception doit être spécifié dans la signature de la procédure appelée. La définition du type d'accès doit comporter,

dans sa liste d'appels de procédure acceptés, le nom de la procédure à laquelle l'exception appartient.

NOTE – La relation entre un appel accepté et une opération **raise** ne peut pas toujours être vérifiée statiquement. Pour les essais, il est permis de spécifier une opération **raise** sans opération **getcall** associée.

La partie valeur de l'opération **raise** se compose d'une référence de signature suivie de la valeur d'exception. Par exemple:

```
MyPort.raise(MySignature, MyVariable + YourVariable – 2);
// Déclenche une exception avec une valeur qui est le résultat de
// l'expression arithmétique à MyPort
```

Les exceptions sont spécifiées sous la forme d'un type. La valeur d'exception peut donc être extraite d'un modèle ou résulter d'une expression (qui peut évidemment être une valeur explicite). Le champ de type facultatif dans la spécification de valeur relative à l'opération **raise** doit être utilisé s'il est nécessaire d'éviter toute ambiguïté quant au type de valeur envoyé. Par exemple:

```
MyPort.raise(MyProc, MyExceptionType:{5, MyVar});
// Déclenche une exception à partir de la procédure distante définie par
// Myproc avec les valeurs définies par le modèle MyExceptionTemplate avec
// les paramètres réels 5 et MyVar à un accès MyPort.
```

En cas de connexions point à multipoint, le correspondant de communication doit être spécifié de manière univoque. Cela doit être indiqué par le mot clé **to**. Par exemple:

```
MyPort.raise(MySignature, "My string") to MyPartner;
// Déclenche une exception de type chaîne avec la valeur "My string" à
// l'accès MyPort vers un composant dont la référence est mémorisée dans la
// variable MyPartner
```

22.2 Opérations de réception

Les opérations de réception sont les suivantes:

a) **receive**: réception d'un message envoyé en mode asynchrone;

b) **trigger**: déclenchement de la réception d'un message spécifique;

c) **getcall**: acceptation d'un appel de procédure;

d) **getreply**: traitement de la réponse à une procédure déjà appelée;

e) **catch**: traite une exception qui a été déclenchée en réaction à une opération "call";

f) **check**: vérifier l'élément sommital de la file d'entrée d'un accès particulier.

22.2.1 Format général des opérations de réception

L'opération de réception se compose d'une partie *réception* et d'une partie *affectation*.

La partie *réception*:

a) spécifie l'accès auquel l'opération doit avoir lieu;

b) définit une partie appariement qui spécifie l'entrée acceptable qui s'appariera à l'instruction;

c) fournit une expression (facultative) d'adressage qui désigne de manière univoque le correspondant de communication (en cas de connexions point à multipoint).

Le nom de l'accès, le nom de l'opération et la partie *valeur* de toutes les opérations de réception doivent être présents. L'identification du correspondant de communication (indiqué par le mot clé **from**) est facultative et n'a besoin d'être spécifiée qu'en cas de connexions point à multipoint où l'entité réceptrice a besoin d'être explicitement identifiée.

22.2.1.1 Exécution d'affectations aux opérations de réception

La partie *affectation* est facultative dans une opération de réception. Aux accès en mode message, elle est utilisée s'il faut mémoriser des messages reçus. Aux accès en mode procédure, elle sert à mémoriser les paramètres **in** et **inout** d'un appel accepté ou à mémoriser des exceptions.

Par ailleurs, la partie *affectation* peut également être utilisée pour affecter à une variable l'adresse de l'expéditeur **sender** d'un message, une exception, une réponse **reply** ou un appel **call**. Cela est utile pour les connexions point à multipoint où, par exemple, le même message ou appel peut être reçu de différents composants. Mais le message, la réponse ou l'exception doit toujours faire l'objet d'un renvoi au composant expéditeur original.

22.2.2 L'opération de réception

L'opération **receive** sert à recevoir une valeur issue d'une file de messages à un accès entrant. Cette valeur peut être spécifiée par référence à un modèle, à une variable ou à une constante. Elle peut également être définie en ligne à partir d'une expression (qui peut évidemment être une valeur explicite). Lors de la définition de la valeur en ligne, le champ facultatif de type doit être utilisé afin d'éviter toute ambiguïté quant au type de la valeur à recevoir. L'opération de réception ne doit être utilisée qu'à des accès en mode message (ou mixtes) et le type de la valeur à recevoir doit être inclus dans la liste des types entrants de la définition du type d'accès.

L'opération de réception retire le message sommital de la file d'accès entrant associée si, et seulement si ce message sommital répond à tous les critères d'appariement associés à l'opération de réception. Aucune association des valeurs entrantes aux termes de l'expression ou au modèle ne doit se produire.

Si l'appariement n'est pas obtenu, le message sommital ne doit pas être retiré de la file d'accès. En d'autres termes, si l'opération de réception n'est pas correctement effectuée, l'exécution du test élémentaire doit se poursuivre par la variante suivante.

Les critères d'appariement sont associés au type et à la valeur du message à recevoir. Le type et la valeur du message à recevoir peuvent soit être déduits d'un modèle soit avoir la valeur donnée par une expression (qui peut évidemment être une valeur explicite).

```
MyPort.receive(MyTemplate(5, MyVar));
// Spécifie la réception d'une valeur qui répond aux conditions définies par
// le modèle MyTemplate avec les paramètres réels 5 et MyVar.

MyPort.receive(A<B);
// Spécifie la réception de la valeur booléenne "true" ou "false" selon le
// résultat de A<B
```

Un champ de type facultatif doit être utilisé dans les critères d'appariement de l'opération de réception afin d'éviter toute ambiguïté quant au type de la valeur à recevoir. Par exemple:

```
MyPort.receive(integer:MyVar);
// Spécifie la réception d'un entier qui possède la même valeur que la
// variable MyVar à MyPort. L'entier (facultatif) identifiant le type n'est
// pas strictement nécessaire parce que le type est déjà donné par la
// définition de MyVar. Cependant, dans des tests élémentaires longs et
// complexes un tel identificateur de type peut être utilisé afin d'améliorer
// la lisibilité.

MyPort.receive(MyVar);
// Est une variante de l'exemple précédent.
```

Si l'appariement est obtenu, la valeur extraite de la file d'accès peut être mémorisée dans une variable et l'adresse du composant qui a envoyé le message peut être extraite et mémorisée dans une variable. C'est ce qui est indiqué par le symbole '->' et par le mot clé **value**. Par exemple:

```
MyPort.receive(MyType:*) from MyPartner -> value MyVar;
// Spécifie la réception d'une valeur arbitraire de MyType (issue d'un
// composant avec une adresse mémorisée dans la variable MyPartner) qui
// ensuite est affectée à la variable MyVar. MyVar doit être du type MyType.
```

Dans le cas des connexions point à multipoint, l'opération de réception peut être limitée à un certain correspondant de communication. Cette restriction doit être indiquée au moyen du mot clé **from**.

```
MyPort.receive(charstring:"Hello")from MyPartner;
// Spécifie la réception de la chaîne de caractères "Hello" issue d'un
// composant avec une référence de composant ou une adresse mémorisée dans la
// variable MyPartner.
```

Il est également possible d'extraire la référence de composant ou l'adresse de l'expéditeur d'un message. Cela est indiqué par le mot clé **sender**. Par exemple:

```
MyPort.receive(MyTemplate:{5, MyVarOne}) -> value MyVarTwo sender MyPartner;
// Spécifie la réception d'une valeur qui répond aux conditions définies par
// le modèle MyTemplate avec les paramètres réels 5 et MyVarOne. Après
// réception, la valeur est affectée à la variable MyVarTwo. La référence du
// composant expéditeur est extraite par l'opération "call" et affectée à la
// variable MyPartner.

MyPort.receive(A<B) -> sender MyPartner;
// Spécifie la réception de la valeur booléenne "true" ou "false" selon le
// résultat de A<B.
// La référence de composant du composant expéditeur est extraite par
// l'opération "call" et affectée à la variable MyPartner.
```

22.2.2.1 Réception de tout message

Une opération de réception sans liste d'arguments pour le type et les critères d'appariement de valeur du message à recevoir doit supprimer le message situé au sommet de la file d'accès entrant (éventuelle) si tous les autres critères d'appariement sont satisfaits.

NOTE – Cette opération est équivalente à l'instruction OTHERWISE de la notation TTCN-2.

Un message reçu par l'opération *ReceiveAnyMessage* ne doit pas être affecté à une variable.

Exemple:

```
MyPort.receive;
// Retire la valeur sommitale de MyPort.MyPort.

MyPort.receive from MyPartner;
// Retire la valeur sommitale de CL1 s'il s'agit d'un message issu du
// composant avec les références d'adresse

MyPort.receive -> sender MySenderVar;
// Retire la valeur sommitale de CL1, mais mémorise l'instance expéditrice
// par mise en mémoire de sa référence dans la variable MySenderVar
```

22.2.2.2 Réception sur tout accès

Le mot clé **any** est utilisé pour recevoir un message sur tout accès. Par exemple:

```
any port.receive(MyMessage);
```

22.2.3 L'opération de déclenchement

L'opération **trigger** filtre les messages possédant certains critères d'appariement à partir d'un flux de messages reçus par un accès entrant donné. L'opération de déclenchement **trigger** ne doit être utilisée qu'aux accès en mode message (ou mixte) et le type de la valeur à recevoir doit être inclus

dans la liste des types entrants de la définition du type d'accès. Tous les messages qui ne répondent pas aux critères d'appariement doivent être supprimés de la file d'attente sans autre action, c'est-à-dire que l'opération de déclenchement attend le prochain message de cette file. Si un message répond aux critères d'appariement, l'opération de déclenchement donne le même comportement que l'opération de réception. Par exemple:

```
MyPort.trigger(MyType:*);
// Spécifie que l'opération déclenchera la réception du premier message
// observé du type MyType avec une valeur arbitraire à un accès MyPort.
```

L'opération **trigger** nécessite le nom de l'accès, les critères d'appariement pour le type et la valeur, une restriction **from** facultative (c'est-à-dire la sélection du correspondant de communication) et une attribution facultative à des variables du composant correspondant de message et d'expéditeur.

Exemple:

```
MyPort.trigger(MyType:*) from MyPartner;
// Spécifie que l'opération déclenchera la réception du premier message
// observé du type MyType avec une valeur arbitraire à un accès MyCL entrant
// issue d'un composant avec une référence identique à celle qui est
// mémorisée dans la variable MyPartner.

MyPort.trigger(MyType:*) from MyPartner -> value MyRecMessage;
// Cet exemple est presque identique au précédent exemple. Le complément est
// que le message qui déclenche (parce que tous les critères d'appariement
// sont satisfaits) est mémorisé dans la variable MyRecMessage.

MyPort.trigger(MyType:*) -> sender MyPartner;
// Spécifie que l'opération déclenchera la réception du premier message
// observé du type MyType avec une valeur arbitraire à MyPort. La
// référence du composant expéditeur de ce message sera mémorisée dans la
// variable MyPartner.

MyPort.trigger(integer:*) -> value MyVar sender MyPartner;
// Spécifie que l'opération déclenchera la réception d'un entier arbitraire
// qui ensuite est mémorisé dans la variable MyVar et la référence du
// composant expéditeur de ce message sera mémorisée dans la variable
// MyPartner.
```

22.2.3.1 Déclenchement à tout message

Une opération **trigger** sans liste d'arguments doit effectuer un déclenchement à la réception d'un message quelconque. Sa signification est donc identique à celle de la réception d'un message quelconque. Un message reçu par l'opération *TriggerOnAnyMessage* ne doit pas être affecté à une variable.

Exemple:

```
MyPort.trigger;

MyPort.trigger from MyPartner;

MyPort.trigger -> sender MySenderVar;
```

22.2.3.2 Déclenchement à tout accès

Le mot clé **any** est utilisé pour effectuer un déclenchement par message sur accès quelconque. Par exemple:

```
any port.trigger
```

22.2.4 L'opération d'obtention d'appel

L'opération **getcall** sert à spécifier qu'un composant de test accepte un appel issu du système SUT ou un autre composant de test. Cette opération ne doit être utilisée qu'aux accès en mode procédure (ou mixtes) et la signature de l'appel de procédure doit être incluse dans la liste des procédures entrantes autorisées de la définition du type d'accès.

```
MyPort.getcall(MyProc(5, MyVar));
// Acceptera un appel de MyProc à MyCL avec les paramètres "in" ou "inout" 5
// et la valeur de MyVar.
```

L'opération **getcall** doit supprimer l'appel situé au sommet de la file d'attente d'accès entrant si, et seulement si, les critères d'appariement associés à cette opération sont satisfaits. Ces critères d'appariement sont associés à la signature de l'appel à traiter et au correspondant de communication. Les critères d'appariement peuvent être spécifiés en ligne ou être déduits d'un modèle de signature.

Une opération **getcall** peut être limitée à un certain correspondant de communication en cas de connexions point à multipoint. Cette restriction doit être indiquée au moyen du mot clé **from**.

```
MyPort.getcall(MyProc:{5, MyVar}) from MyPartner;
// Acceptera un appel de MyProc à MyCL (avec les paramètres "in" ou "inout" 5
// et la valeur de MyVar) en provenance d'une entité homologue avec l'adresse
// ou la référence de composant mémorisée dans la variable MyPartner.
```

La partie affectation de l'opération **getcall** se compose de l'affectation facultative à des variables des valeurs de paramètres **in** et **inout** ainsi que de l'extraction et de l'affectation à une variable de l'adresse du composant appelant.

Le mot clé **param** est utilisé pour extraire les valeurs paramétriques d'un appel. Par exemple:

```
MyPort.getcall(MyProc:{5, MyVar}) from MyPartner -> param (MyPar1Var,
MyPar2Var);
// Les deux paramètres de MyProc sont des paramètres "inout" et leurs
// valeurs sont affectées à MyPar1Var et MyPar2Var. L'identification de
// paramètres définis dans la signature de procédure et les noms contenus
// dans la liste des noms de variable qui suivent le mot clé "param" dans
// l'opération d'acceptation ci-dessus est effectuée dans l'ordre de la
// liste
```

Le mot clé **sender** est utilisé lorsqu'il est nécessaire d'extraire l'adresse de l'expéditeur (par exemple afin d'adresser une réponse **reply** ou une exception à l'appelant dans une configuration point à multipoint).

```
MyPort.getcall(MyProc:{5, MyVar}) -> sender MySenderVar;
// Acceptera un appel de MyProc à MyCL avec les paramètres "in" ou "inout" 5
// et MyVar. L'appelant est extrait par l'opération d'acceptation et est
// mémorisé dans la variable MySenderVar. Cela permet de traiter un appel de
// la même procédure issu de plusieurs composants au même accès de la même
// façon. MySenderVar Peut être utilisée pour réponse ou déclenchement d'une
// exception vers le composant appelant.
```

L'argument de signature de l'opération **getcall** ne doit pas être utilisé pour transmettre des noms de variable dans des paramètres **in** et **inout**. L'affectation à des variables de valeurs de paramètres **in** et **inout** doit être faite dans la partie affectation de l'opération **getcall**. Cela permet d'utiliser des modèles de signature dans les opérations **getcall** comme des modèles sont utilisés pour des types.

Les opérations **getcall** ci-après montrent les possibilités d'utilisation d'attributs d'appariement et d'omission de parties facultatives, ce qui peut n'avoir aucune importance pour la spécification du test.

Exemple:

```
MyPort.getcall(MyProc:{5, MyVar}) -> param(MyPar1Var, MyPar2Var) sender
MySenderVar;

MyPort.getcall(MyProc:{5, *}) -> param(MyPar1Var, MyPar2Var);

MyPort.getcall(MyProc:{*, MyVar}) -> param( - , MyPar2Var);
// La valeur du premier paramètre "inout" n'est pas importante ou est
// inutilisée

// L'exemple suivant doit expliquer les possibilités d'affecter à des
// variables des valeurs de paramètre "in" ou "inout". La signature suivante
// est supposée présente pour la procédure à appeler.


signature MyProc2(in integer A, integer B, integer C, out integer D, integer E,
inout integer F);
     // MyProc2 n'a pas de valeur de retour et pas d'exceptions

MyPort.getcall(MyProc2:{*, *, 3, - , - , *}) ->
          param(MyVarIn1, MyVarIn2, MyVarIn3, - , - ,MyVarInout1);
     // Les paramètres de type "in" A, B et C sont affectés aux variables
     // MyVarIn1, MyVarIn2 et MyVarIn3 le paramètre "inout" F est affecté à la
     // variable MyVarInout1. Les paramètres "out" D et E n'ont pas besoin
     // d'être pris en considération dans la partie affectation de l'opération
     // d'acceptation.

MyPort.getcall(MyProc2:{*, *, *, - , - , *}) ->
          param(MyVarIn1:=A, MyVarIn2:=B, MyVarIn3:=C, MyVarInout1:=F);
     // Notation en variante pour l'affectation à des variables de la valeur de
     // paramètres "in" ou "inout".
     // A noter que les noms contenus dans la liste d'affectation se rapportent
     // aux noms utilisés dans la signature de MyProc2.

MyPort.getcall(MyProc2:{1, 2, 3, - , - ,*}) -> param(MyVarInout1:=F);
     // Seule la valeur du paramètre "inout" est nécessaire pour l'exécution du
     // test élémentaire suivant.
```

22.2.4.1 Acceptation de tout appel

Une opération **getcall** sans liste d'arguments pour les critères d'appariement de signature supprimera l'appel situé au sommet de la file d'attente (éventuelle) d'un accès entrant, si tous les autres critères d'appariement sont remplis. Les paramètres des appels acceptés par *AcceptAnyCall* ne doivent pas être affectés à une variable.

Exemple:

```
MyPort.getcall;
// Supprime l'appel sommital de MyPort.

MyPort.getcall from MyPartner;
// Supprime l'appel sommital de CL1 si l'appelant est une entité avec une
// adresse ou une référence de composant mémorisée dans la variable
// MyPartner.

MyPort.getcall -> sender MySenderVar;
// Supprime l'appel sommital de CL1, mais mémorise l'appelant par mise en
// mémoire de son adresse ou de la référence de composant contenue dans
// MySenderVar
```

22.2.4.2 Obtention d'appel à tout accès

L'opération **getcall** à tout accès est indiquée par le mot clé **any**. Par exemple:

```
any port.getcall(MyProc)
```

22.2.5 L'opération d'obtention de réponse

L'opération **getreply** sert à traiter les réponses issues d'une procédure déjà appelée. Cette opération ne doit être utilisée qu'à un accès en mode procédure (ou mixte). Par exemple:

```
MyPort.getreply(MyProc:{5, MyVar} value 20);
// Accepte une réponse de la procédure MyProc où la valeur renvoyée est 20
// et où les valeurs des deux paramètres "out" ou "inout" sont 5 et la
// valeur de MyVar.

MyPort.getreply(MyProc2:{ - , 5});
// Accepte une réponse de MyProc2. MyProc2 n'a pas de valeur de retour mais
// deux paramètres. Le premier paramètre est un paramètre de type "in"
// c'est-à-dire que sa valeur ne fera pas l'objet d'une réponse et donc ne
// sera pas prise en considération pour l'appariement. Le second paramètre
// est soit un paramètre de type "out" ou "inout". Sa valeur doit être 5.
```

Cette opération peut être utilisée soit dans la partie **getreply** et exception d'un appel, par exemple:

```
MyPort.call (MyProc) to MyPeer
{
     [ ] MyPort.getreply(MyProc:*) {}
     [ ] MyPort.catch {}
}
```

ou à l'intérieur d'une instruction **alt**, par exemple:

```
MyPort.call (MyProc, nowait) to MyPeer;
:
alt
{
     [ ] MyPort.getreply(MyProc:*) {}
     :
}
```

Si elle est utilisée dans une instruction **alt**, l'opération **getcall** devrait couvrir les cas où la réponse d'une procédure déjà appelée arrive trop tard, c'est-à-dire qu'une exception de temporisation a été déclenchée.

Comme dans le cas des autres opérations de réception, des mécanismes d'appariement sont autorisés dans l'opération **getreply** afin d'établir une distinction entre les réponses issues d'une procédure déjà appelée, qui diffèrent soit par leur valeur renvoyée et/ou par la valeur de paramètres **in** et **inout**.

```
MyPort.getreply(MyProc1:{*, MyVar});
// Dans cet exemple, il n'y a pas de restriction quant à la valeur renvoyée
// et quant à la valeur du premier paramètre.

MyPort.getreply(MyProc1:{*, *});
// L'opération getreply apparie toute valeur d'entier de toute réponse de
// MyProc1 avec toute valeur renvoyée. Les astérisques sont des définitions
// de modèle en ligne for MyProc1 et le type de renvoi de MyProc1.
```

En cas de connexions point à multipoint, l'opération **getreply** permet d'établir une distinction entre différents correspondants de communication au moyen d'un mot clé **from**.

```
MyPort.getreply(MyProc2:{ - ,5}) from MyPartner;
// La réponse n'est acceptée que si elle est issue d'un composant ayant la
// référence spécifiée dans la variable MyPartner
```

La partie facultative d'affectation de l'opération **getreply** permet d'affecter à des variables des valeurs des paramètres **out** et **inout** ainsi que des valeurs renvoyées.

Exemple:

```
MyPort.getreply(MyProc1:{*, *} value *) -> value MyReturnValue
         param(MyPar1,MyPar2);
// Après acceptation, la valeur renvoyée est affectée à la variable
// MyReturnValue et la valeur des deux paramètres "out" ou "inout" est
// affectée aux variables MyPar1 et MyPar2.


MyPort.getreply(MyProc1:{*, *} value *) -> value MyReturnValue param( - ,
MyPar2)
     sender MySender;
// La valeur du premier paramètre n'est pas prise en considération pour
// l'exécution du test suivant et l'adresse ou référence de composant de
// l'entité  de laquelle la réponse a été reçue est mémorisée dans la
// variable MySender.

// Les exemples suivants décrivent certaines possibilités d'affecter à des
// variables des valeurs de paramètre "out" et "inout"
// La signature suivante est supposée présente pour la procédure qui a été
// appelée
signature MyProc2(in integer A, integer B, integer C, out integer D, integer E,
     inout integer F);
// A noter que MyProc2 n'a pas de valeur de retour et pas d'exceptions.

MyPort.getreply(MyProc2:*) -> param( - , - , - , MyVarOut1, MyVarOut2, - ,
     MyVarInout1);
// Les paramètres D et E, de type "in" sont affectés aux variables MyVarOut1
// et MyVarOut2. Le paramètre inout F est affecté à la variable MyVarInout1.

MyPort.getreply(MyProc2:*) -> param(MyVarOut1:=D, MyVarOut2:=E, MyVarInout1:=F);
// Notation en variante pour l'affectation à des variables de la valeur de
// paramètres "in" ou "inout".
// A noter que les noms contenus dans la liste d'affectation se rapportent
// aux noms utilisés dans la signature de MyProc2

MyPort.getreply(MyProc2:{ - , - , - , 3, *, *}) -> param(MyVarInout1:=F);
// Seule la valeur du paramètre "inout" est nécessaire pour l'exécution du
// test élémentaire suivant
```

22.2.5.1 Acceptation de toute réponse à tout appel

Une opération **getreply** sans liste d'arguments pour les critères d'appariement de signature doit supprimer un message de réponse au sommet de la file d'attente (éventuelle) de l'accès entrant si tous les autres critères d'appariement sont satisfaits. Les paramètres ou valeurs de retour des réponses acceptées par une opération *GetAnyReply* ne doivent pas être attribués à une variable.

Exemple:

```
MyPort.getreply;
// Supprime la réponse sommitale de MyPort.

MyPort.getreply from MyPartner;
// Supprime la réponse sommitale de CL1 si le répondeur est une entité avec
// l'adresse ou référence de composant mémorisée dans la variable MyPartner.
```

```
MyPort.getreply -> sender MySenderVar;
// Supprime la réponse sommitale de CL1, mais mémorise le répondeur en
// l'affectant à la variable MySenderVar
```

22.2.5.2 Obtention de réponse à tout accès

Le mot clé **any** permet d'obtenir une réponse à tout accès. Par exemple:

```
any port.getreply(Myproc)
```

22.2.6 L'opération d'acquisition

L'opération **catch** sert à acquérir des exceptions déclenchées par une entité homologue en réaction à un appel de procédure. Cette opération ne doit être utilisée qu'aux accès en mode procédure (ou mixtes). Le type de l'exception acquise doit être spécifié dans la signature de la procédure qui a déclenché l'exception.

```
MySyncPort.catch(MySignature, integer: MyVar);
// Spécifie l'acquisition d'une exception déclenchée par une procédure avec
// une signature Mysignature à l'accès MySyncPort. L'exception est un entier
// qui possède la même valeur que la variable
// MyVar. L'entier (facultatif) identifiant le type n'est pas strictement
// nécessaire parce que le type est déjà donné par la définition de MyVar.
// Cependant, dans des tests élémentaires longs et complexes un tel
// identificateur de type peut être utilisé afin d'améliorer la lisibilité.

MySyncPort.catch(MySignature, MyVar);
// Est une variante de l'exemple précédent.

MySyncPort.catch(MySignature, A<B);
// Acquiert une exception booléenne de valeur "true" ou "false" selon le
// résultat de A<B déclenchée par une procédure avec une signature
// MySignature à un accès MySyncPort.
```

L'opération **catch** peut faire partie de la partie acceptation d'un appel ou être utilisée pour déterminer une variante dans une instruction **alt**. Si l'opération **catch** est utilisée dans la partie acceptation d'une opération **call**, les informations relatives au nom de l'accès et à la référence de signature indiquant la procédure qui a déclenché l'exception sont redondantes car elles font suite à l'opération **call**. Cependant, pour des raisons de lisibilité (par exemple dans le cas d'instructions **call** complexes) ces informations doivent être répétées.

Les exceptions sont spécifiées sous la forme de types et peuvent donc être traitées comme des messages. Par exemple, des modèles peuvent être utilisés afin d'établir une distinction entre différentes valeurs du même type d'exception.

```
MySyncPort.catch(MySignature, MyTemplate:{5, MyVar});
// Acquiert une exception déclenchée par une procédure avec une signature
// Mysignature à un accès MySyncPort qui répond aux conditions définies par
// le modèle MyTemplate avec les paramètres réels 5 et MyVar.
```

L'opération **catch** nécessite le nom de l'accès, les critères d'appariement pour le type et la valeur, une restriction facultative **from** (c'est-à-dire la sélection du correspondant de communication) et une affectation facultative à des variables de l'exception d'appariement et du composant **sender**. Par exemple:

```
MySyncPort.catch(MySignature, charstring:"Hello")from MyPartner;
// Acquiert la chaîne IA5 "Hello" déclenchée par une procédure avec une
// signature Mysignature à un accès
// MySyncPort issue d'une entité avec une adresse ou référence de composant
// mémorisée dans la variable MyPartner.
```

```
MySyncPort.catch(MySignature, MyType:*) from MyPartner -> value MyVar;
// Acquiert une exception avec une valeur arbitraire de MyType (déclenchée
// par une procédure avec une signature Mysignature à un accès MySyncPort
// issue d'un composant avec une référence mémorisée dans
// la variable MyPartner) qui ensuite est affectée à la variable MyVar. // MyVar
doit être du type MyType.

MySyncPort.catch(MySignature, MyTemplate (5, MyVarOne)) -> value MyVarTwo sender
MyPartner;
// Acquiert une exception déclenchée par une procédure avec une signature
// Mysignature avec une valeur qui répond aux conditions définies par le
// modèle MyTemplate avec les paramètres réels 5 et
// MyVarOne. Ensuite l'exception est affectée à MyVarTwo. L'adresse ou
// référence de l'entité expéditrice est extraite par l'opération
// d'acquisition et affectée à MyPartner.
```

22.2.6.1 L'exception de temporisation

Il existe une exception **timeout** particulière qui est acquise par l'opération **catch**. L'exception **timeout** offre une sortie d'urgence si une procédure appelée ne répond pas ou ne déclenche pas d'exception dans un délai prédéterminé. Par exemple:

```
MyPort.catch(timeout); // Acquiert une exception de temporisation.
```

L'acquisition d'exceptions de temporisation doit être limitée à la partie traitement d'exception d'un appel. Aucun autre critère d'appariement (y compris une partie **from**) et aucune partie d'affectation ne sont autorisés pour une opération **catch** qui traite une exception de temporisation.

22.2.6.2 Acquisition de toute exception

Une opération d'acquisition sans liste d'arguments permet d'acquérir toute exception valide. Le cas le plus général est la non-utilisation du mot clé **from** et l'absence de partie affectation. Cette instruction permet également d'acquérir l'exception **timeout**. Par exemple:

```
MyPort.catch;

MyPort.catch from MyPartner;

MyPort.catch -> sender MySenderVar;
```

22.2.6.3 Acquisition à tout accès

Le mot clé **any** permet d'acquérir une exception à tout accès. Par exemple:

```
any port.catch(timeout)
```

22.2.7 L'opération de vérification

L'opération **check** est une opération générique qui permet un accès en lecture à l'élément sommital de files d'attente à un accès *entrant* en mode message et en mode procédure, sans retirer cet élément sommital de la file d'attente. L'opération de vérification doit traiter des valeurs d'un certain type aux accès en mode message. Elle doit également établir une distinction entre appels à accepter, exceptions à acquérir et réponses à des appels précédents aux accès en mode procédure.

Les opérations de réception **receive, getcall, getreply** et **catch** sont utilisées par l'opération **check** avec leurs parties d'appariement et d'affectation afin de définir la condition qui doit être vérifiée et d'extraire la valeur ou les valeurs de ses paramètres, si nécessaire.

```
MyAsyncPort.check(receive(integer: 5));
// Vérifiera l'existence d'un entier de 5 en tant que message sommital à
// l'accès asynchrone MyAsyncPort.
```

```
MyPort.check(getcall(MyProc:{5, MyVar}) from MyPartner);
// Vérifiera l'existence d'un appel de MyProc à MyCL (avec les paramètres
// "in" ou "inout" 5 et MyVar) issu d'une entité homologue avec l'adresse ou
// la référence de composant mémorisée dans la variable MyPartner.

MyPort.check(getreply(MyProc:{5, MyVar} value 20));
// Vérifiera l'existence d'une réponse issue de la procédure MyProc à MyPort
// où la valeur renvoyée est 20 et où les valeurs des deux paramètres "out"
// ou "inout" sont 5 et la valeur de MyVar.

MySyncPort.check(catch(MySignature, MyTemplate (5, MyVar)));
// Vérifiera l'existence d'une exception déclenchée par une procédure avec
// une signature Mysignature à un accès MySyncPort
// qui répond aux conditions définies par le modèle MyTemplate avec les
// paramètres réels 5 et MyVar.
```

C'est l'élément *sommital* d'une file d'attente à un accès entrant qui doit être vérifié (il n'est pas possible d'examiner *l'intérieur* d'une file). Si la file est vide, l'opération **check** échoue. Si la queue n'est pas vide, une copie de l'élément sommital est effectuée et l'opération de réception spécifiée dans l'opération **check** échoue. Si la file n'est pas vide, une copie de l'élément sommital est effectuée et l'opération de réception spécifiée dans l'opération **check** est effectuée sur la copie. L'opération de vérification échoue si la fonction de réception échoue, c'est-à-dire si les critères d'appariement ne sont pas satisfaits. Dans ce cas, la *copie* de l'élément sommital de la file est rejetée et l'exécution du test se poursuit normalement, c'est-à-dire que l'on évalue la prochaine option venant en variante de l'opération de vérification. Celle-ci réussit si la fonction de réception réussit.

L'utilisation incorrecte de l'opération **check**, par exemple la vérification d'une exception à un accès en mode message, doit provoquer une erreur de test élémentaire.

NOTE – Dans la plupart des cas, l'utilisation correcte de l'opération de vérification peut être contrôlée statiquement, c'est-à-dire avant compilation.

Exemple:

```
MyPort.check(getreply(MyProc1:{*, MyVar} value *) -> value MyReturnValue
param(MyPar1));
// Dans cet exemple la valeur renvoyée est affectée à la variable
// MyReturnValue et la valeur du premier paramètre "out" ou "inout" est
// affectée à la variable MyPar1.

MyPort.check(getcall(MyProc:{5, MyVar}) from MyPartner -> param (MyPar1Var,
MyPar2Var));
// Dans cet exemple les deux paramètres de MyProc sont considérés comme des
// paramètres "inout" et leurs valeurs sont affectées à MyPar1Var et
// MyPar2Var.

MyPort.check(getcall(MyProc:{5, MyVar}) -> sender MySenderVar);
// Acceptera un appel de MyProc à MyCL avec les paramètres "in" ou "inout" 5
// et MyVar. L'appelant est extrait et mémorisé dans la variable MySenderVar.
```

22.2.7.1 L'opération de vérification totale

Une opération **check** sans liste d'arguments permet de vérifier s'il y a attente de traitement dans une file d'accès entrant. L'opération *CheckAny* permet d'établir une distinction entre différents expéditeurs (dans le cas de connexions point à multipoint) au moyen du mot clé **from**. Elle permet également d'extraire l'expéditeur au moyen d'une partie affectation abrégée contenant un champ d'expéditeur.

Exemple:

```
MyPort.check;

MyPort.check(from MyPartner);

MyPort.check(-> sender MySenderVar);
```

22.3 Opérations de commande des accès de communication

Les opérations de commande des accès de communication en mode message, en mode procédure et mixtes sont les suivantes en notation TTCN-3:

- **clear**: suppression du contenu d'une file d'attente à un accès entrant;
- **start**: mise en état de réception et raccordement d'un accès;
- **stop**: terminaison de l'état de réception et interdiction d'opérations d'envoi à un accès.

22.3.1 L'opération de libération d'accès

L'opération **clear** supprime le contenu de la file *entrante* de l'accès nommé. Si la file d'accès est déjà vide, cette opération n'a pas de suite.

```
MyPort.clear;// Libère l'accès MyPort
```

22.3.2 L'opération d'ouverture d'accès

Si un accès est défini comme permettant des opérations de réception telles que **receive**, **getcall**, etc., l'opération **start** vide la file entrante de l'accès nommé et met l'accès en état de réception du trafic. Si l'accès est défini comme permettant des opérations d'émission, les opérations telles que **send, call, raise**, etc. peuvent également être exécutées à cet accès. Par exemple:

```
MyPort.start;// Ouvre MyPort
```

Par défaut, tous les accès d'un composant doivent être ouverts lorsqu'un composant commence son exécution.

22.3.3 L'opération de fermeture d'accès

Si un accès est défini comme permettant des opérations de réception telles que **receive, getcall**, etc., l'opération **stop** met fin à l'état de réception par l'accès nommé. Si l'accès est défini comme permettant des opérations d'émission, l'opération de fermeture d'accès interdit l'exécution d'opérations telles que **send**, **call**, **raise**, etc. Par exemple:

```
MyPort.stop; // Ferme MyPort
```

22.4 Utilisation des mots clés *any* et *all* avec les accès

Les mots clés **any** et **all** peuvent être utilisés avec les opérations de configuration indiquées dans le Tableau 16.

Tableau 16/Z.140 – Utilisation des mots clés Any et All avec des accès

Opération	Permise		Exemple
	any	**all**	
Opérations de communication de type réception (**receive, trigger, getcall, getreply, catch, check**)	Oui		**any port.receive**
connect / map			
start		Oui	**all port.start**
stop		Oui	**all port.stop**
clear		Oui	**all port.clear**

23 Opérations de temporisation

La notation TTCN-3 prend en charge un certain nombre d'opérations de temporisation (voir Tableau 17) qui peuvent être utilisées dans des tests élémentaires, dans des fonctions et dans la partie commande d'un module.

Tableau 17/Z.140 – Aperçu général des opérations de temporisation TTCN-3

Opérations de temporisation	
Instruction	**Mot clé ou symbole associé**
Armement de temporisateur	**start**
Arrêt de temporisateur	**stop**
Lecture de la durée écoulée	**read**
Vérification d'armement de temporisateur	**running**
Événement de temporisation	**timeout**

23.1 L'opération d'armement de temporisateur

L'opération d'armement de temporisateur **start** sert à indiquer qu'un temporisateur doit être armé. Les valeurs de temporisation doivent être de type **float**. Par exemple:

```
MyTimer1.start;     // MyTimer1 est armé avec la durée par défaut.
MyTimer2.start(20E-3);  // MyTimer2 est armé avec une durée de 20 ms.
```

Le paramètre facultatif de temporisation doit être utilisé si aucune valeur de durée par défaut n'est indiquée ou si l'on souhaite neutraliser la valeur par défaut qui est spécifiée dans la déclaration de temporisation. Lorsqu'une durée de temporisation est neutralisée, la nouvelle valeur ne s'applique qu'à l'instance actuelle du temporisateur. D'éventuelles opérations **start** ultérieures, ne spécifiant pas de durée, doivent utiliser la durée par défaut pour ce temporisateur. L'horloge de temporisation commence par la valeur en virgule flottante zéro (0.0) jusqu'à une valeur maximale indiquée par le paramètre de durée.

23.2 L'opération de désarmement de temporisateur

L'opération **stop** sert à désarmer un temporisateur actif et à le retirer de la liste des temporisateurs actifs. Un temporisateur désarmé devient inactif et son temps écoulé est mis à la valeur flottante zéro (0.0). Si le nom du temporisateur dans l'opération **stop** est **all**, tous les temporisateurs en cours (c'est-à-dire actifs) sont désarmés. Par exemple:

```
MyTimer1.stop;      // Désarme MyTimer1
all timer.stop;     // Désarme tous les temporisateurs actifs
```

Le désarmement d'un temporisateur inactif est une opération valide, bien qu'elle n'ait pas d'effet.

23.3 L'opération de lecture du temporisateur

L'opération **read** sert à extraire la durée qui s'est écoulée depuis l'armement du temporisateur spécifié et à mémoriser cette durée dans la variable spécifiée. Cette variable doit être de type **float**. Par exemple:

```
var float Myvar;
MyVar := MyTimer1.read; // Affecte à MyVar la durée qui s'est écoulée
                        // depuis que MyTimer1 a été armé
```

L'application de l'opération **read** à un temporisateur inactif renvoie la valeur zéro.

23.4 L'opération d'exécution de temporisation

L'opération **running** sert à vérifier si un temporisateur est armé ou non (c'est-à-dire s'il a été armé et n'a pas encore expiré ou n'a pas encore été annulé). L'opération renvoie la valeur **true** si le temporisateur est armé et la valeur **false** dans l'autre cas. Par exemple:

```
if (MyTimer1.running) { ... }
```

23.5 L'événement de fin de temporisation

L'opération **timeout** indique l'expiration d'un temporisateur préalablement armé. Cette opération peut être utilisée dans les variantes en même temps que les opérations **receive, getcall, getreply, catch** et **other timeout**.

Exemple:

```
MyTimer1.timeout;  // Vérifie l'expiration du temporisateur déjà armé
                   // MyTimer1
```

Le mot clé **any** est utilisé pour indiquer l'expiration de tout temporisateur (plutôt qu'un temporisateur explicitement nommé) armé dans les limites de la temporisation. Par exemple:

```
any timer.timeout; // Vérifie l'expiration de tout temporisateur déjà armé
```

23.6 Utilisation des mots clés *any* et *all* avec les temporisateurs

Les mots clés **any** et **all** peuvent être utilisés avec les opérations de temporisation comme indiqué dans le Tableau 18.

Tableau 18/Z.140 – Mots clés Any et All avec les temporisateurs

Opération	Permise		Exemple
	any	**all**	
start			
stop		Oui	**All timer.stop**
read			
running	Oui		**if (any timer.running) {...}**
timeout	Oui		**Any timer.timeout**

24 Opération de verdict de test

Les opérations de verdict (voir Tableau 19) permettent de poser et d'extraire des verdicts au moyen, respectivement, des opérations **get** et **set**. Ces opérations ne doivent être utilisées que dans les tests élémentaires et les fonctions.

Tableau 19/Z.140 – Aperçu général des opérations de verdict de test TTCN-3

Opérations de verdict de test	
Instruction	**Mot clé ou symbole associé**
Définition du verdict local	**Verdict.set**
Obtention du verdict local	**Verdict.get**

Chaque composant de test de la configuration active doit conserver son propre verdict local. Celui-ci est un objet créé pour chaque composant de test au moment de son instanciation. Il sert à suivre le verdict individuel dans chaque composant de test (c'est-à-dire dans le composant MTC et dans chacun des composants PTC).

NOTE – Contrairement à la notation TTCN-2, l'affectation d'un verdict final n'arrête pas l'exécution du composant de test dans lequel le comportement est en train de s'exécuter. Au besoin, cet arrêt doit être explicitement indiqué au moyen de l'instruction **stop**.

24.1 Verdict de test élémentaire

Il y a par ailleurs un verdict global qui est mis à jour lorsque chaque composant de test (c'est-à-dire le MTC et chaque PTC) termine l'exécution. Ce verdict n'est pas accessible aux opérations **get** et **set**. La valeur de ce verdict doit être renvoyée par le test élémentaire lorsqu'il termine l'exécution. Si le verdict renvoyé n'est pas explicitement sauvegardé dans la partie commande (par exemple s'il n'est pas affecté à une variable) il est perdu.

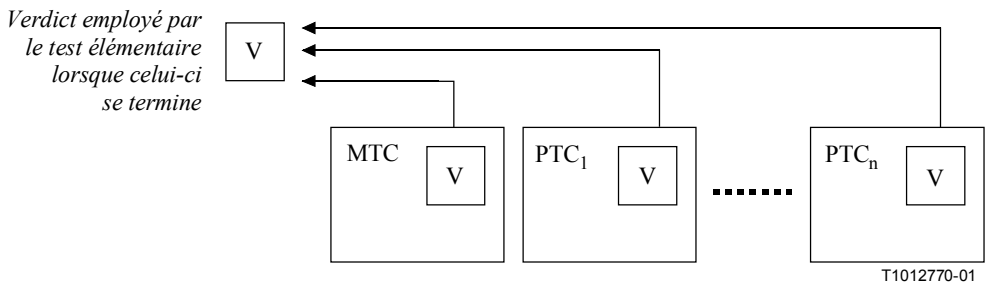


Figure 10/Z.140 – Illustration de la relation entre verdicts

NOTE – La notation TTCN-3 ne spécifie pas les mécanismes réels qui effectuent la mise à jour des verdicts locaux et de test élémentaire. Ces mécanismes dépendent de l'implémentation.

24.2 Valeurs de verdict et règles d'effacement par recouvrement

Le verdict peut avoir cinq valeurs différentes: **pass, fail, inconc, none** et **error**, qui sont les valeurs distinctives du type de verdict (voir § 6.1).

NOTE – La valeur **inconc** indique un verdict non concluant.

L'opération **set** ne doit être utilisée qu'avec les valeurs **pass**, **fail**, **inconc** et **none**. Par exemple:

```
verdict.set(pass);
verdict.set(inconc);
```

La valeur du verdict local peut être extraite au moyen de l'opération **get**. Par exemple:

```
MyResult := verdict.get; // Où MyResult est une variable de type verdicttype
```

Lorsqu'un composant de test est instancié, son objet de verdict local est créé et mis à la valeur **none**.

Lors d'une modification de la valeur du verdict (par exemple au moyen de l'opération **set**), l'effet de cette modification doit suivre les règles d'effacement par recouvrement énumérées dans le Tableau 20. Le verdict de test élémentaire est implicitement posé à la fin d'un composant de test. L'effet de cette opération implicite doit également suivre les règles de recouvrement énumérées dans le Tableau 20.

Tableau 20/Z.140 – Règles de recouvrement pour le verdict

Valeur actuelle du verdict	Nouvelle valeur d'affectation de verdict			
	pass	**inconc**	**fail**	**none**
none	**pass**	**inconc**	**fail**	**none**
pass	**pass**	**inconc**	**fail**	**pass**
inconc	**inconc**	**inconc**	**fail**	**inconc**
fail	**fail**	**Fail**	**fail**	**fail**

Exemple:

```
:
verdict.set(pass); // Le verdict local est posé à la valeur "pass"
:
verdict.set(fail); // Jusqu'à ce que cette ligne soit exécutée ce qui
                   // donnera la valeur du verdict local recouverte par la
                   // valeur "fail". Lorsque le composant PTC se termine, le
                   // verdict de test élémentaire est posé à la valeur
                   // "fail"
```

24.2.1 Verdict d'erreur

Le verdict **error** est spécial parce qu'il est posé par le système de test afin d'indiquer qu'une erreur de test élémentaire (c'est-à-dire d'exécution) s'est produite. Ce verdict ne doit pas être posé par l'opération **set**. Aucune autre valeur de verdict ne peut neutraliser un verdict d'erreur. En d'autres termes, un verdict d'erreur ne peut résulter que d'une opération d'exécution de test élémentaire.

25 Opérations relatives au système sous test (SUT)

Dans certaines situations de test, il peut n'y avoir aucune interface explicite avec le système SUT et il peut être nécessaire de faire en sorte que ce système déclenche certaines actions (par exemple envoyer un message au système de test).

Cette action peut être définie par une chaîne. Par exemple:

```
sut.action("Send MyTemplate on lower PCO");     // Description informelle de
                                                // l'action du système SUT
```

ou sous la forme d'une référence à un modèle qui spécifie la structure du message qui doit être envoyé par le système SUT, par exemple:

```
sut.action(MyTemplate); // Cela est équivalent à l'instruction TTCN-2
                        // IMPLICIT SEND.
```

Dans les deux cas, il n'y a pas de spécification concernant le message à destination ou en provenance du système SUT qui déclenche cette action. Il n'y a qu'une spécification informelle de la réaction réellement requise.

Les actions de système SUT peuvent être spécifiées dans des tests élémentaires, dans des fonctions, dans des variantes nommées et dans la partie commande d'un module.

26 Partie commande d'un module

Les tests élémentaires sont définis dans la partie définitions et exécutés dans la partie commande du module. Toutes les variables, temporisations, etc. (éventuelles), définies dans la partie commande d'un module, doivent être transmises au test élémentaire par paramétrisation si elles doivent être utilisées dans la définition comportementale de ce test élémentaire. En d'autres termes, la notation TTCN-3 ne prend pas en charge les variables globales de sorte quelconque.

La configuration de test doit être réinitialisée au début de chaque test élémentaire. En d'autres termes, toutes les opérations **create, connect**, etc. pouvant avoir été effectuées dans un test élémentaire précédent ne sont pas 'visibles' par le nouveau test élémentaire.

26.1 Exécution des tests élémentaires

Un test élémentaire est appelé au moyen d'une instruction **execute**. Le résultat de l'exécution d'un test élémentaire est renvoyé sous la forme d'un verdict de test ayant la valeur **none, pass, inconclusive, fail** ou **error**. Cette valeur peut être affectée à une variable pour traitement complémentaire.

En variante, l'instruction **execute** permet de superviser un test élémentaire au moyen d'une durée de temporisation. Si le test élémentaire ne se termine pas dans cette durée, le résultat de l'exécution du test élémentaire doit donner un verdict d'erreur et le système de test doit mettre fin à l'essai élémentaire.

Exemple:

```
execute(MyTestCase1());                 // Exécute MyTestCase1, sans mémoriser
                                        // le verdict de test renvoyé et sans
                                        // supervision par durée

MyVerdict := execute(MyTestCase2());    // Exécute MyTestCase2 et mémorise le
                                        // verdict résultant dans la variable
                                        // MyVerdict

MyVerdict := execute(MyTestCase3(),5E-3);  // Exécute MyTestCase3 et mémorise
                                           // le verdict résultant dans la
                                           // variable MyVerdict. Si le test
                                           // ne s'arrête pas dans les 5 ms,
                                           // MyVerdict prend la valeur
                                           // 'error'
```

26.2 Terminaison des tests élémentaires

Un test élémentaire se termine en même temps que le composant MTC. Après la fin de celui-ci, tous les composants de test parallèles doivent se terminer au moyen de tests (c'est-à-dire au moyen du système de test).

NOTE 1 – Le mécanisme concret d'arrêt de tous les composants PTC dépend de l'utilitaire employé. Il est donc hors du domaine d'application de la présente Recommandation.

Le verdict final d'un test élémentaire est calculé sur la base des verdicts locaux finals des différents composants de test, conformément aux règles définies dans le § 24. Le verdict local réel d'un composant de test devient son verdict lorsque ce composant se termine ou est arrêté au moyen de tests (c'est-à-dire au moyen du système de test).

NOTE 2 – Afin d'éviter des conditions de concurrence lors du calcul des verdicts de test à cause de l'arrêt différé de composants PTC, il y a lieu que le composant MTC veille à ce que tous les composants PTC se soient arrêtés (au moyen de l'instruction **done**) avant que lui-même s'arrête.

26.3 Contrôle de l'exécution des tests élémentaires

Des instructions de programmation, limitées à celles qui sont définies dans le Tableau 11, peuvent être utilisées dans la partie commande d'un module afin de spécifier des comportements comme l'ordre dans lequel les tests doivent être exécutés ou le nombre de fois qu'un test élémentaire peut être exécuté. Par exemple:

```
module MyTestSuite
{    :
     control
     { :
       // Faire ce test 10 fois
       count:=0;
       while (count < 10)
       {  execute (MySimpleTestCase1());
          count := count+1;
       }
     }
}
```

Si aucune instruction de programmation n'est utilisée, les tests élémentaires sont, par défaut, exécutés dans l'ordre séquentiel de leurs apparitions dans la partie commande du module.

NOTE – Cela n'exclut pas la possibilité que certains utilitaires cherchent à neutraliser ce séquencement par défaut afin de permettre à un utilisateur ou à un utilitaire de sélectionner un ordre d'exécution différent.

Les tests élémentaires renvoient une valeur unique de type **verdicttype**, de sorte qu'il est possible de commander l'ordre d'exécution en fonction du résultat d'un test élémentaire. Par exemple:

```
if (MySimpleTestCase() == pass) { log("Success!") }
```

26.4 Sélection de test élémentaire

Des expressions booléennes peuvent être utilisées pour sélectionner et désélectionner les tests élémentaires à exécuter. Cela comprend évidemment l'utilisation de fonctions qui renvoient une valeur booléenne.

NOTE – Cela est équivalent aux expressions de sélection de tests nommés dans la notation TTCN-2.

Exemple:

```
module MyTestSuite
{    :
     control
     { :
       if (MySelectionExpression1())
       {  execute(MySimpleTestCase1());
          execute(MySimpleTestCase2());
          execute(MySimpleTestCase3());
       }
       if (MySelectionExpression2())
       {  execute(MySimpleTestCase4());
          execute(MySimpleTestCase5());
          execute(MySimpleTestCase6());
       }
     :
     }
}
```

Une autre façon d'exécuter collectivement des tests élémentaires consiste à les regrouper dans une fonction et à exécuter cette fonction à partir de la commande de module. Par exemple:

```
:
function MyTestCaseGroup1()
{    execute(MySimpleTestCase1());
     execute(MySimpleTestCase2());
     execute(MySimpleTestCase3());
}
function MyTestCaseGroup2()
{    execute(MySimpleTestCase4());
     execute(MySimpleTestCase5());
     execute(MySimpleTestCase6());
}
:
control
{    if (MySelectionExpression1()) { MyTestCaseGroup1(); }
     if (MySelectionExpression1()) { MyTestCaseGroup2(); }
     :
}
:
```

26.5 Utilisation de temporisateurs dans les commandes

Les temporisateurs peuvent être utilisés pour commander l'exécution de tests élémentaires. A cette fin, l'on peut utiliser une temporisation explicite dans l'instruction d'exécution. Par exemple:

```
MyReturnVal := execute (MyTestCase(), 7E-3); // Variable de type verdicttype
// Où le verdict renvoyé aura la valeur "error" si le test élémentaire
// n'assure pas l'exécution dans les 7 ms
```

Les opérations de temporisation peuvent également être utilisées. Par exemple:

```
// Exemple de l'utilisation de l'opération de temporisation active
while (T1.running or x<10)  // Où T1 est un temporisateur déjà armé
{    execute(MyTestCase());
     x := x+1;
}

// Exemple de l'utilisation des opérations "start" et "timeout"

timer T1 := 1;
:
execute(MyTestCase1());
T1.start;
T1.timeout;  // Pause avant l'exécution du prochain test élémentaire
execute(MyTestCase2());
```

27 Spécification des attributs

Des attributs peuvent être associés aux éléments linguistiques de la notation TTCN-3 au moyen de l'instruction **with**. La syntaxe de l'argument de l'instruction **with** (c'est-à-dire les attributs réels) est définie simplement sous la forme d'une chaîne alphanumérique libre.

Il y a trois sortes d'attributs:

a) **display**: permet la spécification des attributs d'affichage relatifs aux formats spécifiques de présentation;

b) **encode**: permet des références à des règles de codage spécifiques;

c) **extension**: permet la spécification des attributs définis par l'utilisateur.

27.1 Attributs d'affichage

Tous les éléments linguistiques de la notation TTCN-3 peuvent avoir des attributs d'affichage **`display`** permettant de spécifier la façon dont des éléments linguistiques particuliers doivent être affichés, par exemple en format graphique.

Des chaînes d'attributs particulières, associées aux attributs d'affichage pour le format de présentation (conformité) tabulaire, peuvent être trouvées dans la Rec. UIT-T Z.141 [1].

Des chaînes d'attributs particulières, associées aux attributs d'affichage pour le format de présentation graphique, peuvent être trouvées dans le projet de Rec. UIT-T Z.142 [2].

D'autres attributs d'affichage peuvent être définis par l'utilisateur.

NOTE – Etant donné que les attributs définis par l'utilisateur ne sont pas normalisés, leur interprétation peut différer ou même ne pas être prise en charge selon les utilitaires vendus par différents vendeurs.

27.2 Codage des attributs

Les règles de codage définissent la façon dont une valeur particulière, un modèle, etc., est codé et transmis, habituellement sous la forme d'un flux binaire, par un accès de communication. La notation TTCN-3 ne possède pas de mécanisme de codage par défaut. En d'autres termes, les règles ou directives de codage sont définies de façon quelque peu externe à la notation TTCN-3.

L'attribut **`encode`** permet l'association, à une définition de type TTCN-3 (et seulement à une définition de type), d'une règle ou directive de codage à appliquer.

Des chaînes d'attribut spéciales, associées aux attributs de codage en notation ASN.1, sont décrites dans l'Annexe E.

La façon dont les règles de codage actuelles sont définies (par exemple en langage courant, sous forme de fonctions, etc.) est hors du domaine d'application de la présente Recommandation. Si aucune règle spécifique n'est citée en référence, le codage doit relever de chaque implémentation.

Les attributs de codage seront le plus souvent utilisés de façon hiérarchique. Le niveau sommital est le module entier; le niveau suivant est un groupe de types et le plus bas niveau est un type individuel:

a) **`module`**: le codage s'applique à tous les types définis dans le module, y compris les types de base TTCN-3;

b) **`group`**: le codage s'applique à un groupe de définitions de type définies par l'utilisateur;

c) **`type`**: le codage s'applique à un unique type défini par l'utilisateur;

d) **`field`**: le codage s'applique à un champ dans un **`type "record" ou "set"`**;

Exemple:

```
module MyTTCNmodule
{    :
     import type MyRecord from MySecondModule with {encode "MyRule 1"}
          // Toutes les instances de MyRecord seront codées selon MyRule 1
     :
     type charstring MyType; // Codé normalement selon la règle globale
     :
     group MyRecords
     {    :
       type record MyPDU1
       {
       integer   field1,     // field1 sera codé selon la Règle 3
       boolean   field2,     // field2 sera codé selon la Règle 3
       Mytype    field3      // field3 sera codé selon la Règle 2
       }
       with {encode (field1, field2) "Rule 3"}
```

```
      :
    }
    with {encode "Rule 2"}

}
with {encode "Global encoding rule"}
```

27.2.1 Codages invalides

Si l'on souhaite spécifier des règles de codage invalides, ces règles doivent être spécifiées dans une source pouvant faire l'objet de références et externe au module, de la même façon que les règles de codage valides sont citées en référence.

27.3 Attributs d'extension

Tous les éléments linguistiques de la notation TTCN-3 peuvent avoir des attributs d'extension spécifiés par l'utilisateur.

NOTE – Etant donné que les attributs définis par l'utilisateur ne sont pas normalisés, leur interprétation peut différer ou même ne pas être prise en charge selon les utilitaires fournis par différents vendeurs.

27.4 Portée des attributs

Une instruction **with** associe toujours des attributs à des éléments linguistiques isolés. Il est également possible d'associer des attributs à un certain nombre d'éléments linguistiques en associant une instruction **with** à l'unité de portée voisine ou à un groupe d'éléments linguistiques voisins.

L'instruction **with** suit les règles de portée définies au § 5.4, c'est-à-dire qu'une instruction **with** qui est placée à l'intérieur de la portée d'une autre instruction **with** doit neutraliser l'instruction **with** la plus extérieure. Cette règle doit également s'appliquer à l'utilisation de l'instruction **with** avec des groupes. Il convient de prendre des précautions lors de l'utilisation du procédé d'effacement par recouvrement en combinaison avec des références à des définitions isolées. En règle générale, les attributs doivent être affectés et recouverts en fonction de leur ordre d'apparition.

Exemple:

```
// MyPDU1 sera représentée sous la forme d'une unité PDU
type record MyPDU1 { ... } with { display "PDU"}

// MyPDU2 sera représentée sous la forme d'une unité PDU avec l'attribut
// d'extension propre à l'application MyRule
type record MyPDU2 { ... }
with
{
     display "PDU";
     extension "MyRule"
}

// La définition de groupe suivante ...
group MyPDUs {
     type record MyPDU3 { ... }
     type record MyPDU4 { ... }
}
with {display "PDU"}    // Tous les types de groupe d'unités PDU seront
                        // représentés sous la forme d'une unité PDU.

// est identique à
group MyPDUs {
     type record MyPDU3 { ... } with { display "PDU"}
     type record MyPDU4 { ... } with { display "PDU"}
}
```

```
// Exemple de l'utilisation du procédé de recouvrement de l'instruction "with"
group MyPDUs
{
     type record MyPDU1 { ... }
     type record MyPDU2 { ... }

     group MySpecialPDUs
     {
       type record MyPDU3 { ... }
       type record MyPDU4 { ... }
     }
     with {extension "MySpecialRule"}      // MyPDU3 et MyPDU4 auront l'attribut
                                           // d'extension spécifique
                                           // MySpecialRule.
}
with
{
     display "PDU";          // Tous les types d'unités MPDUs de groupe seront
                             // représentés sous la forme d'une unité PDU et
     extension "MyRule";     // (s'ils ne sont pas recouverts) ont l'attribution
                             // d'extension MyRule
}

// est identique à ...
group MyPDUs
{
     type record MyPDU1 { ... } with {display "PDU"; extension "MyRule" }
     type record MyPDU2 { ... } with {display "PDU"; extension "MyRule" }
     group MySpecialPDUs {
       type record MyPDU3 { ... } with {display "PDU"; extension "MySpecialRule"
}
       type record MyPDU4 { ... } with {display "PDU"; extension "MySpecialRule"
}
     }
}
```

27.5 Règles d'effacement par recouvrement pour les attributs

Une définition d'attribut contenue dans une unité de portée inférieure neutralisera une définition générale d'attribut contenue dans une unité de portée supérieure. Par exemple:

```
type record MyRecordA
{
     :
} with {encode "RuleA"}

// Ci-dessous, MyRecordA est codé selon la Règle A et non selon la Règle B.
type record MyRecordB
{    :
     fieldMyRecordA
} with {encode "RuleB"}
```

Une définition d'attribut contenue dans une unité de portée inférieure peut être recouverte dans une unité de portée supérieure au moyen de la directive **override**. Par exemple:

```
type record MyRecordA
{
     :
} with {encode "RuleA"}
```

```
// Ci-dessous, MyRecordA est codé selon la Règle B
type record MyRecordB
{    :
     fieldA   MyRecordA
} with {encode override "RuleB"}
```

La directive de recouvrement force à l'attribut spécifié tous les types contenus dans des unités de portée inférieures.

27.6 Modification des attributs d'éléments linguistiques importés

En général, un élément linguistique est importé en même temps que ses attributs. Dans certains cas, ces attributs peuvent nécessiter une modification lors de l'importation de l'élément linguistique. Un type peut par exemple être affiché comme primitive ASP dans un certain module puis être importé par un autre module où il sera affiché sous la forme d'une unité PDU. Dans de tels cas, il est permis de modifier les attributs dans l'instruction d'importation.

Exemple:

```
Import type MyType from MyModule with {display "ASP"} // MyType sera représenté
                                                      // sous la forme d'une
                                                      // primitive ASP.

Import group MyGroup from MyModule with
{
     display "ASP";                    // Par défaut tous les types seront
                                       // représentés sous la forme d'une
                                       // primitive ASP.
     extension "MyRule"
}
```

Annexe A

Formalisme BNF et sémantique statique

A.1 Formalisme BNF de la notation TTCN-3

Cette annexe définit la syntaxe de la notation TTCN-3 au moyen du formalisme BNF étendu (abrégé ci-dessous en BNF).

A.1.1 Conventions pour la description syntaxique

Le Tableau A.1 définit la métanotation utilisée pour spécifier la grammaire du formalisme BNF étendu en notation TTCN-3.

Tableau A.1/Z.140 – La métanotation syntaxique

::=	sont définis comme étant
abc xyz	abc suivi par xyz
|	alternative (entre 2 variantes)
[abc]	0 ou 1 instances de "abc"
{abc}	0, 1 ou plusieurs instances de "abc"
{abc}+	1 ou plusieurs instances de "abc"
(...)	groupement textuel
Abc	symbole non terminal abc
abc	symbole terminal abc
"abc"	symbole terminal abc

A.1.2 Symboles du terminateur d'instruction

En général, toutes les structures linguistiques de la notation TTCN-3 (c'est-à-dire définitions, déclarations, instructions et opérations) sont terminées par un point-virgule (**;**). Le point-virgule est facultatif si la structure linguistique se termine par une accolade de fermeture (}) ou si le symbole suivant est une accolade de fermeture (}), c'est-à-dire que la structure linguistique est la dernière instruction d'un bloc d'instructions.

A.1.3 Identificateurs

En notation TTCN-3, les identificateurs sont sensibles à la hauteur de casse et ne peuvent contenir que des lettres minuscules (**a-z**), des lettres majuscules (**A-Z**) et des chiffres numériques (**0-9**). L'utilisation du symbole de soulignement (_) est également permise. Un identificateur doit commencer par une lettre (c'est-à-dire par un caractère autre qu'un nombre ou un soulignement).

A.1.4 Commentaires

Des commentaires rédigés en texte courant peuvent apparaître à tout endroit d'une spécification en notation TTCN-3.

Les commentaires de bloc doivent être ouverts par la paire de symboles /* et être fermés par la paire de symboles */. Par exemple:

```
/* Ceci est un commentaire de bloc
réparti sur deux lignes */
```

Un commentaire de bloc ne doit pas être imbriqué.

```
/* Ceci n'est pas /* un commentaire */ autorisé */
```

Un commentaire de ligne doit être ouvert par la paire de symboles // et être fermé par un symbole <*newline*>. Par exemple:

```
// Ceci est un commentaire de ligne
// réparti sur deux lignes
```

Des commentaires de ligne peuvent suivre des instructions de programmation TTCN-3 mais ne doivent pas être imbriqués dans une instruction. Par exemple:

```
// Ce qui suit n'est pas légal:
const // Ceci est MyConst integer MyConst := 1;

// Ce qui suit est légal:
const integer MyConst := 1; // Ceci est MyConst
```

A.1.5 Symboles terminaux de la notation TTCN-3

Les symboles terminaux et les mots réservés de la notation TTCN-3 sont énumérés dans les Tableaux A.2 et A.3.

Tableau A.2/Z.140 – Liste des symboles terminaux spéciaux TTCN-3

Symboles de début/fin de bloc	{ }
Symboles de début/fin de liste	( )
Symboles de variante	[ ]
Symbole de transition (dans une étendue)	..
Commentaire de ligne et commentaire de bloc	/* */ //
Symbole terminateur de ligne/d'instruction	;
Symboles d'opérateur arithmétique	+ / –

Tableau A.2/Z.140 – Liste des symboles terminaux spéciaux TTCN-3

Symbole d'opérateur de concaténation de chaînes	`&`
Symboles d'opérateur d'équivalence	`!=  ==  >=  <=`
Symboles de délimitation de chaîne	`"  '`
Symboles de structure générique/d'appariement	`?  *`
Symbole d'affectation	`:=`
Affectation d'opération de communication	`->`
Valeurs de chaîne binaire, de chaîne hexadécimale et de chaîne d'octets	`B  H  O`
Exposant de nombre en virgule flottante	`E`

Ce qui suit énumère les identificateurs spéciaux réservés aux fonctions prédéfinies décrites dans l'Annexe D:

```
int2char, char2int, int2unichar, unichar2int, bit2int, hex2int, int2bit,
int2hex, int2oct, int2str, oct2int, str2int, lengthof, sizeof, ischosen,
ispresent
```

Les terminaux TTCN-3 énumérés dans le Tableau A.3 ne doivent pas être utilisés comme identificateurs dans un module TTCN-3. Ces terminaux doivent être écrits en lettres minuscules.

Tableau A.3/Z.140 – Liste des terminaux TTCN-3 qui sont des mots réservés

action	**fail**	**named**	**self**
activate	**false**	**none**	**send**
address	**float**	**nonrecursive**	**sender**
all	**for**	**not**	**set**
alt	**from**	**not4b**	**signature**
and	**function**	**nowait**	**start**
and4b		**null**	**stop**
any	**get**		**sut**
	getcall	**objid**	**system**
bitstring	**getreply**	**octetstring**	
boolean	**goto**	**of**	**template**
	group	**omit**	**testcase**
call		**on**	**timeout**
catch	**hexstring**	**optional**	**timer**
char		**or**	**to**
charstring	**if**	**or4b**	**trigger**
check	**ifpresent**	**out**	**true**
clear	**import**	**override**	**type**
complement	**in**		
component	**inconc**	**param**	**union**
connect	**infinity**	**pass**	**universal**
const	**inout**	**pattern**	**unmap**
control	**integer**	**port**	
create	**interleave**	**procedure**	**value**
			valueof

Tableau A.3/Z.140 – Liste des terminaux TTCN-3 qui sont des mots réservés

deactivate	**label**	**raise**	**var**
disconnect	**language**	**read**	**verdict**
display	**length**	**receive**	**verdicttype**
do	**log**	**record**	
done		**rem**	**while**
	map	**repeat**	**with**
else	**match**	**reply**	
encode	**message**	**return**	**xor**
enumerated	**mixed**	**running**	**xor4b**
error	**mod**	**runs**	
exception	**modifies**		
execute	**module**		
expand	**mtc**		
extension			
external			

A.1.6 Productions BNF de la syntaxe de notation TTCN--3

A.1.6.1 Module de notation TTCN

```
1.  TTCN3Module ::= TTCN3ModuleKeyword TTCN3ModuleId [ModuleParList]
                BeginChar
                [ModuleDefinitionsPart]
                [ModuleControlPart]
                EndChar
                [WithStatement] [SemiColon]
2.  TTCN3ModuleKeyword ::= "module"
3.  TTCN3ModuleId ::= ModuleIdentifier [DefinitiveIdentifier]
4.  ModuleIdentifier ::= Identifier
5.  DefinitiveIdentifier ::= Dot ObjectIdentifierKeyword "{"
                DefinitiveObjIdComponentList "}"
6.  DefinitiveObjIdComponentList ::= {DefinitiveObjIdComponent}+
7.  DefinitiveObjIdComponent ::= NameForm |
                DefinitiveNumberForm |
                DefinitiveNameAndNumberForm
8.  DefinitiveNumberForm ::= Number
9.  DefinitiveNameAndNumberForm ::= Identifier  "(" DefinitiveNumberForm ")"
10. ModuleParList ::= "(" ModulePar {"," ModulePar} ")"
11. ModulePar ::= [InParKeyword] ModuleParType ModuleParIdentifier
                [AssignmentChar ConstantExpression]
/* SÉMANTIQUE STATIQUE – La valeur de l'expression de constante doit être du
même type que le type indiqué pour le paramètre */
12. ModuleParType ::= Type
13. ModuleParIdentifier ::= Identifier
```

A.1.6.2 Partie définitions d'un module

```
14. ModuleDefinitionsPart ::= ModuleDefinitionsList
15. ModuleDefinitionsList ::= {ModuleDefinition [SemiColon]}+
16. ModuleDefinition ::= (TypeDef |
                ConstDef |
                TemplateDef |
                FunctionDef |
                SignatureDef |
                TestcaseDef |
                NamedAltDef |
```

```
                ImportDef |
                GroupDef |
                ExtFunctionDef |
                ExtConstDef) [WithStatement]
```

A.1.6.2.1 Définitions de type (Typedef)

```
17. TypeDef ::= TypeDefKeyword TypeDefBody
18. TypeDefBody ::= StructuredTypeDef | SubTypeDef
19. TypeDefKeyword ::= "type"
20. StructuredTypeDef ::= RecordDef | UnionDef | SetDef | RecordOfDef |
                SetOfDef | EnumDef | PortDef | ComponentDef
21. RecordDef ::= RecordKeyword StructDefBody
22. RecordKeyword ::= "record"
23. StructDefBody ::= (StructTypeIdentifier  [StructDefFormalParList] |
                AddressKeyword)
                BeginChar
                [StructFieldDef {"," StructFieldDef}]
                EndChar
24. StructTypeIdentifier ::= Identifier
25. StructDefFormalParList ::= "(" StructDefFormalPar {"," StructDefFormalPar}
                ")"
26. StructDefFormalPar ::= FormalValuePar | FormalTypePar
/* SÉMANTIQUE STATIQUE - FormalValuePar doit se réduire à un paramètre de type
"in"*/
27. StructFieldDef ::= Type StructFieldIdentifier [ArrayDef] [SubTypeSpec]
                 [OptionalKeyword]
28. StructFieldIdentifier ::= Identifier
29. OptionalKeyword ::= "optional"
30. UnionDef ::= UnionKeyword UnionDefBody
31. UnionKeyword ::= "union"
32. UnionDefBody ::= (StructTypeIdentifier [StructDefFormalParList] |
                AddressKeyword)
                BeginChar
                UnionFieldDef {"," UnionFieldDef}
                EndChar
33. UnionFieldDef ::= Type StructFieldIdentifier [ArrayDef] [SubTypeSpec]
34. SetDef ::= SetKeyword StructDefBody
35. SetKeyword ::= "set"
36. RecordOfDef ::= RecordKeyword OfKeyword [StringLength]  StructOfDefBody
37. OfKeyword ::= "of"
38. StructOfDefBody ::= Type (StructTypeIdentifier | AddressKeyword)
                 [SubTypeSpec]
39. SetOfDef ::= SetKeyword OfKeyword [StringLength]  StructOfDefBody
40. EnumDef ::= EnumKeyword (EnumTypeIdentifier | AddressKeyword)
                BeginChar
                NamedValueList
                EndChar
41. EnumKeyword ::= "enumerated"
42. EnumTypeIdentifier ::= Identifier
43. NamedValueList ::= NamedValue {"," NamedValue}
44. NamedValue ::= NamedValueIdentifier ["(" Number ")"]
45. NamedValueIdentifier ::= Identifier
46. SubTypeDef ::= Type (SubTypeIdentifier | AddressKeyword)  [ArrayDef]
                 [SubTypeSpec]
47. SubTypeIdentifier ::= Identifier
48. SubTypeSpec ::= AllowedValues | StringLength
/* SÉMANTIQUE STATIQUE - Les valeurs doivent être du même type que le champ à
sous-type*/
49. AllowedValues ::= "(" ValueOrRange {"," ValueOrRange} ")"
50. ValueOrRange ::= IntegerRangeDef | SingleConstExpression
/* SÉMANTIQUE STATIQUE - La production IntegerRangeDef ne doit être utilisée
qu'avec des types à base d'entier */
51. IntegerRangeDef ::= LowerBound ".." UpperBound
```

```
52. StringLength ::= LengthKeyword "(" SingleConstExpression [".." UpperBound]
                ")"
/* SÉMANTIQUE STATIQUE - StringLength ne doit être utilisé qu'avec des types
chaîne ou pour limiter un ensemble (set of) et un enregistrement (record of) */
53. LengthKeyword ::= "length"
54. PortType ::= [GlobalModuleId Dot] PortTypeIdentifier
55. PortDef ::= PortKeyword PortDefBody
56. PortDefBody ::= PortTypeIdentifier PortDefAttribs
57. PortKeyword ::= "port"
58. PortTypeIdentifier ::= Identifier
59. PortDefAttribs ::= MessageAttribs | ProcedureAttribs | MixedAttribs
60. MessageAttribs ::= MessageKeyword
                BeginChar
                {MessageList  [SemiColon]}+
                EndChar
61. MessageList ::= Direction AllOrTypeList
62. Direction ::= InParKeyword | OutParKeyword | InOutParKeyword
63. MessageKeyword ::= "message"
64. AllOrTypeList ::= AllKeyword | TypeList
65. AllKeyword ::= "all"
66. TypeList ::= Type {"," Type}
67. ProcedureAttribs ::= ProcedureKeyword
                BeginChar
                {ProcedureList  [SemiColon]}+
                EndChar
68. ProcedureKeyword ::= "procedure"
69. ProcedureList ::= Direction AllOrSignatureList
70. AllOrSignatureList ::= AllKeyword | SignatureList
71. SignatureList ::= Signature {"," Signature}
72. MixedAttribs ::= MixedKeyword
                BeginChar
                {MixedList  [SemiColon]}+
                EndChar
73. MixedKeyword ::= "mixed"
74. MixedList ::= Direction ProcOrTypeList
75. ProcOrTypeList ::= AllKeyword | (ProcOrType {"," ProcOrType})
76. ProcOrType ::= Signature | Type
77. ComponentDef ::= ComponentKeyword ComponentTypeIdentifier
                BeginChar
                [ComponentDefList]
                EndChar
78. ComponentKeyword ::= "component"
79. ComponentType ::= [GlobalModuleId Dot] ComponentTypeIdentifier
80. ComponentTypeIdentifier ::= Identifier
81. ComponentDefList ::= {ComponentElementDef [SemiColon]}+
82. ComponentElementDef ::= PortInstance | VarInstance | TimerInstance |
                  ConstDef
83. PortInstance ::= PortKeyword PortType PortElement {"," PortElement}
84. PortElement ::= PortIdentifier [ArrayDef]
85. PortIdentifier ::= Identifier
```

A.1.6.2.2 Définitions de constante

```
86. ConstDef ::= ConstKeyword Type ConstList
87. ConstList ::= SingleConstDef  {"," SingleConstDef}
88. SingleConstDef ::= ConstIdentifier [ArrayDef] AssignmentChar
                ConstantExpression
/* SÉMANTIQUE STATIQUE – La valeur de l'expression de constante doit être du
même type indiqué pour la constante */
89. ConstKeyword ::= "const"
90. ConstIdentifier ::= Identifier
```

A.1.6.2.3 Définitions de modèle

```
91. TemplateDef ::= TemplateKeyword BaseTemplate  [DerivedDef]
                AssignmentChar TemplateBody
92. BaseTemplate ::= (Type | Signature) TemplateIdentifier  ["("
                TemplateFormalParList ")"]
93. TemplateKeyword ::= "template"
94. TemplateIdentifier ::= Identifier
95. DerivedDef ::= ModifiesKeyword TemplateRef
96. ModifiesKeyword ::= "modifies"
97. TemplateFormalParList ::= TemplateFormalPar {"," TemplateFormalPar}
98. TemplateFormalPar ::= FormalValuePar |
                FormalTemplatePar
/* SÉMANTIQUE STATIQUE – FormalValuePar doit se réduire à un paramètre de type
"in" */
99. TemplateBody ::= SimpleSpec | FieldSpecList |
                ArrayValueOrAttrib
100. SimpleSpec ::= SingleValueOrAttrib
/* SÉMANTIQUE STATIQUE – SimpleSpec ne doit pas être utilisé pour les types
construits */
101. FieldSpecList ::= "{"[FieldSpec {"," FieldSpec}] "}"
102. FieldSpec ::= FieldReference AssignmentChar TemplateBody
103. FieldReference ::= RecordRef | ArrayOrBitRef | ParRef
104. RecordRef ::= StructFieldIdentifier
105. ParRef ::= SignatureParIdentifier
/* SÉMANTIQUE OPÉRATIONNELLE – SignatureParIdentifier doit être un identificateur
de paramètre formel issu de la définition associée de signature */
106. SignatureParIdentifier ::= ValueParIdentifier
107. ArrayOrBitRef ::= "[" FieldOrBitNumber "]"
/* SÉMANTIQUE STATIQUE – ArrayRef doit être facultativement utilisé pour les
types de séquence tabulaire et pour les notations ASN.1 SET OF et SEQUENCE OF
ainsi que pour les notations TTCN suivantes: record, record of, set et set
Identifier of. La même notation peut être utilisée pour une référence de bit
située à l'intérieur d'un type chaîne binaire en notation ASN.1 ou TTCN */
108. FieldOrBitNumber ::= SingleExpression
/* SÉMANTIQUE STATIQUE – SingleExpression se réduira à une valeur de type
entier */
109. SingleValueOrAttrib ::= MatchingSymbol [ExtraMatchingAttributes] |
                SingleExpression [ExtraMatchingAttributes] |
                TemplateRefWithParList
/* SÉMANTIQUE STATIQUE – VariableIdentifier (obtenu par singleExpression) ne
peut être utilisé que dans des définitions de modèle en ligne afin de faire
référence à des variables dans la portée actuelle */
110. ArrayValueOrAttrib ::= "{" ArrayElementSpecList "}"
111. ArrayElementSpecList ::= ArrayElementSpec {"," ArrayElementSpec}
112. ArrayElementSpec ::= NotUsedSymbol | TemplateBody
113. NotUsedSymbol ::= Dash
114. MatchingSymbol ::= Complement | Omit | AnyValue | AnyOrOmit | ValueList |
                IntegerRange | BitStringMatch | HexStringMatch |
                OctetStringMatch | CharStringMatch
115. ExtraMatchingAttributes ::= LengthMatch | IfPresentMatch
116. BitStringMatch ::= "'" {BinOrMatch} "'" B
117. BinOrMatch ::= Bin | AnyValue | AnyOrOmit
118. HexStringMatch ::= "'" {HexOrMatch} "'" H
119. HexOrMatch ::= Hex | AnyValue | AnyOrOmit
120. OctetStringMatch ::= "'" {OctOrMatch} "'" O
121. OctOrMatch ::= Oct | AnyValue | AnyOrOmit
122. CharStringMatch ::= PatternKeyword CharStringPattern {StringOp
                CharStringPattern}
/* SÉMANTIQUE STATIQUE – Toutes les structures CharStringPatterns doivent se
réduire au même type de caractère ou de chaîne de caractères */
123. CharStringPattern ::= CharStringValue | TemplateRefWithParList
124. PatternKeyword ::= "pattern"
125. Complement ::= ComplementKeyword (SingleConstExpression | ValueList)
```

```
126. ComplementKeyword ::= "complement"
127. Omit ::= OmitKeyword
128. OmitKeyword ::= "omit"
129. AnyValue ::= "?"
130. AnyOrOmit ::= "*"
131. ValueList ::= "(" SingleConstExpression {"," SingleConstExpression}+ ")"
132. LengthMatch ::= StringLength
133. IfPresentMatch ::= IfPresentKeyword
134. IfPresentKeyword ::= "ifpresent"
135. IntegerRange ::= "(" LowerBound ".." UpperBound ")"
136. LowerBound ::= SingleConstExpression | Minus InfinityKeyword
137. UpperBound ::= SingleConstExpression | InfinityKeyword
138. InfinityKeyword ::= "infinity"
139. TemplateInstance ::= InLineTemplate
140. TemplateRefWithParList ::= [GlobalModuleId Dot] TemplateIdentifier
               [TemplateActualParList] | TemplateParIdentifier
141. TemplateRef ::= [GlobalModuleId Dot] TemplateIdentifier |
               TemplateParIdentifier
142. InLineTemplate ::= [(Type | Signature) Colon] [DerivedDef AssignmentChar]
               TemplateBody
/* SÉMANTIQUE STATIQUE – Le champ de type ne peut être omis quand le type est
implicitement univoque */
143. TemplateActualParList ::= "(" TemplateActualPar {"," TemplateActualPar} ")"
144. TemplateActualPar ::= TemplateInstance
/* SÉMANTIQUE STATIQUE – Lorsque le paramètre formel correspondant n'est pas de
type modèle, la production TemplateInstance doit se réduire à une ou plusieurs
expressions simples SingleExpressions */
145. TemplateOps ::= MatchOp | ValueofOp
146. MatchOp ::= MatchKeyword "(" Expression "," TemplateInstance")"
/* SÉMANTIQUE STATIQUE – Le type de la valeur renvoyée par l'expression doit
être le même que le type de modèle et chaque champ du modèle doit se réduire à
une unique valeur */
147. MatchKeyword ::= "match"
148. ValueofOp ::= ValueofKeyword "(" TemplateInstance")"
149. ValueofKeyword ::= "valueof"
```

A.1.6.2.4 Définitions de fonction

```
150. FunctionDef ::= FunctionKeyword FunctionIdentifier
               "("[FunctionFormalParList] ")" [RunsOnSpec] [ReturnType]
               BeginChar
               FunctionBody
               EndChar
151. FunctionKeyword ::= "function"
152. FunctionIdentifier ::= Identifier
153. FunctionFormalParList ::= FunctionFormalPar {"," FunctionFormalPar}
154. FunctionFormalPar ::= FormalValuePar |
               FormalTimerPar |
               FormalTemplatePar |
               FormalPortPar
155. ReturnType ::= ReturnKeyword Type
156. ReturnKeyword ::= "return"
157. RunsOnSpec ::= RunsKeyword OnKeyword (ComponentType | MTCKeyword)
158. RunsKeyword ::= "runs"
159. OnKeyword ::= "on"
160. MTCKeyword ::= "mtc"
161. FunctionBody ::= [FunctionStatementOrDefList]
162. FunctionStatementOrDefList ::= {FunctionStatementOrDef [SemiColon]}+
163. FunctionStatementOrDef ::= FunctionLocalDef |
               FunctionLocalInst |
               FunctionStatement
164. FunctionLocalInst ::= VarInstance |
               TimerInstance
165. FunctionLocalDef ::= ConstDef
```

```
166. FunctionStatement ::= ConfigurationStatements |
               TimerStatements |
               CommunicationStatements |
               BasicStatements |
               BehaviourStatements |
               VerdictStatements |
               SUTStatements
167. FunctionInstance ::= FunctionRef "(" [FunctionActualParList] ")"
168. FunctionRef ::= [GlobalModuleId Dot] FunctionIdentifier
169. FunctionActualParList ::= FunctionActualPar {"," FunctionActualPar}
170. FunctionActualPar ::= TimerRef |
               TemplateInstance |
               Port |
               ComponentRef
/* SÉMANTIQUE STATIQUE – Lorsque le paramètre formel correspondant n'est pas de
type modèle, la production TemplateInstance doit se réduire à une ou plusieurs
expressions SingleExpression c'est-à-dire qu'elle équivaut à la production
Expression */
```

A.1.6.2.5 Définitions de signature

```
171. SignatureDef ::= SignatureKeyword SignatureIdentifier
               "("[SignatureFormalParList] ")" [ReturnType]
               [ExceptionSpec]
172. SignatureKeyword ::= "signature"
173. SignatureIdentifier ::= Identifier
174. SignatureFormalParList ::= SignatureFormalPar {"," SignatureFormalPar}
175. SignatureFormalPar ::= FormalValuePar
176. ExceptionSpec ::= ExceptionKeyword "(" ExceptionTypeList ")"
177. ExceptionKeyword ::= "exception"
178. ExceptionTypeList ::= Type {"," Type}
179. Signature ::= [GlobalModuleId Dot] SignatureIdentifier
```

A.1.6.2.6 Définition de test élémentaire

```
180. TestcaseDef ::= TestcaseKeyword TestcaseIdentifier
               "("[TestcaseFormalParList] ")" ConfigSpec
               BeginChar
               FunctionBody
               EndChar
181. TestcaseKeyword ::= "testcase"
182. TestcaseIdentifier ::= Identifier
183. TestcaseFormalParList ::= TestcaseFormalPar {"," TestcaseFormalPar}
184. TestcaseFormalPar ::= FormalValuePar |
               FormalTemplatePar
185. ConfigSpec ::= RunsOnSpec [SystemSpec]
186. SystemSpec ::= SystemKeyword ComponentType
187. SystemKeyword ::= "system"
188. TestcaseInstance ::= ExecuteKeyword "(" TestcaseRef "("
                [TestcaseActualParList] ")" ["," TimerValue]   ")"
189. ExecuteKeyword ::= "execute"
190. TestcaseRef ::= [GlobalModuleId Dot] TestcaseIdentifier
191. TestcaseActualParList ::= TestcaseActualPar {"," TestcaseActualPar}
192. TestcaseActualPar ::=
               TemplateInstance
/* SÉMANTIQUE STATIQUE – Lorsque le paramètre formel correspondant n'est pas de
type modèle, la production TemplateInstance doit se réduire à une ou plusieurs
expressions SingleExpressions c'est-à-dire qu'elle équivaut à la production
Expression */
```

A.1.6.2.7 Définition d'élément nommé (NamedAlt)

```
193. NamedAltDef ::= NamedKeyword AltKeyword NamedAltIdentifier
                "("  [NamedAltFormalParList] ")"
                BeginChar
                AltGuardList                    EndChar
194. NamedKeyword ::= "named"
195. NamedAltIdentifier ::= Identifier
196. NamedAltFormalParList ::= NamedAltFormalPar {"," NamedAltFormalPar}
197. NamedAltFormalPar ::= FormalValuePar |
                FormalTimerPar |
                FormalTemplatePar |
                FormalPortPar
198. NamedAltInstance ::= NamedAltRef  "(" [NamedAltActualParList]")"
199. NamedAltRef ::= [GlobalModuleId Dot] NamedAltIdentifier
200. NamedAltActualParList ::= NamedAltActualPar {"," NamedAltActualPar}
201. NamedAltActualPar ::=
                TimerRef |
                TemplateInstance |
                Port |
                ComponentRef
/* SÉMANTIQUE STATIQUE – Lorsque le paramètre formel correspondant n'est pas de
type modèle, la production TemplateInstance doit se réduire à une ou plusieurs
SingleExpressions c'est-à-dire qu'elle équivaut à la production Expression */
```

A.1.6.2.8 Définition d'importation

```
202. ImportDef ::= ImportKeyword ImportSpec
203. ImportKeyword ::= "import"
204. ImportSpec ::= ImportAllSpec |
                ImportGroupSpec |
                ImportTypeDefSpec |
                ImportTemplateSpec |
                ImportConstSpec |
                ImportTestcaseSpec |
                ImportNamedAltSpec |
                ImportFunctionSpec |
                ImportSignatureSpec
205. ImportAllSpec ::= AllKeyword [DefKeyword] ImportFromSpec
206. ImportFromSpec ::= FromKeyword ModuleId [NonRecursiveKeyword]
207. ModuleId ::= GlobalModuleId [LanguageSpec]
/* SÉMANTIQUE STATIQUE – LanguageSpec ne peut être omis que si le module
référencé contient une notation TTCN-3 */
208. LanguageKeyword ::= "language"
209. LanguageSpec ::= LanguageKeyword  FreeText
210. GlobalModuleId ::= ModuleIdentifier [Dot ObjectIdentifierValue]
211. DefKeyword ::= TypeDefKeyword |
                ConstKeyword |
                TemplateKeyword |
                TestcaseKeyword |
                FunctionKeyword |
                SignatureKeyword |
                NamedKeyword AltKeyword
212. NonRecursiveKeyword ::= "nonrecursive"
213. ImportGroupSpec ::= GroupKeyword GroupIdentifier {"," GroupIdentifier}
                ImportFromSpec
214. ImportTypeDefSpec ::= TypeDefKeyword TypeDefIdentifier {","
                TypeDefIdentifier} ImportFromSpec
215. TypeDefIdentifier ::= StructTypeIdentifier |
                EnumTypeIdentifier |
                PortTypeIdentifier |
                ComponentTypeIdentifier |
                SubTypeIdentifier
```

```
216. ImportTemplateSpec ::= TemplateKeyword TemplateIdentifier {","
               TemplateIdentifier} ImportFromSpec
217. ImportConstSpec ::= ConstKeyword ConstIdentifier {"," ConstIdentifier}
               ImportFromSpec
218. ImportTestcaseSpec ::= TestcaseKeyword TestcaseIdentifier {","
               TestcaseIdentifier} ImportFromSpec
219. ImportFunctionSpec ::= FunctionKeyword FunctionIdentifier {","
               FunctionIdentifier} ImportFromSpec
220. ImportSignatureSpec ::= SignatureKeyword SignatureIdentifier {","
               SignatureIdentifier} ImportFromSpec
221. ImportNamedAltSpec ::= NamedKeyword AltKeyword NamedAltIdentifier {","
               NamedAltIdentifier} ImportFromSpec
```

A.1.6.2.9 Définition de groupe

```
222. GroupDef ::= GroupKeyword GroupIdentifier
               BeginChar
               [ModuleDefinitionsPart]
               EndGroupChar
223. GroupKeyword ::= "group"
224. EndGroupChar ::= "}"
225. GroupIdentifier ::= Identifier
```

A.1.6.2.10 Définitions de fonction externe

```
226. ExtFunctionDef ::= ExtKeyword FunctionKeyword ExtFunctionIdentifier
               "("[FunctionFormalParList] ")" [ReturnType]
227. ExtKeyword ::= "external"
228. ExtFunctionIdentifier ::= Identifier
```

A.1.6.2.11 Définitions de constante externe

```
229. ExtConstDef ::= ExtKeyword ConstKeyword Type ExtConstIdentifier
230. ExtConstIdentifier ::= Identifier
```

A.1.6.3 Partie commande

```
231. ModuleControlPart ::= ControlKeyword
               BeginChar
               ModuleControlBody
               EndChar
               [WithStatement] [SemiColon]
232. ControlKeyword ::= "control"
233. ModuleControlBody ::= [ControlStatementOrDefList]
234. ControlStatementOrDefList ::= {ControlStatementOrDef [SemiColon]}+
235. ControlStatementOrDef ::= FunctionLocalInst |
               ControlStatement |
               FunctionLocalDef
236. ControlStatement ::= TimerStatements |
               BasicStatements |
               BehaviourStatements |
               SUTStatements
```

A.1.6.3.1 Instanciation de variable

```
237. VarInstance ::= VarKeyword Type VarList
238. VarList ::= SingleVarInstance {"," SingleVarInstance}
239. SingleVarInstance ::= VarIdentifier [ArrayDef] [AssignmentChar
               VarInitialValue]
240. VarInitialValue ::= Expression
241. VarKeyword ::= "var"
242. VarIdentifier ::= Identifier
243. VariableRef ::= (VarIdentifier | ValueParIdentifier)
               [ExtendedFieldReference]
```

A.1.6.3.2 Instanciation de temporisateur

```
244. TimerInstance ::= TimerKeyword TimerIdentifier [ArrayDef]
                [AssignmentChar TimerValue]
245. TimerKeyword ::= "timer"
246. TimerIdentifier ::= Identifier
247. TimerValue ::= SingleExpression
/* SÉMANTIQUE STATIQUE – SingleExpression doit se réduire à une valeur de type
float */
248. TimerRef ::= TimerIdentifier [ArrayOrBitRef]|
                TimerParIdentifier [ArrayOrBitRef]
```

A.1.6.3.3 Opérations sur composant

```
249. ConfigurationStatements ::= ConnectStatement |
                MapStatement |
                DisconnectStatement |
                UnmapStatement |
                DoneStatement |
                StartTCStatement |
                StopTCStatement
250. ConfigurationOps ::= CreateOp | SelfOp | SystemOp | MTCOp | RunningOp
251. CreateOp ::= ComponentType Dot CreateKeyword
252. SystemOp ::= "system"
253. SelfOp ::= "self"
254. MTCOp ::= MTCKeyword
255. DoneStatement ::= ComponentId Dot DoneKeyword
256. ComponentId ::= ComponentIdentifier | (AnyKeyword | AllKeyword)
                ComponentKeyword
257. DoneKeyword ::= "done"
258. RunningOp ::= ComponentId Dot RunningKeyword
259. RunningKeyword ::= "running"
260. CreateKeyword ::= "create"
261. ConnectStatement ::= ConnectKeyword PortSpec
262. ConnectKeyword ::= "connect"
263. PortSpec ::= "(" PortRef "," PortRef ")"
264. PortRef ::= ComponentRef Colon Port
265. ComponentRef ::= ComponentIdentifier | SystemOp | SelfOp | MTCOp
266. DisconnectStatement ::= DisconnectKeyword PortSpec
267. DisconnectKeyword ::= "disconnect"
268. MapStatement ::= MapKeyword PortSpec
269. MapKeyword ::= "map"
270. UnmapStatement ::= UnmapKeyword PortSpec
271. UnmapKeyword ::= "unmap"
272. StartTCStatement ::= ComponentIdentifier Dot StartKeyword "("
                FunctionInstance ")"
/* SÉMANTIQUE STATIQUE – L'instance Function ne peut avoir que des paramètres de
type "in" */
273. StartKeyword ::= "start"
274. StopTCStatement ::= StopKeyword
275. ComponentIdentifier ::= VariableRef | FunctionInstance
/* SÉMANTIQUE STATIQUE – La variable associée à VariableRef ou le type "Retour"
associé à l'instance de fonction FunctionInstance doit être du type "composant"
*/
```

A.1.6.3.4 Opérations sur accès

```
276. Port ::= (PortIdentifier | PortParIdentifier) [ArrayOrBitRef]
277. CommunicationStatements ::= SendStatement | CallStatement | ReplyStatement
| RaiseStatement |
                ReceiveStatement | TriggerStatement | GetCallStatement |
                GetReplyStatement | CatchStatement | CheckStatement |
                ClearStatement | StartStatement | StopStatement
278. SendStatement ::= Port Dot PortSendOp
```

```
279. PortSendOp ::= SendOpKeyword "(" SendParameter ")" [ToClause]
280. SendOpKeyword ::= "send"
281. SendParameter ::= TemplateInstance
282. ToClause ::= ToKeyword AddressRef
283. ToKeyword ::= "to"
284. AddressRef ::= VariableRef | FunctionInstance
/* SÉMANTIQUE STATIQUE – Les retours de VariableRef et FunctionInstance doivent
être de type "adresse" ou "composant" */
285. CallStatement ::= Port Dot PortCallOp  [PortCallBody]
286. PortCallOp ::= CallOpKeyword "(" CallParameters ")" [ToClause]
287. CallOpKeyword ::= "call"
288. CallParameters ::= TemplateInstance ["," CallTimerValue]
/* SÉMANTIQUE STATIQUE – Seuls les paramètres "out" peuvent être omis ou
spécifiés avec un attribut d'appariement */
289. CallTimerValue ::= TimerValue | NowaitKeyword
/* SÉMANTIQUE STATIQUE – La valeur doit être de type float */
290. NowaitKeyword ::= "nowait"
291. PortCallBody ::= BeginChar
                     CallBodyStatementList
                     EndChar
292. CallBodyStatementList ::= {CallBodyStatement [SemiColon]}+
293. CallBodyStatement ::= CallBodyGuard  StatementBlock
294. CallBodyGuard ::= AltGuardChar CallBodyOps
295. CallBodyOps ::= GetReplyStatement | CatchStatement
296. ReplyStatement ::= Port Dot PortReplyOp
297. PortReplyOp ::= ReplyKeyword "(" TemplateInstance [ReplyValue]")"
                     [ToClause]
298. ReplyKeyword ::= "reply"
299. ReplyValue ::= ValueKeyword Expression
300. RaiseStatement ::= Port Dot PortRaiseOp
301. PortRaiseOp ::= RaiseKeyword "(" Signature "," TemplateInstance ")"
                     [ToClause]
302. RaiseKeyword ::= "raise"
303. ReceiveStatement ::= PortOrAny Dot PortReceiveOp
304. PortOrAny ::= Port | AnyKeyword PortKeyword
305. PortReceiveOp ::= ReceiveOpKeyword ["(" ReceiveParameter ")"] [FromClause]
                     [PortRedirect]
/* SÉMANTIQUE STATIQUE – L'option PortRedirect ne peut être présente que si
l'option ReceiveParameter est également présente */
306. ReceiveOpKeyword ::= "receive"
307. ReceiveParameter ::= TemplateInstance
308. FromClause ::= FromKeyword AddressRef
309. FromKeyword ::= "from"
310. PortRedirect ::= PortRedirectSymbol (ValueSpec [SenderSpec] | SenderSpec)
311. PortRedirectSymbol ::= "->"
312. ValueSpec ::= ValueKeyword VariableRef
313. ValueKeyword ::= "value"
314. SenderSpec ::= SenderKeyword VariableRef
/* SÉMANTIQUE STATIQUE – La référence de variable doit être de type "adresse" ou
"composant" */
315. SenderKeyword ::= "sender"
316. TriggerStatement ::= PortOrAny Dot PortTriggerOp
317. PortTriggerOp ::= TriggerOpKeyword ["(" ReceiveParameter ")"] [FromClause]
                     [PortRedirect]
/* SÉMANTIQUE STATIQUE – L'option PortRedirect ne peut être présente que si
l'option ReceiveParameter est également présente */
318. TriggerOpKeyword ::= "trigger"
319. GetCallStatement ::= PortOrAny  Dot PortGetCallOp
320. PortGetCallOp ::= GetCallOpKeyword ["(" ReceiveParameter ")"] [FromClause]
                     [PortRedirectWithParam]
/* SÉMANTIQUE STATIQUE – L'option PortRedirectWithParam ne peut être présente
que si l'option ReceiveParameter est également présente */
321. GetCallOpKeyword ::= "getcall"
322. PortRedirectWithParam ::= PortRedirectSymbol RedirectSpec
```

```
323. RedirectSpec ::= ValueSpec  [ParaSpec] [SenderSpec] |
               ParaSpec [SenderSpec] |
               SenderSpec
324. ParaSpec ::= ParaKeyword ParaAssignmentList
325. ParaKeyword ::= "param"
326. ParaAssignmentList ::= "(" (AssignmentList | VariableList) ")"
327. AssignmentList ::= VariableAssignment {"," VariableAssignment}
328. VariableAssignment ::= VariableRef AssignmentChar ParameterIdentifier
/* SÉMANTIQUE STATIQUE – Les identificateurs parameterIdentifiers doivent
provenir de la définition correspondante de signature */
329. ParameterIdentifier ::= ValueParIdentifier |
               TimerParIdentifier |
               TemplateParIdentifier |
               PortParIdentifier
330. VariableList ::= VariableEntry  {"," VariableEntry}
331. VariableEntry ::= VariableRef | NotUsedSymbol
332. GetReplyStatement ::= PortOrAny Dot PortGetReplyOp
333. PortGetReplyOp ::= GetReplyOpKeyword ["(" ReceiveParameter [ValueMatchSpec]
               ")"] [FromClause] [PortRedirectWithParam]
/* SÉMANTIQUE STATIQUE – L'option PortRedirectWithParam ne peut être présente
que si l'option ReceiveParameter est également présente */
334. GetReplyOpKeyword ::= "getreply"
335. ValueMatchSpec ::= ValueKeyword TemplateInstance
336. CheckStatement ::= PortOrAny Dot PortCheckOp
337. PortCheckOp ::= CheckOpKeyword ["(" CheckParameter ")"]
338. CheckOpKeyword ::= "check"
339. CheckParameter ::= PortReceiveOp | PortGetCallOp | PortGetReplyOp |
               PortCatchOp | [FromClause] [PortRedirectSymbol SenderSpec]
340. CatchStatement ::= PortOrAny Dot PortCatchOp
341. PortCatchOp ::= CatchOpKeyword ["("CatchOpParameter ")"] [FromClause]
               [PortRedirect]
/* SÉMANTIQUE STATIQUE – L'option PortRedirect option ne peut être présente que
si l'option CatchOpParameter est également présente */
342. CatchOpKeyword ::= "catch"
343. CatchOpParameter ::= Signature "," TemplateInstance | TimeoutKeyword
344. ClearStatement ::= PortOrAll Dot PortClearOp
345. PortOrAll ::= Port | AllKeyword PortKeyword
346. PortClearOp ::= ClearOpKeyword
347. ClearOpKeyword ::= "clear"
348. StartStatement ::= PortOrAll Dot PortStartOp
349. PortStartOp ::= StartKeyword
350. StopStatement ::= PortOrAll Dot PortStopOp
351. PortStopOp ::= StopKeyword
352. StopKeyword ::= "stop"
353. AnyKeyword ::= "any"
```

A.1.6.3.5 Opérations sur temporisateur

```
354. TimerStatements ::= StartTimerStatement | StopTimerStatement |
               TimeoutStatement
355. TimerOps ::= ReadTimerOp | RunningTimerOp
356. StartTimerStatement ::= TimerRef Dot StartKeyword ["(" TimerValue ")"]
357. StopTimerStatement ::= TimerRefOrAll Dot StopKeyword
358. TimerRefOrAll ::= TimerRef | AllKeyword TimerKeyword
359. ReadTimerOp ::= TimerRef Dot ReadKeyword
360. ReadKeyword ::= "read"
361. RunningTimerOp ::= TimerRefOrAny  Dot RunningKeyword
362. TimeoutStatement ::= TimerRefOrAny  Dot TimeoutKeyword
363. TimerRefOrAny ::= TimerRef | AnyKeyword TimerKeyword
364. TimeoutKeyword ::= "timeout"
```

A.1.6.4 Type

```
365. Type ::= PredefinedType | ReferencedType
366. PredefinedType ::= BitStringKeyword |
                BooleanKeyword |
                CharStringKeyword |
                UniversalCharString |
                CharKeyword |
                UniversalChar |
                IntegerKeyword |
                OctetStringKeyword |
                ObjectIdentifierKeyword |
                HexStringKeyword |
                VerdictKeyword |
                FloatKeyword |
                AddressKeyword
367. BitStringKeyword ::= "bitstring"
368. BooleanKeyword ::= "boolean"
369. IntegerKeyword ::= "integer"
370. OctetStringKeyword ::= "octetstring"
371. ObjectIdentifierKeyword ::= "objid"
372. HexStringKeyword ::= "hexstring"
373. VerdictKeyword ::= "verdict"
374. FloatKeyword ::= "float"
375. AddressKeyword ::= "address"
376. CharStringKeyword ::= "charstring"
377. UniversalCharString ::= UniversalKeyword CharStringKeyword
378. UniversalKeyword ::= "universal"
379. CharKeyword ::= "char"
380. UniversalChar ::= UniversalKeyword CharKeyword
381. ReferencedType ::= [GlobalModuleId Dot] TypeReference
                [ExtendedFieldReference]
382. TypeReference ::= StructTypeIdentifier[TypeActualParList] |
                EnumTypeIdentifier |
                SubTypeIdentifier |
                TypeParIdentifier |
                ComponentTypeIdentifier
383. TypeActualParList ::= "(" TypeActualPar {"," TypeActualPar} ")"
384. TypeActualPar ::= SingleConstExpression | Type
```

A.1.6.4.1 Types de séquence tabulaire

```
385. ArrayDef ::= {"[" ArrayBounds [".." ArrayBounds] "]"}+
386. ArrayBounds ::= SingleConstExpression
/* SÉMANTIQUE STATIQUE – ArrayBounds se réduira à une valeur non négative de
type entier */
```

A.1.6.5 Valeur

```
387. Value ::= PredefinedValue | ReferencedValue
388. PredefinedValue ::= BitStringValue |
                BooleanValue |
                CharStringValue |
                IntegerValue |
                OctetStringValue |
                ObjectIdentifierValue |
                HexStringValue |
                VerdictValue |
                EnumeratedValue |
                FloatValue |
                AddressValue
389. BitStringValue ::= Bstring
390. BooleanValue ::= "true" | false
391. IntegerValue ::= Number
```

```
392. OctetStringValue ::= Ostring
393. ObjectIdentifierValue ::= ObjectIdentifierKeyword  "{" ObjIdComponentList
                "}"
/* SÉMANTIQUE STATIQUE – ReferencedValue doit être de type "identificateur
d'objet"*/
394. ObjIdComponentList ::= {ObjIdComponent}+
395. ObjIdComponent ::= NameForm |
                NumberForm |
                NameAndNumberForm
396. NumberForm ::= Number | ReferencedValue
/* SÉMANTIQUE STATIQUE – referencedValue doit être le type "entier" et avoir une
valeur non négative */
397. NameAndNumberForm ::= Identifier  NumberForm
398. NameForm ::= Identifier
399. HexStringValue ::= Hstring
400. VerdictValue ::= "pass" | fail | inconc | none | error
401. EnumeratedValue ::= NamedValueIdentifier
402. CharStringValue ::= Cstring | Quadruple | ReferencedValue
/* SÉMANTIQUE STATIQUE – ReferencedValue doit se réduire à un type "chaîne" */
403. Quadruple ::= "(" Group "," Plane "," Row "," Cell ")"
404. Group ::= Number
405. Plane ::= Number
406. Row ::= Number
407. Cell ::= Number
408. FloatValue ::= FloatDotNotation | FloatENotation
409. FloatDotNotation ::= Number Dot DecimalNumber
410. FloatENotation ::= Number [Dot DecimalNumber] Exponential [Minus] Number
411. Exponential ::= E
412. ReferencedValue ::= ValueReference [ExtendedFieldReference]
413. ValueReference ::= [GlobalModuleId Dot] ConstIdentifier |
                ExtConstIdentifier |
                ValueParIdentifier |
                ModuleParIdentifier |
                VarIdentifier
414. Number ::= (NonZeroNum {Num}) | 0
415. NonZeroNum ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
416. DecimalNumber ::= {Num}
417. Num ::= 0 | NonZeroNum
418. Bstring ::= "'" {Bin} "'" B
419. Bin ::= 0 | 1
420. Hstring ::= "'" {Hex} "'" H
421. Hex ::= Num | A | B | C | D | E | F| a | b | c | d | e | f
422. Ostring ::= "'" {Oct} "'" O
423. Oct ::= Hex Hex
424. Cstring ::= """ {Char} """
425. Char ::= /* RÉFÉRENCE – Caractère défini par le type "chaîne de caractères"
                correspondant */
426. Identifier ::= Alpha{AlphaNum | Underscore}
427. Alpha ::= UpperAlpha | LowerAlpha
428. AlphaNum ::= Alpha | Num
429. UpperAlpha ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
                P | Q | R | S | T | U | V | W | X | Y | Z
430. LowerAlpha ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
                p | q | r | s | t | u | v | w | x | y | z
431. ExtendedAlphaNum ::= /* RÉFÉRENCE – Caractère issu de tout jeu de caractère
                défini dans l'ISO/IEC 10646-1 */
432. FreeText ::= """ {ExtendedAlphaNum} """
433. AddressValue ::= "null"
```

A.1.6.6 Paramétrisation

```
434. InParKeyword ::= "in"
435. OutParKeyword ::= "out"
436. InOutParKeyword ::= "inout"
```

```
437. FormalValuePar ::= [(InParKeyword | InOutParKeyword | OutParKeyword)] Type
               ValueParIdentifier
438. ValueParIdentifier ::= Identifier
439. FormalTypePar ::= [InParKeyword] TypeParIdentifier
440. TypeParIdentifier ::= Identifier
441. FormalPortPar ::= [InOutParKeyword] PortTypeIdentifier PortParIdentifier
442. PortParIdentifier ::= Identifier
443. FormalTimerPar ::= [InOutParKeyword] TimerKeyword TimerParIdentifier
444. TimerParIdentifier ::= Identifier
445. FormalTemplatePar ::= [InParKeyword] TemplateKeyword Type
               TemplateParIdentifier
446. TemplateParIdentifier ::= Identifier
```

A.1.6.7 Instruction associative

```
447. WithStatement ::= WithKeyword WithAttribList
448. WithKeyword ::= "with"
449. WithAttribList ::= "{" MultiWithAttrib "}"
450. MultiWithAttrib ::= {SingleWithAttrib [SemiColon]}+
451. SingleWithAttrib ::= AttribKeyword  [OverrideKeyword] [AttribQualifier]
               AttribSpec
452. AttribKeyword ::= EncodeKeyword |
               DisplayKeyword |
               ExtensionKeyword
453. EncodeKeyword ::= "encode"
454. DisplayKeyword ::= "display"
455. ExtensionKeyword ::= "extension"
456. OverrideKeyword ::= "override"
457. AttribQualifier ::= "(" DefOrFieldRefList ")"
458. DefOrFieldRefList ::= DefOrFieldRef {"," DefOrFieldRef}
459. DefOrFieldRef ::= DefinitionRef | FieldReference
460. DefinitionRef ::= StructTypeIdentifier |
               EnumTypeIdentifier |
               PortTypeIdentifier |
               ComponentTypeIdentifier |
               SubTypeIdentifier |
               ConstIdentifier |
               TemplateIdentifier |
               NamedAltIdentifier |
               TestcaseIdentifier |
               FunctionIdentifier |
               SignatureIdentifier
461. AttribSpec ::= FreeText
```

A.1.6.8 Instructions comportementales

```
462. BehaviourStatements ::= TestcaseInstance |
               FunctionInstance |
               ReturnStatement |
               AltConstruct |
               InterleavedConstruct |
               LabelStatement |
               GotoStatement |
               ActivateStatement |
               DeactivateStatement |
               NamedAltInstance
/* SÉMANTIQUE STATIQUE – La production TestcaseInstance ne doit pas être appelée
à partir de l'intérieur d'un test élémentaire en cours d'exécution ni à partir
d'une chaîne de fonctions appelée à partir d'un test élémentaire c'est-à-dire
que les tests élémentaires ne peuvent pas être instanciés à partir de la partie
commande ou à partir de fonctions directement appelées à partir de la partie
commande */
463. VerdictStatements ::= SetLocalVerdict
464. VerdictOps ::= GetLocalVerdict
```

```
465. SetLocalVerdict ::= SetVerdictKeyword "(" SingleExpression ")"
/* SÉMANTIQUE STATIQUE – SingleExpression doit se réduire à une valeur de type
verdict */
/* SÉMANTIQUE STATIQUE – La production SetLocalVerdict ne doit pas être utilisée
pour affecter l'erreur "Valeur" */
466. SetVerdictKeyword ::= VerdictKeyword Dot SetKeyword
467. GetLocalVerdict ::= VerdictKeyword Dot GetKeyword
468. GetKeyword ::= "get"
469. SUTStatements ::= SUTAction "(" (FreeText | TemplateRefWithParList)  ")"
470. SUTAction ::= SUTKeyword Dot ActionKeyword
471. SUTKeyword ::= "sut"
472. ActionKeyword ::= "action"
473. ReturnStatement ::= ReturnKeyword  [Expression]
474. AltConstruct ::= AltKeyword BeginChar AltGuardList EndChar
475. AltKeyword ::= "alt"
476. AltGuardList ::= {AltGuardElement [SemiColon]}+  [ElseStatement
               [SemiColon]]
477. AltGuardElement ::= GuardStatement | ExpandStatement
478. GuardStatement ::= AltGuardChar GuardOp  StatementBlock
479. ExpandStatement ::= "["ExpandKeyword "]" NamedAltInstance
480. ElseStatement ::= "["ElseKeyword "]" StatementBlock
481. ExpandKeyword ::= "expand"
482. AltGuardChar ::= "[" [BooleanExpression] "]"
483. GuardOp ::= TimeoutStatement | ReceiveStatement | TriggerStatement |
               GetCallStatement | CatchStatement | CheckStatement |
               GetReplyStatement | DoneStatement
/* SÉMANTIQUE STATIQUE – La production GuardOp est utilisée avec la partie
commande d'un module. Ne doit contenir que l'instruction "timeout"*/
484. StatementBlock ::= BeginChar [FunctionStatementOrDefList] EndChar
485. InterleavedConstruct ::= InterleavedKeyword BeginChar InterleavedGuardList
               EndChar
486. InterleavedKeyword ::= "interleave"
487. InterleavedGuardList ::= {InterleavedGuardElement [SemiColon]}+
488. InterleavedGuardElement ::= InterleavedGuard InterleavedAction
489. InterleavedGuard ::= "[" "]" GuardOp
490. InterleavedAction ::= StatementBlock
/* SÉMANTIQUE STATIQUE – La production StatementBlock ne doit pas contenir
d'instructions itératives, goto, activate, deactivate, stop, return ni d'appels
de fonction */
491. LabelStatement ::= LabelKeyword LabelIdentifier
492. LabelKeyword ::= "label"
493. LabelIdentifier ::= Identifier
494. GotoStatement ::= GotoKeyword (LabelIdentifier | AltKeyword)
/* SÉMANTIQUE STATIQUE – L'option AltKeyword ne peut être utilisée que dans une
structure ALT */
495. GotoKeyword ::= "goto"
496. ActivateStatement ::= ActivateKeyword "(" NamedAltList ")"
497. ActivateKeyword ::= "activate"
498. NamedAltList ::= NamedAltInstance {"," NamedAltInstance}
499. DeactivateStatement ::= DeactivateKeyword ["(" NamedAltRefList ")"]
500. DeactivateKeyword ::= "deactivate"
501. NamedAltRefList ::= NamedAltRef {"," NamedAltRef}
```

A.1.6.9 Instructions de base

```
502. BasicStatements ::= Assignment | LogStatement | LoopConstruct |
               ConditionalConstruct
503. Expression ::= SingleExpression | CompoundExpression
/* SÉMANTIQUE STATIQUE – Une expression ne doit pas contenir d'opérations de
configuration ou de verdict dans la partie commande d'un module */
504. CompoundExpression ::= FieldExpressionList | ArrayExpression
505. FieldExpressionList ::= "{" FieldExpressionSpec {"," FieldExpressionSpec}
               "}"
506. FieldExpressionSpec ::= FieldReference AssignmentChar Expression
```

```
507. ArrayExpression ::= "{" [ArrayElementExpressionList] "}"
508. ArrayElementExpressionList ::= NotUsedOrExpression  {","
                NotUsedOrExpression}
509. NotUsedOrExpression ::= Expression | NotUsedSymbol
510. ConstantExpression ::= SingleConstExpression | CompoundConstExpression
511. SingleConstExpression ::= SingleExpression
/* SÉMANTIQUE STATIQUE – La production SingleConstExpression ne doit pas
contenir de variables ou de paramètres de module et doit se réduire à une valeur
constante au moment de la compilation */
512. BooleanExpression ::= SingleExpression
/* SÉMANTIQUE STATIQUE – La production BooleanExpression doit se réduire à une
valeur de type "boolean" */
513. CompoundConstExpression ::= FieldConstExpressionList | ArrayConstExpression
514. FieldConstExpressionList ::= "{" FieldConstExpressionSpec {","
                FieldConstExpressionSpec} "}"
515. FieldConstExpressionSpec ::= FieldReference AssignmentChar
                ConstantExpression
516. ArrayConstExpression ::= "{" [ArrayElementConstExpressionList] "}"
517. ArrayElementConstExpressionList ::= ConstantExpression {","
                ConstantExpression}
518. Assignment ::= VariableRef  ":=" Expression
/* SÉMANTIQUE OPÉRATIONNELLE – L'expression située à droite de l'affectation
doit se réduire à une valeur explicite du type de gauche. */
519. SingleExpression ::= SimpleExpression {BitOp SimpleExpression}
/* SÉMANTIQUE OPÉRATIONNELLE – Si les deux expressions SimpleExpressions et
l'opération BitOp existent, alors ces expressions SimpleExpressions doivent se
réduire à des valeurs spécifiques de types compatibles */
520. SimpleExpression ::= SubExpression [RelOp SubExpression]
/* SÉMANTIQUE OPÉRATIONNELLE – Si les deux expressions SubExpressions et
l'opération RelOp existent, alors ces expressions SubExpressions doivent se
réduire à des valeurs spécifiques de types compatibles. */
/* SÉMANTIQUE OPÉRATIONNELLE – Si l'opération relationnelle RelOp est "<" | ">"
| ">=" | "<=", alors chaque expression SubExpression doit se réduire à un entier
spécifique, Enumerated ou une valeur en virgule flottante (ces valeurs peuvent
être des valeurs TTCN ou ASN.1) */
521. SubExpression ::= Product [ShiftOp Product]
/* SÉMANTIQUE OPÉRATIONNELLE – Chaque production doit se réduire à une valeur
spécifique. Si plusieurs productions existent, l'opérande de droite doit être de
type entier et si l'opération de décalage est '<<' or '>>' alors l'opérande de
gauche doit se réduire à un type bitstring, hexstring, octetstring ou "integer".
Si l'opération de décalage est '<@' or '@>', alors l'opérande de gauche doit
être de type bitstring, hexstring, charstring ou universal charstring */
522. Product ::= Term {AddOp Term}
/* SÉMANTIQUE OPÉRATIONNELLE – Chaque terme doit se réduire à une valeur
spécifique. Si plusieurs termes existent, alors ces termes doivent se réduire à
un type "integer" ou "float".  */
523. Term ::= Factor {MultiplyOp Factor}
/* SÉMANTIQUE OPÉRATIONNELLE – Chaque facteur doit se réduire à une valeur
spécifique. Si plusieurs facteurs existent, alors les facteurs doivent se
réduire à un type "integer" ou "float". */
524. Factor ::= [UnaryOp] Primary
/* SÉMANTIQUE OPÉRATIONNELLE – L'opération primaire doit se réduire à une valeur
spécifique. Si la production UnaryOp existe et a la valeur "not", alors
l'opération primaire doit se réduire à un type BOOLEAN. Si l'opération UnaryOp a
la valeur "+" ou "-", alors l'opération primaire doit se réduire à un type
"integer" ou "float". Si l'opération UnaryOp se réduit à not4b alors l'opération
primaire doit se réduire au type bitstring, hexstring ou octetstring. */
525. Primary ::= OpCall | Value | "(" SingleExpression ")"
526. ExtendedFieldReference ::= {(Dot StructFieldIdentifier | ArrayOrBitRef)}+
527. OpCall ::= ConfigurationOps | VerdictOps | TimerOps | TestcaseInstance |
                FunctionInstance | TemplateOps
528. AddOp ::= "+" | "-"
/* SÉMANTIQUE OPÉRATIONNELLE – Les opérandes des opérateurs "+" ou "-" doivent
être de type "integer" ou "float"(c'est-à-dire prédéfinis en notation TTCN ou
```

```
ASN.1) ou être des dérivés des types "integer" ou "float" (c'est-à-dire être
sous-types) */
529. MultiplyOp ::= "*" | "/" | mod | rem
/* SÉMANTIQUE OPÉRATIONNELLE – Les opérandes des opérateurs "*", "/", rem ou mod
doivent être de type "integer" ou "float"(c'est-à-dire prédéfinis en notation
TTCN ou ASN.1) ou être des dérivés des types "integer" ou "float" (c'est-à-dire
être sous-types). */
530. UnaryOp ::= "+" | "-" | not | not4b
/* SÉMANTIQUE OPÉRATIONNELLE – Les opérandes des opérateurs "+" ou "-" doivent
être de type "integer" ou "float"(c'est-à-dire prédéfinis en notation TTCN ou
ASN.1) ou être des dérivés des types "integer" ou "float" (c'est-à-dire être
sous-types). Les opérandes de l'opérateur "not" doivent être de type "boolean"
(TTCN ou ASN.1) ou des dérivés de type "boolean". Les opérandes des opérateurs
not4b seront de type bitstring, octetstring or hexstring. */
531. RelOp ::= "==" | "<" | ">" | "!=" | ">=" | "<="
/* SÉMANTIQUE OPÉRATIONNELLE – La priorité des opérateurs est définie dans le
Tableau 7 */
532. BitOp ::= "and4b" | xor4b |or4b | and | xor | or | StringOp
/* SÉMANTIQUE OPÉRATIONNELLE – Les opérandes des opérateurs and, or ou xor
doivent être de type "boolean" (TTCN ou ASN.1) ou des dérivés de type "boolean".
Les opérandes des opérateurs and4b, or4b ou xor4b doivent être de type
bitstring, hexstring ou octetstring (TTCN ou ASN.1) ou des dérivés de ces types.
*/
/* SÉMANTIQUE OPÉRATIONNELLE – La priorité des opérateurs est définie dans le
Tableau 7 */
533. StringOp ::= "&"
/* SÉMANTIQUE OPÉRATIONNELLE – Les opérandes des opérateurs string doivent être
bitstring, hexstring, octetstring ou character string */
534. ShiftOp ::= "<<" | ">>" | "<@" | "@>"
535. LogStatement ::= LogKeyword "(" [FreeText] ")"
536. LogKeyword ::= "log"
537. LoopConstruct ::= ForStatement |
                      WhileStatement |
                      DoWhileStatement
538. ForStatement ::= ForKeyword "(" Initial [SemiColon] Final [SemiColon] Step
                      ")"
                      StatementBlock
539. ForKeyword ::= "for"
540. Initial ::= VarInstance | Assignment
541. Final ::= BooleanExpression
542. Step ::= Assignment
543. WhileStatement ::= WhileKeyword "(" BooleanExpression ")"
                      StatementBlock
544. WhileKeyword ::= "while"
545. DoWhileStatement ::= DoKeyword StatementBlock
                      WhileKeyword "(" BooleanExpression ")"
546. DoKeyword ::= "do"
547. ConditionalConstruct ::= IfKeyword "(" BooleanExpression ")"
                      StatementBlock
                      {ElseIfClause}   [ElseClause]
548. IfKeyword ::= "if"
549. ElseIfClause ::= ElseKeyword  IfKeyword "(" BooleanExpression ")"
                      StatementBlock
550. ElseKeyword ::= "else"
551. ElseClause ::= ElseKeyword  StatementBlock
```

A.1.6.10 Productions diverses

```
552. Dot ::= "."
553. Dash ::= "-"
554. Minus ::= Dash
555. SemiColon ::= ";"
556. Colon ::= ":"
557. Underscore ::= "_"
```

```
558. BeginChar ::= "{"
559. EndChar ::= "}"
560. AssignmentChar ::= ":="
```

Annexe B

Sémantique opérationnelle

Cette annexe définit de façon intuitive et univoque la signification d'un comportement TTCN-3. La sémantique opérationnelle ne vise pas à être formelle et la possibilité de construire des preuves mathématiques sur la base de cette sémantique est donc très limitée.

Cette sémantique opérationnelle offre une vue orientée vers les états lors de l'exécution d'un module TTCN. Différentes sortes d'état sont introduites et la signification des différentes structures TTCN-3 est décrite:

1) au moyen d'informations d'état définissant les préconditions d'exécution d'une structure;

2 au moyen de la définition de la façon dont l'exécution d'une structure modifiera un état.

La sémantique opérationnelle est limitée à la signification du comportement en notation TTCN-3, c'est-à-dire aux fonctions, aux tests élémentaires, à la partie commande d'un module et aux structures linguistiques définissant le comportement de test, comme les opérations **`send`** et **`receive`**, ou les instructions **`if-else`**, **`while`**. L'on explique la signification de plusieurs structures TTCN-3 en les remplaçant par d'autres structures linguistiques. Par exemple, les variantes nommées sont des macros et l'on explique entièrement leur signification en remplaçant toutes les références de macro par les définitions de macro correspondantes, ce qui inclut le traitement du comportement par défaut.

Le plus souvent, la définition sémantique d'un langage est fondée sur une description en syntaxe abstraite de l'arborescence du code à décrire. Cette sémantique ne s'applique pas à la syntaxe abstraite de l'arborescence mais nécessite une représentation graphique, par graphes orientés, des descriptions comportementales de la notation TTCN-3. Un graphe orienté décrit le flux des signaux de commande lors de l'exécution d'un test élémentaire, d'une fonction ou de la partie commande d'un module. Le mappage des descriptions comportementales TTCN-3 sur les graphes orientés est direct.

B.1 Structure de cette annexe

Cette annexe est structurée en deux parties:

1) la première partie (voir § B.2) définit la signification de la notation abrégée TTCN-3 et les notations de macro en les remplaçant par d'autres structures linguistiques TTCN-3. Ces remplacements dans un module TTCN-3 peuvent être considérés comme une étape de prétraitement avant que le module puisse être interprété conformément à la description de sémantique opérationnelle ci-après;

2) la deuxième partie (voir § B.3) décrit la sémantique opérationnelle de la notation TTCN-3 par interprétation de graphes orientés et par modification d'états.

B.2 Remplacement des notations abrégées et des macro-instructions

Les notations abrégées doivent être étendues et les références de macro doivent être remplacées par les définitions correspondantes à un niveau textuel avant que cette sémantique opérationnelle puisse être utilisée pour l'explication du comportement TTCN-3.

Les notations abrégées de la notation TTCN-3 sont les suivantes:

- opérations de réception autonomes;
- opérations **trigger**;
- usages du mot clé **any** dans les opérations de temporisation et de réception;
- usages du mot clé **all** dans les opérations de temporisation et de réception;
- instructions **return** et **stop** manquantes à la fin de définitions de fonction et de test élémentaire.

Les macros TTCN-3 sont des variantes nommées, c'est-à-dire des définitions de type **named alt**. Elles sont appelées comme suit:

- explicitement au lieu d'une instruction **alt**, c'est-à-dire qu'elles apparaissent comme un appel de fonction;
- explicitement dans les instructions **alt** par utilisation d'un mot clé **expand**;
- implicitement si elles sont référencées en tant que comportement par défaut dans des instructions **activate** et **deactivate**.

En plus des notations abrégées et des appels de macro, la sémantique opérationnelle nécessite un traitement spécial pour les paramètres de module et les constantes globales, c'est-à-dire les constantes qui sont définies dans la partie définitions du module. Toutes les références aux paramètres de module et aux constantes globales doivent être remplacées par des valeurs concrètes. En d'autres termes, l'on part du principe que la valeur des paramètres de module et des constantes globales peut être déterminée avant que la sémantique opérationnelle devienne applicable.

NOTE 1 – Le traitement des paramètres de module et des constantes globales dans la sémantique opérationnelle sera différent de leur traitement dans un compilateur TTCN-3. La sémantique opérationnelle décrit la signification du comportement TTCN-3 et n'est pas un guide de réalisation d'un compilateur TTCN-3.

NOTE 2 – La sémantique opérationnelle traite les paramètres et les constantes locales contenus dans des composants de test, dans des tests élémentaires, dans des fonctions et dans des commandes de module comme des variables. L'utilisation erronée de constantes locales ou de paramètres **in, out** et **inout** doit faire l'objet d'une vérification statique.

B.2.1 Ordre des étapes de remplacement

Les remplacements textuels de notations abrégées, d'appels de macro, de constantes globales et de paramètres de module doivent être effectués dans l'ordre suivant:

1) adjonction d'instructions **stop** et **return** dans une commande de module, des fonctions et des tests élémentaires;
2) remplacement de constantes globales et de paramètres de module par des valeurs concrètes;
3) imbrication d'opérations de réception autonomes dans des instructions **alt**;
4) expansion en macro d'*appels de macro* purs, c'est-à-dire:
   - expansions explicites d'instructions **alt** comportant le mot clé **expand** (et se rapportant à une définition de variante nommée **named alt**)
   - expansion explicite d'appels de définitions de variantes nommées.
5) expansion d'instructions d'entrelacement **interleave**;
6) expansion de comportements par défaut;
7) remplacement de toutes les opérations de déclenchement **trigger** par des opérations de réception **receive** et par des instructions **goto** équivalentes;
8) remplacement de toutes les utilisations des mots clés **any** et **all** dans les opérations de temporisation et d'accès.

NOTE – Si cet ordre des étapes de remplacement n'était pas respecté, le résultat des remplacements ne représenterait pas le comportement défini.

B.2.2 Adjonction des opérations d'arrêt et de retour aux descriptions comportementales

La notation TTCN-3 permet de laisser des commandes de module, des tests élémentaires et des fonctions qui ne renvoient aucune valeur sans spécifier d'opération **stop** ou **return** explicite. Pour la sémantique opérationnelle, l'on part du principe que les opérations de retour et d'arrêt manquantes sont ajoutées, c'est-à-dire que des opérations d'arrêt sont ajoutées dans les commandes de module et dans les tests élémentaires, tandis que des opérations de retour sont ajoutées dans des fonctions.

Exemple:

```
// Définition de fonction et de test élémentaire sans retour explicite et
// instructions d'arrêt à la fin de leur description de comportement

     function MyFunction(inout integer MyPar) {
                                // MyFunction ne renvoie pas de valeur mais
                                // change la valeur
       MyPar := 10 * MyPar1;    // depuis MyPar qui est transmis par référence
       if (MyPar == 999) stop;  // Ferme l'exécution si MyPar a la valeur 999

       // Retour IMPLICIT si MyPar != 999
     }

     testcase MyTestCase() runs on MyMTCtype {
       MyMTCbehaviour();        // Fonction qui définit un comportement MTC

       // Arrêt IMPLICITE après retour de MyMTCbehaviour
     }


// MyFunction et MyTestCase après adjonction d'opérations explicites de retour
// et d'arrêt

     function MyFunction(inout integer MyPar) {
                                // MyFunction ne renvoie pas de valeur mais
                                // change la valeur
       MyPar := 10 * MyPar1;    // de MyPar qui est transmis par référence
       if (MyPar == 999) stop;  // Ferme l'exécution si MyPar a la valeur 999

          return;               // Retour explicite
       }

       testcase MyTestCase() runs on MyMTCtype {

       MyMTCbehaviour();        // Fonction qui définit un comportement MTC

       stop;                    // arrêt EXPLICITE
       }
```

B.2.3 Remplacement des constantes mondiales et des paramètres de module mondiaux

Les constantes déclarées dans la partie définitions d'un module sont globales pour la partie commande d'un module et pour tous les composants de test qui sont créés au cours de l'exécution d'un module TTCN-3. Les paramètres de module sont censés être des constantes globales au moment de l'exécution.

Toutes les références à des constantes globales et à des paramètres de module doivent être remplacées par les valeurs réelles avant que la sémantique opérationnelle commence l'interprétation du module. Si la valeur d'une constante ou d'un paramètre de module est donnée sous la forme d'une

expression, celle-ci doit être évaluée. Ensuite, le résultat de l'évaluation doit remplacer toutes les références de la constante ou du paramètre de module.

B.2.4 Imbrication d'opérations de réception dans des instructions d'alternative

Les opérations de réception sont, en notation TTCN-3: **receive**, **trigger**, **getcall**, **getreply**, **catch**, **check**, **timeout**, et **done**.

NOTE – Les opérations **receive**, **trigger**, **getcall**, **getreply**, **catch** et **check** fonctionnent aux accès. Elles permettent un branchement dû à la réception de messages, à des appels de procédure, à des réponses et à des exceptions. Les opérations **timeout** et **done** ne sont pas des opérations réelles de réception, mais elles peuvent être utilisées de la même façon que les opérations de réception, c'est-à-dire comme des variantes dans des instructions **alt**. Donc, la sémantique opérationnelle traite **timeout** et **done** comme des opérations de réception.

Une opération de réception peut être utilisée comme instruction autonome dans une fonction, dans une variante nommée ou dans un test élémentaire. Dans ce cas, l'opération de réception est considérée comme étant une notation abrégée représentant une instruction d'alternative avec une seule variante, définie par l'opération de réception. En sémantique opérationnelle, une instruction d'alternative dans laquelle l'instruction de réception est imbriquée doit remplacer toutes les instances autonomes des opérations de réception.

Exemple:

```
// L'instance autonome de
      :
     MyCL.trigger(MyType:*);
      :

// doit être remplacée par
      :
     alt {
       [] MyCL.trigger (MyType:*);
     }
      :

// ou l'instance
      :
     MyPTC.done;
      :

// doit être remplacée par
      :
     alt {
       [] MyPTC.done;
     }
      :
```

B.2.5 Expansion de macro-instruction

L'expansion de macro-instruction en notation TTCN-3 est associée à l'utilisation de variantes nommées (définitions de type **named alt**) contenues dans des instructions **alt** ou remplaçant de telles instructions, c'est-à-dire que la définition de variante nommée est référencée comme un appel de fonction dans une séquence d'instructions.

B.2.5.1 Expansion de variantes nommées contenues dans des instructions d'alternative

L'expansion de variantes nommées contenues dans des instructions **alt** est associée aux branches d'alternative qui sont indiquées par le mot clé **expand** entre crochets, suivi d'une référence à une définition de variante nommée (en tant que seule instruction de cette branche). Dans ce cas, les branches d'alternative de la variante nommée en référence remplacent la branche contenant le mot clé **expand**. En sémantique opérationnelle, l'on part du principe que ce remplacement est effectué au

niveau syntaxique. L'on trouvera un exemple de cette expansion dans le corps de la présente Recommandation.

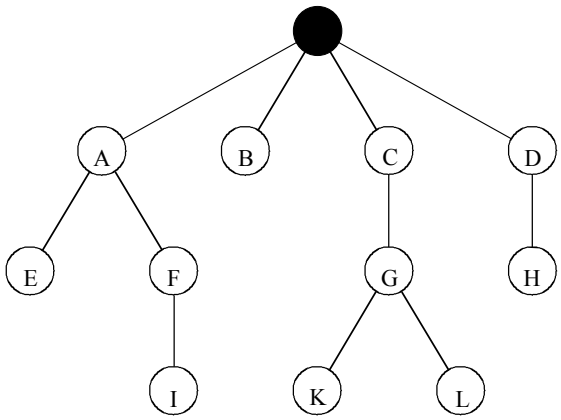
B.2.5.2 Appel explicite d'une variante nommée

Les variantes nommées peuvent également être référencées comme un appel de fonction dans une séquence d'instructions. Dans ce cas, la référence doit être développée par la définition **`named alt`** correspondante. L'on trouvera un exemple de cette expansion dans le corps de la présente Recommandation.

B.2.6 Remplacement de la création syntaxique d'entrelacement

La signification de l'instruction **`interleave`** est définie par son remplacement par une série d'instructions **`alt`** imbriquées et ayant la même signification. L'algorithme de construction du remplacement d'une instruction **`interleave`** est décrit ci-après. Le remplacement doit être effectué au niveau syntaxique.

Une série d'instructions **`alt`** imbriquées peut être décrite au moyen d'une arborescence. Trois nœuds représentent les instructions contenues dans les instructions **`alt`**. Un embranchement indique une instruction **`alt`** et les instructions contenues dans la même branche correspondent à des instructions faisant partie de la même variante. C'est ce que représente schématiquement la Figure B.1. La Figure B.1 a) présente une arborescence, tandis que la Figure B.1 b) donne la représentation correspondante sous forme d'une série d'instructions **`alt`** imbriquées.



a) Vue arborescente

```
alt {

     []   A {
                alt {
                    [] E ;
                    [] F {
                                I
                          }
                }
          }
     []   B;
     []   C {
                G;
                alt {
                [] K;
                [] L;
                }
          }
      []  D{
                H
          }
}
```

T1012780-01

b) Représentation en notation TTCN-3 de la vue a)

Figure B.1/Z.140 – Instructions "alt" imbriquées et représentation arborescente correspondante

La construction d'une représentation arborescente d'une instruction **`interleave`** est présentée ci-après. La transformation de l'arborescence en série d'instructions **`alt`** imbriquées est évidente et ne nécessite pas d'autre explication.

Une instruction **`interleave`** peut être vue comme un ensemble partiellement ordonné ***POS*** d'instructions TTCN-3 autorisées. Formellement:

- ***POS*** = (***S***, <)

où:

$\boldsymbol{S}$ ensemble d'instructions TTCN-3 permises;

$< \subseteq (\boldsymbol{S} \times \boldsymbol{S})$ relation d'ordre, réflexive et transitive.

Le terme *instructions TTCN-3 permises* se rapporte au fait que l'utilisation des instructions de transfert de commande **`for`**, **`while`**, **`do-while`**, **`goto`**, **`activate`**, **`deactivate`**, **`stop`**, **`return`** et des appels de fonctions définis par l'utilisateur, contenant des opérations de communication, n'est pas autorisée dans les instructions **`interleave`**. Il n'est pas non plus permis de sauvegarder des branches d'instruction **`interleave`** au moyen d'expressions booléennes, de développer des instructions **`interleave`** au moyen de variantes nommées ou de spécifier des branches **`else`**.

Les fonctions suivantes doivent être définies pour l'algorithme de construction:

- La fonction *DISCARD* supprime un élément ***s*** d'un ensemble partiellement ordonné ***POS*** et renvoie l'ensemble partiellement ordonné résultant ***POS'***:

  $$DISCARD(\boldsymbol{s}, \boldsymbol{POS}) = \boldsymbol{POS'}$$

  où:

  $\boldsymbol{POS'} = (\boldsymbol{S'}, <')$; et

  $\boldsymbol{S'} = \boldsymbol{S} \backslash \{\boldsymbol{s}\}$; et

  $<' = < \cap (\boldsymbol{S} \backslash \{\boldsymbol{s}\} \times \boldsymbol{S} \backslash \{\boldsymbol{s}\})$.

- La fonction *ENABLED* prend un ensemble partiellement ordonné $\boldsymbol{POS} = (\boldsymbol{S}, <)$ et renvoie tous les éléments qui n'ont pas de prédécesseurs dans l'ensemble ***POS***:

  $$ENABLED(\boldsymbol{POS}) = \{ \boldsymbol{s} \mid \boldsymbol{s} \in \boldsymbol{S} \wedge (< \cap (\boldsymbol{S} \times \{\boldsymbol{s}\}) = \varnothing)\}$$

- La fonction *RECEIVING* prend un ensemble d'instructions TTCN-3 ***S*** et renvoie toutes les instructions de réception issues de cet ensemble.

  $$RECEIVING(\boldsymbol{S}) = \{ \boldsymbol{s} \mid \boldsymbol{s} \in \boldsymbol{S} \wedge kind(\boldsymbol{s}) \in \{\texttt{receive}, \texttt{trigger}, \texttt{getcall}, \texttt{getreply}, \texttt{catch}, \texttt{check}, \texttt{done}, \texttt{timeout}\}\}$$

- La fonction *SELECT* sélectionne aléatoirement un élément ***s*** issu d'un ensemble donné ***S*** et renvoie ***s***.

  $$SELECT(\boldsymbol{S}) = \boldsymbol{s} \qquad \text{où } \boldsymbol{s} \in \boldsymbol{S}$$

NOTE – La fonction *kind* contenue dans la fonction *RECEIVING* ci-dessus n'est pas définie formellement. La fonction (ou le type) *kind* renvoie la sorte d'une instruction TTCN-3 donnée.

L'algorithme de construction de l'arborescence est une procédure récursive dans laquelle les nœuds successeurs d'un nœud donné sont construits dans chaque appel récursif. Cette procédure est fournie en notation de pseudo-code de type C faisant appel aux fonctions définies ci-dessus et quelques notations mathématiques complémentaires:

```
CONSTRUCT-SUCCESSORS (treeNode *predecessor, partiallyOrderedSet POS) {
     // treeNode se rapporte au type de nœud de l'arborescence à construire
     // partiallyOrderedSet indique le type d'un ensemble partiellement ordonné
     // d'instructions TTCN-3

     var statement myStmt;                  // pour la mémorisation d'une
                                            // instruction TTCN-3
     var treeNode *newSonNode;              // pour le traitement de nouveaux
                                            // nœuds d'arborescence

// EXTRACTION D'ENSEMBLE D'INSTRUCTIONS TTCN-3 N'AYANT PAS DE PRÉDÉCESSEURS DANS
// 'POS'
     var statementSet enabStmts := ENABLED(POS);
                    // toutes les instructions sans prédécesseur
```

```
    var statementSet enabRecStmts := RECEIVING(enabStmts);
                  // instructions réceptrices dans 'enabStmts'
    var statementSet enabNonRecStmts := enabStmts\enabRecStmts;
                  // instructions non réceptrices dans 'enabStmts'

    if (POS == ∅)
      return;          // CRITERE DE TERMINAISON DE LA RECURRENCE
    else {
      if (enabNonRecStmts != ∅) {    // Traitement des instructions non
                                     // réceptrices dans 'enabStmts'
         myStmt := SELECT(enabNonRecStmts);
         newSonNode := create(myStmt, predecessor);
              // Création d'un nouveau nœud d'arbre représentant 'myStmt' dans
              // l'arbre et mise à jour des pointeurs dans
              // 'newSonNode' et 'predecessor'.
         CONSTRUCT-SUCCESSORS(newSonNode, DISCARD(myStmt, POS));
              // PROCHAINE ETAPE DE RECURRENCE
      }
      else { // Traitement des événements de réception avec embranchement dans
             // l'arbre
         for each (myStmt in enabRecStmts) {
             newSonNode := create(myStmt, predecessor); // Nouveau nœud d'arbre
             CONSTRUCT-SUCCESSORS(newSonNode, DISCARD(myStmt, POS));
             // PROCHAINE(S) ETAPE(S) DE RECURRENCE
         }
      }
    }
}
```

Initialement, la fonction CONSTRUCT-SUCCESSORS sera appelée avec un *nœud racine* d'une arborescence vide et avec l'ensemble partiellement ordonné des instructions TTCN-3 décrivant l'instruction **interleave** qui doit être remplacée. Après terminaison, le *nœud racine* peut être utilisé pour accéder à l'arborescence construite.

B.2.7 Expansions de comportement par défaut

Le mécanisme de comportement par défaut de la notation TTCN-3 est défini au moyen d'un mécanisme d'expansion de macro. Ce comportement par défaut doit être fourni sous la forme de définitions **named alt**. Une définition **named alt** utilisée comme comportement par défaut est référencée dans une instruction **activate**. La portée d'un comportement par défaut est déterminée par une instruction **activate** et par les instructions **deactivate** correspondantes ou par une instruction **activate** et la fin de la fonction ou du test élémentaire dans lequel l'instruction **activate** est utilisée. A l'intérieur de cette portée, les variantes de toutes les instructions **alt** sont étendues par le comportement spécifié dans les définitions **named alt** activées. La sémantique opérationnelle part du principe que cette extension est effectuée au niveau syntaxique. L'on trouvera un exemple de ce mécanisme d'extension dans le corps de la présente Recommandation.

B.2.8 Remplacement d'opérations de déclenchement

L'opération **trigger** filtre selon un certain critère d'appariement les messages provenant d'un flux de messages à un accès donné. L'on peut décrire la sémantique de l'opération de déclenchement en remplaçant celle-ci par deux opérations **receive** et par une instruction **goto**. La sémantique opérationnelle part du principe que ce remplacement est effectué au niveau syntaxique.

Exemple:

```
// L'opération de déclenchement suivante ...

     alt  {
          []  MyCL.trigger (MyType:*);
          }
```

```
// doit être remplacée par ...

     alt  {
          []  MyCL.receive (MyType:*);
          []  MyCL.receive {
                 goto alt
          }
     }
```

Si l'instruction de déclenchement est utilisée **trigger** dans une instruction **alt** plus complexe, le remplacement est effectué de la même façon.

Exemple:

```
// L'instruction "alt" suivante contient une instruction de déclenchement …
     alt  {
          []  PCO2.receive {
                 stop;
              }
          []  MyCL.trigger (MyType:*);
          []  PCO3.catch {
                 verdict.set(fail);
                 stop;
              }
       }

// qui sera remplacée par

     alt  {
          []  PCO2.receive {
                 stop;
              }
          []  MyCL.receive (MyType:*);
          []  MyCL.receive {
                 goto alt;
              }
          []  PCO3.catch {
                 verdict.set(fail);
                 stop;
              }
       }
```

B.2.9 Remplacement des mots clés *any* et *all*

L'utilisation du mot clé **any** est autorisée pour les opérations suivantes:

- les opérations de temporisation **running** et **timeout;**
- les opérations de réception **receive**, **trigger**, **getcall**, **getreply**, **catch**, **check**.

L'usage du mot clé **all** est permis pour:

- l'opération de temporisation **stop**;
- les opérations aux accès **start**, **stop** et **clear**.

L'utilisation de deux mots clés est permis pour:

- les opérations **done** et **running** pour composants.

B.2.9.1 Remplacement du mot clé *all* dans les opérations de temporisateur et d'accès

L'application des opérations de temporisation et d'accès est associée à la portée dans laquelle elles sont effectuées. En d'autres termes, le mot clé **all** s'adresse à tous les temporisateurs et à tous les accès identifiés dans l'unité de portée où le mot clé **all** (+ l'opération) est utilisé. Le remplacement des usages du mot clé **all** dans les opérations de temporisation et d'accès est simple.

Chaque utilisation des mots clés **all port** dans une opération **start**, **stop**, ou **clear** doit être remplacée par une opération **start**, **stop**, ou **clear** distincte pour chaque accès connu. Chaque utilisation des mots clés **all timer** dans une opération **stop** doit être remplacée par une opération **stop** distincte pour chaque temporisateur connu.

Exemple:

```
// Part du principe que les accès PCO1, PCO2 et les temporisateurs T1 et T2 sont
// connus

      :
     all port.clear;
      :
      :
     all timer.stop;
      :

// sera remplacé par

      :
     PCO1.clear;
     PCO2.clear;
      :
      :
     T1.stop;
     T2.stop;
      :
```

B.2.9.2 Remplacement du mot clé *any* dans les opérations de temporisation et de réception

L'application des opérations de temporisation et de réception est associée à la portée dans laquelle elles sont effectuées. En d'autres termes, le mot clé **any** s'adresse à tous les temporisateurs et à tous les accès (en cas d'opérations de réception) identifiés dans l'unité de portée où le mot clé **any** (+ l'opération) est utilisé. Le remplacement des usages du mot clé **any** dans les opérations de temporisation et de réception est simple.

Chaque utilisation des mots clés **any port** dans une opération **receive**, **trigger**, **getcall**, **getreply**, **catch** ou **check** doit être remplacée par des opérations en variante distinctes pour chaque accès connu et possible. Le terme "possible" signifie qu'une occurrence de type **any port.receive** n'est applicable qu'aux accès en mode message.

Chaque utilisation des mots clés **any timer** dans une opération **timeout** doit être remplacée par des opérations en variante distinctes pour chaque temporisateur connu dans l'unité de portée.

Exemple:

```
// Part du principe que les accès PCO1, PCO2 et les temporisateurs T1 et T2 sont
// connus

     alt  {
          []  PCO2.receive {
                 aTestStep();
              }
          []  any port.receive {
                 verdict.set(fail);
```

```
                stop;
            }
        []  any timer.timeout {
                verdict.set(fail);
                stop;
            }
    }

// sera remplacée par

    alt  {
        []  PCO2.receive {
                stop;
            }
        []  PCO1.receive {
                verdict.set(fail);
                stop;
            }
        []  PCO1.receive {
                verdict.set(fail);
                stop;
            }
        []  T1.receive {
                verdict.set(fail);
                stop;
            }
        []  T2.receive {
                verdict.set(fail);
                stop;
            }
    }
```

Chaque utilisation des mots clés **any timer** dans une opération **running** doit être remplacée par des opérations **running** distinctes pour chaque temporisateur connu dans l'unité de portée, ces opérations étant combinées au moyen d'opérateurs de type **or**.

Exemple:

```
// Part du principe que les temporisateurs T1 et T2 sont connus dans l'unité
// de portée

     :
    if (any timer.running) {
      verdict.set(fail);
      stop;
    }
     :

// sera remplacée par

     :
    if (T1.running or T2.running) {
      verdict.set(fail);
      stop;
    }
     :
```

B.2.9.3 Les mots clés *any* et *all* dans les opérations de fin d'exécution et d'exécution

Les opérations **any component.done**, **all component.done**, **any component.running** et **all component.running** ne peuvent être exécutées que par le composant MTC. En raison de la création dynamique des composants de test, le composant MTC ne peut pas connaître tous les composants qui ont été créés au cours de l'exécution du test élémentaire. L'exécution de ces

opérations nécessite donc une communication avec le dispositif de test. Les opérations **`any component.done`**, **`all component.done`**, **`any component.running`** et **`all component.running`** sont censées être des commandes de système, c'est-à-dire qu'elles ne peuvent pas être remplacées par d'autres commandes.

B.3 Sémantique des graphes orientés de la notation TTCN-3

La sémantique opérationnelle de la notation TTCN-3 est fondée sur l'interprétation de graphes orientés. Le présent paragraphe présente les graphes orientés (voir § B.3.1); explique la construction des graphes orientés représentant la commande de module, les tests élémentaires, les fonctions et les définitions de type de composant en notation TTCN-3 (voir § B.3.2); définit les états de module et de composant pour la description des états d'exécution d'un module TTCN-3 (voir § B.3.3); décrit le traitement des messages, les appels de procédure distante, les réponses aux appels de procédure distante et les exceptions (voir § B.3.4); explique la procédure d'évaluation de la commande de module et des tests élémentaires (voir § B.3.6) et décrit la signification des différentes instructions TTCN-3 (voir § B.3.7).

B.3.1 Graphes orientés

Un graphe orienté se compose de nœuds et de bords étiquetés. Le parcours d'un graphe orienté décrit le flux de commande au cours de l'exécution d'une description du comportement représenté.

B.3.1.1 Cadre de graphe orienté

Un graphe orienté doit être inscrit dans un cadre qui en définit les bords. Le nom du graphe orienté suit les mots clés **`flow graph`** (qui ne font pas partie du langage noyau de la notation TTCN-3) et doit être inscrit dans le coin supérieur gauche du graphe orienté. L'on admet la convention que le nom du graphe orienté se rapporte à la description du comportement TTCN représenté par le graphe orienté. La Figure B.2 représente un simple graphe orienté.

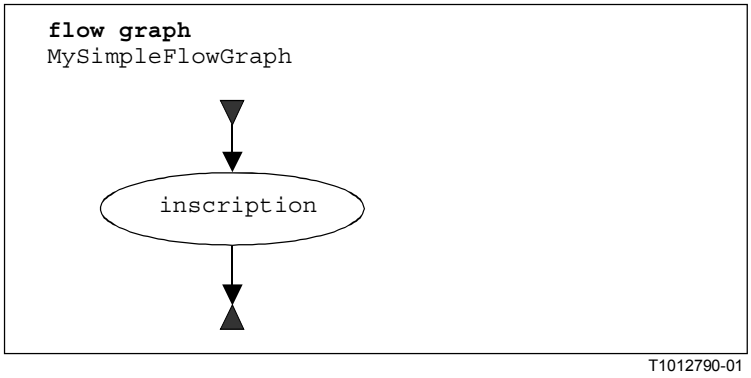


Figure B.2/Z.140 – Simple graphe orienté

B.3.1.2 Nœuds de graphe orienté

Les graphes orientés se composent de *nœuds de début, nœuds de fin, nœuds de base* et *nœuds de référence*.

B.3.1.2.1 Nœuds de début

Les nœuds de début décrivent le point de départ – qui doit être unique – d'un graphe orienté, comme représenté par la Figure B.3 a).

a) Nœud de début de graphe orienté **b) Nœud de fin de graphe orienté**

Figure B.3/Z.140 – Nœuds de début et de fin

B.3.1.2.2 Nœuds de fin

Un nœud de fin décrit une extrémité d'un graphe orienté. Celui-ci peut avoir plusieurs nœuds de fin ou, en cas de boucles, n'avoir aucun nœud de fin. Les nœuds de base (voir § B.3.1.2.3) et les nœuds de référence (voir § B.3.1.2.4) qui n'ont pas de nœuds successeurs doivent être raccordés à un nœud de fin pour indiquer qu'ils décrivent la dernière action d'un chemin dans un graphe orienté. Un nœud de fin est représenté par la Figure B.3 b).

B.3.1.2.3 Nœuds de base

Un nœud de base décrit une unité d'exécution, c'est-à-dire qu'il est exécuté en une seule étape. Un nœud de base possède un type et, selon celui-ci, peut avoir une liste d'attributs associée. Un nœud de base est représenté par la Figure B.4 a).

Dans l'inscription d'un nœud de base, les attributs d'un nœud suivent le type de celui-ci et sont mis entre parenthèses. Les types et les attributs servent à déterminer l'action à effectuer au cours de l'exécution de la structure linguistique représentée. Les attributs fournissent des informations qui doivent être extraites à partir de la structure TTCN-3 correspondante.

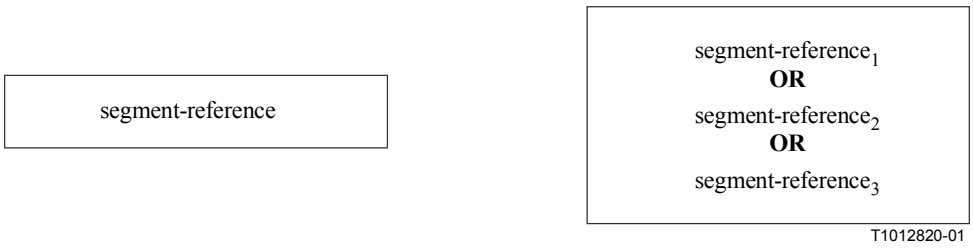
Les attributs possèdent des valeurs que la sémantique opérationnelle extrait par référence au nom d'attribut. Au besoin, il est permis d'affecter des valeurs explicites dans des nœuds de base au moyen de l'affectation '='. Un exemple est donné dans la Figure B.4 b).

a) **b)**

Figure B.4/Z.140 – Nœuds de base avec attributs

B.3.1.2.4 Nœuds de référence

Les nœuds de référence se rapportent aux segments de graphe orienté (voir § B.3.1.4) qui forment des sous-graphes orientés. La signification d'un nœud de référence est définie par son remplacement au moyen du segment de graphe orienté référencé dans le graphe principal. L'inscription nodale du nœud de référence fournit la référence à un segment de graphe orienté. Un nœud de référence est représenté dans la Figure B.5 a).



a) Nœud de référence unique

b) Combinaison par opérateur OU de trois nœuds de référence

Figure B.5/Z.140 – Nœud de référence

B.3.1.2.4.1 Combinaison par opérateur OU des nœuds de référence

Dans certains cas, plusieurs segments de graphe orienté peuvent remplacer un nœud de référence. Un opérateur OR peut alors être utilisé pour faire référence à plusieurs segments de graphe orienté (Figure B.5 b). Dans le graphe orienté proprement dit qui représente la partie commande d'un module, un test élémentaire ou une fonction, une seule variante est déterminée par la structure représentée.

B.3.1.2.4.2 Occurrences multiples de nœuds de référence

Il se peut que la même sorte de nœud de référence apparaisse zéro, une ou plusieurs fois dans un graphe orient. Dans les expressions régulières, la possibilité de répéter des parties d'une expression régulière est décrite au moyen des symboles d'opérateur '+' (une ou plusieurs répétitions) et '*' (zéro, une ou plusieurs répétitions). Comme indiqué dans la Figure B.6, ces opérateurs ont été adaptés aux graphes orientés par introduction de nœuds de référence à double encadrement du symbole d'opérateur associé.

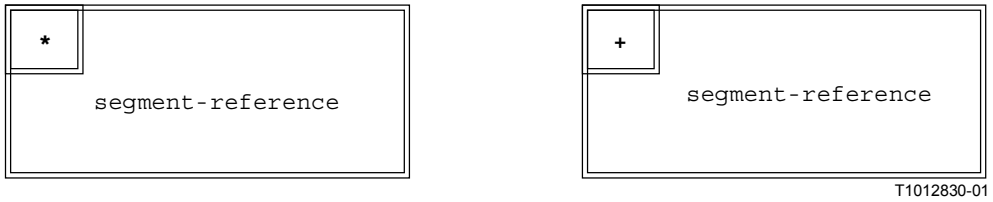


Figure B.6/Z.140 – Répétition de nœuds de référence

Il est possible d'indiquer un nombre maximal de répétitions possibles d'un nœud de référence sous la forme d'un nombre entier entre parenthèses après le symbole '*' ou '+', dans le nœud de référence à double encadrement. La référence de segment indiquée dans la Figure B.7 peut apparaître de zéro à 5 fois.

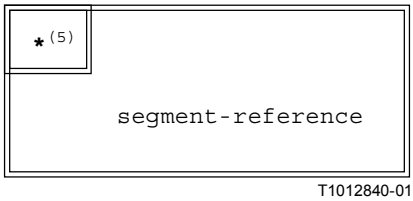
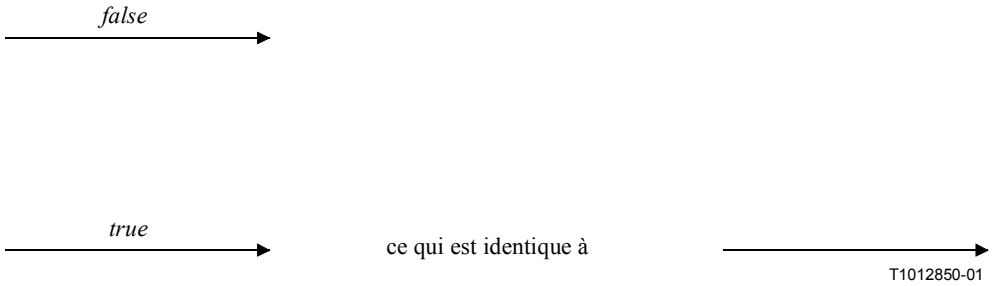


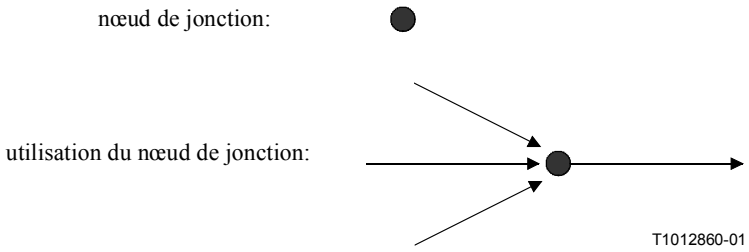
Figure B.7/Z.140 – Répétition limitée de nœuds de référence

B.3.1.3 Lignes de flux

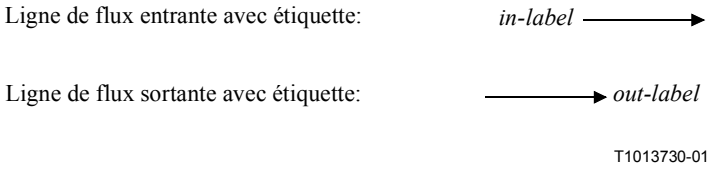
Les lignes de flux sont représentées par des flèches. Une ligne de flux possède une inscription *true* ou *false* qui indique un état dans lequel la ligne de flux est choisie au cours de l'interprétation du graphe orienté. Au titre de la notation abrégée, il est permis d'omettre l'inscription "true". Des exemples de lignes de flux sont donnés ci-dessous.



Afin d'assurer la jonction de plusieurs lignes de flux en une seule au niveau graphique, un nœud de jonction spécial est introduit. Le nœud de jonction et un exemple de son utilisation sont représentés ci-dessous:



Il est malaisé de dessiner de longues lignes de flux dans les grands diagrammes, comme cela est par exemple nécessaire pour modéliser les structures TTCN-3 **`goto`** et **`label`**. A cette fin, l'on peut utiliser des lignes de flux entrantes et sortantes, comme illustré ci-dessous.



Une ligne de flux sortante avec étiquette est connectée à une ligne de flux entrante avec étiquette, si les étiquettes sont identiques. Les étiquettes des lignes de flux entrantes doivent être uniques. S'il y a plusieurs lignes de flux sortantes avec la même étiquette, cette configuration est considérée comme étant une jonction de lignes à la ligne de flux entrante portant la même étiquette.

B.3.1.4 Segments de graphe orienté

Les segments de graphe orienté sont des sous-graphes orientés. Ils sont référencés dans les nœuds de référence, dont ils définissent la signification. Ils peuvent inclure d'autres nœuds de référence.

Comme indiqué dans la Figure B.8, les segments de graphe orienté ont des interfaces précises qui se composent de lignes de flux entrantes et sortantes. Il n'y a qu'une seule ligne de flux entrante sans étiquette et qu'une seule ou aucune ligne de flux sortante sans étiquette. Il peut également y avoir plusieurs lignes de flux entrantes et sortantes avec étiquette. Les lignes de flux entrantes et sortantes avec étiquette sont nécessaires pour décrire la signification des instructions **`goto`** de la notation TTCN-3.

Les segments de graphe orienté sont inscrits dans un cadre et le nom du segment de graphe orienté doit suivre le mot clé **segment** dans le coin supérieur gauche du cadre. Les lignes de flux décrivant l'interface avec le segment de flux orienté doit traverser le cadre de ce segment.

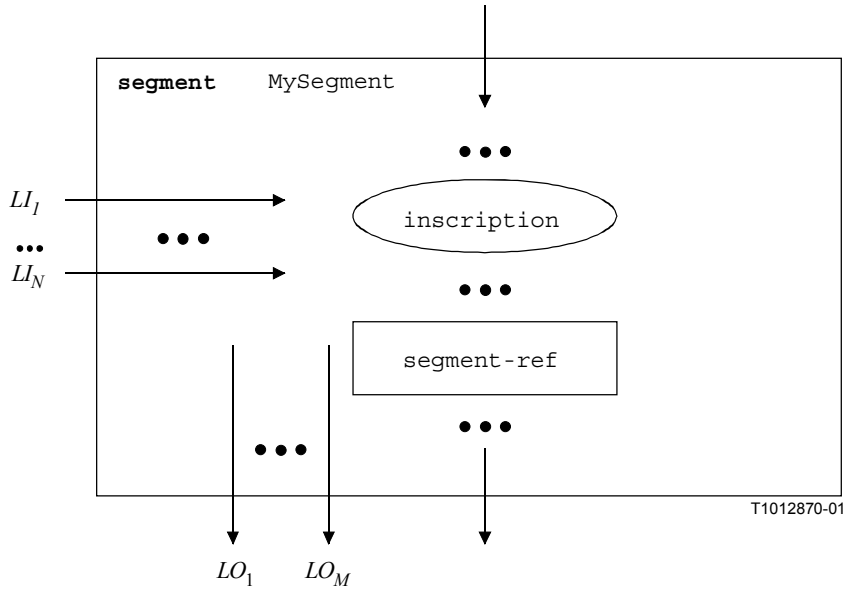
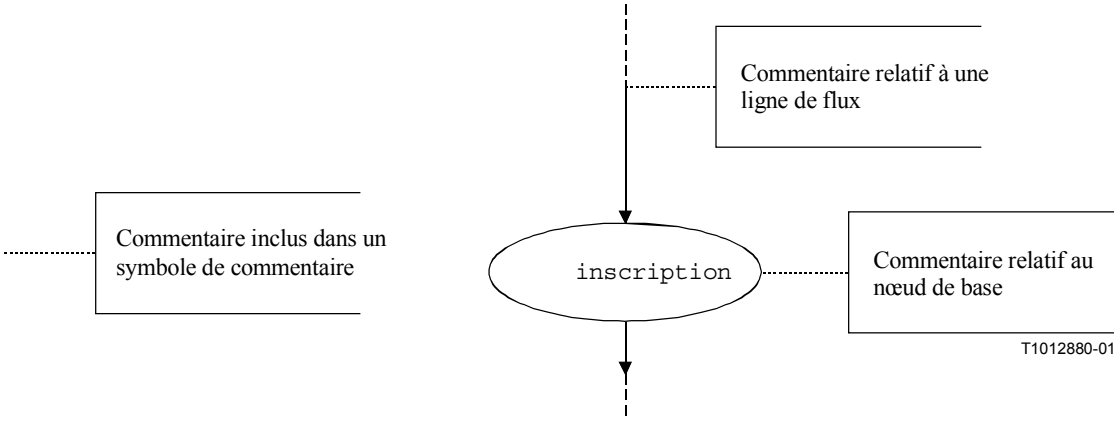


Figure B.8/Z.140 – Description schématique d'un segment de graphe orienté

B.3.1.5 Commentaires

Afin d'améliorer la lisibilité et la cohérence, un symbole de commentaire spécial peut être utilisé afin d'associer des commentaires à des nœuds de graphe orienté et à des lignes de flux. Ce symbole de commentaire et son utilisation sont indiqués dans la Figure B.9.



a) Symbole de commentaire **b) Utilisation des symboles de commentaire**

Figure B.9/Z.140 – Graphe orienté représentant les commentaires

B.3.1.6 Traitement des descriptions de graphe orienté

La procédure d'évaluation de la sémantique opérationnelle traverse les graphes orientés qui ne se composent que de nœuds de base, c'est-à-dire que tous les nœuds de référence sont étendus par les définitions des segments de graphe orienté correspondantes. La fonction *NEXT* est nécessaire pour prendre en charge cette traversée. La fonction *NEXT* est définie comme suit:

<actualNodeRef>.NEXT(<bool>) = <successorNodeRef>

où:

<*actualNodeRef*> est la référence d'un nœud de graphe orienté de base;

<*successorNodeRef*> est la référence d'un nœud successeur référencée par <*actualNodeRef*>;

<*bool*> est un booléen indiquant si une valeur *true* ou *false* est renvoyée pour le successeur (voir § B.3.1.3).

B.3.2 Représentation par graphe orienté du comportement TTCN-3

La sémantique opérationnelle part du principe que les descriptions de comportement TTCN-3 sont fournies sous la forme d'un ensemble de graphes orientés, c'est-à-dire qu'un graphe orienté distinct doit être construit pour chaque description de comportement TTCN-3.

La sémantique opérationnelle interprète les sortes suivantes de définitions TTCN-3 comme des descriptions de comportement:

a) commande de module;

b) définitions de test élémentaire;

c) définitions de fonction;

d) définitions de type de composant.

La partie commande du module spécifie la campagne de test, c'est-à-dire l'ordre d'exécution (qui peut être répétitif) des tests élémentaires proprement dits. Les définitions de test élémentaire définissent le comportement du composant MTC. Les définitions de fonction décrivent le comportement qui doit être exécuté par la commande de module ou par les composants de test. Les définitions de type de composant sont censées être des descriptions de comportement car elles spécifient la création, la déclaration et l'initialisation d'accès, de constantes, de variables et de temporisateurs au cours de la création d'une instance d'un type de composant.

B.3.2.1 La procédure de construction d'un graphe orienté

Les graphes orientés qui sont présentés dans les Figures B.10 et B.11 ainsi que les segments de graphe orienté qui sont présentés dans le § B.3.6 ne sont que des modèles. Ils comportent des *paramètres fictifs* contenant les informations qui doivent être fournies afin de produire concrètement un graphe orienté ou un segment de graphe orienté. Les paramètres fictifs sont mis entre parenthèses angulaires: '<' et '>'.

La construction d'une représentation par graphe orienté d'un module TTCN-3 s'effectue en trois étapes, comme suit:

1) un segment de graphe orienté concret est construit pour chaque instruction TTCN-3 contenue dans une commande modulaire, des tests élémentaires, des fonctions et des définitions de type de composant;

2) un graphe orienté concret est construit (avec nœuds de référence) pour chaque test élémentaire, chaque fonction et chaque définition de type de composant;

3) dans une procédure par étapes, tous les nœuds de référence contenus dans les graphes orientés concrets sont remplacés par les définitions de segment de graphe orienté correspondantes jusqu'à ce que tous les graphes orientés ne comportent qu'un seul nœud de début, des nœuds de fin et des nœuds de base des graphes orientés.

NOTE 1 – Les nœuds de base des graphes orientés décrivent des unités d'exécution indivisibles. La sémantique opérationnelle du comportement TTCN-3 est fondée sur l'interprétation des nœuds de base des graphes orientés. Le § B.4 présente les méthodes d'exécution que pour ces nœuds de base des graphes orientés.

Le remplacement d'un nœud de référence par la définition du segment de graphe orienté correspondante peut conduire à des parties non connectées dans un graphe orienté, c'est-à-dire à des parties qui ne peuvent pas être atteintes à partir du nœud de début par traversée du graphe orienté le long des lignes de flux. La sémantique opérationnelle ne tiendra pas compte des parties non connectées d'un graphe orienté.

NOTE 2 – Une partie non connectée d'un graphe orienté résulte de la procédure de remplacement mécanique. Pour la construction d'une représentation optimale d'un graphe orienté, les différentes combinaisons d'instructions TTCN-3 doivent également être prises en considération. L'objectif de la présente annexe est cependant d'offrir une sémantique correcte et complète, plutôt qu'une représentation optimale des graphes orientés.

B.3.2.2 Représentation d'une commande de module par graphe orienté

Schématiquement, la structure syntaxique d'un module TTCN-3 est la suivante:

```
module <identifier> (<parameter>) <module-definitions-part> control
                                                          <statement-block>
```

Pour la représentation du comportement d'un graphe orienté, seules les informations suivantes sont applicables:

```
module <identifier> <statement-block>
```

Ces informations sont comparables à une définition de fonction. La représentation par graphe orienté d'une commande de module est donc similaire à la représentation par graphe orienté d'une fonction (voir § B.3.2.4). La sémantique accédera au graphe orienté représentant la commande de module en utilisant le nom de celui-ci.

NOTE – La signification de la partie définitions de module est en dehors du domaine d'application de la présente sémantique opérationnelle. Les paramètres de module sont définis comme des constantes globales au moment de l'exécution. Les références à des paramètres de module doivent être remplacées par leurs valeurs concrètes au niveau syntaxique (voir § B.2.3).

B.3.2.3 Représentation de test élémentaire par graphe orienté

Schématiquement, la structure syntaxique d'une définition de test élémentaire TTCN-3 est la suivante:

```
testcase <identifier> (<parameter>) <testcase-interface> <statement-block>
```

Le terme `<testcase-interface>` ci-dessus se rapporte aux articles de la définition de test élémentaire **`runs on`** (obligatoire) et **`system`** (facultatif). La description par graphe orienté d'un test élémentaire décrit le comportement du composant MTC. Les informations fournies par l'interface de test élémentaire ne sont pas applicables au composant MTC. Elles seront utilisées par l'instruction **`execute`** mais n'ont pas besoin d'être représentées dans la représentation par graphe orienté d'un test élémentaire. Pour la représentation par graphe orienté, les seules informations suivantes sont donc nécessaires:

```
testcase <identifier> (<parameter>) <statement-block>
```

Ces informations sont comparables à une définition de fonction. La représentation par graphe orienté d'un test élémentaire est donc similaire à la représentation par graphe orienté d'une fonction (voir § B.3.2.4). La sémantique accédera au graphe orienté représentant les tests élémentaires en utilisant le nom de ceux-ci.

B.3.2.4 Représentation de fonctions par graphe orienté

Schématiquement, la structure syntaxique d'une fonction TTCN-3 est la suivante:

```
function <identifier> (<parameter>) [<function-interface>] <statement-block>
```

Le terme facultatif `<function-interface>` ci-dessus se rapporte aux articles de la définition de fonction **`runs on`** et **`return`**. Les informations fournies par l'interface de fonction ne sont pas applicables à la description du comportement. Elles seront utilisées pour les vérifications de sémantique statique mais n'ont pas besoin d'être représentées dans la représentation par graphe orienté. Pour la représentation par graphe orienté, les seules informations suivantes sont donc nécessaires:

```
function <identifier> (<parameter>) <statement-block>
```

La sémantique accédera aux graphes orientés représentant les fonctions en utilisant les noms de celles-ci.

Le schéma de la représentation par graphe orienté d'une fonction est reproduit dans la Figure B.10. Le nom du graphe orienté `<identifier>` se rapporte au nom de la fonction représentée (ou de la commande de module ou du test élémentaire). Les nœuds du graphe orienté ont des commentaires associés qui décrivent la signification des différents nœuds.

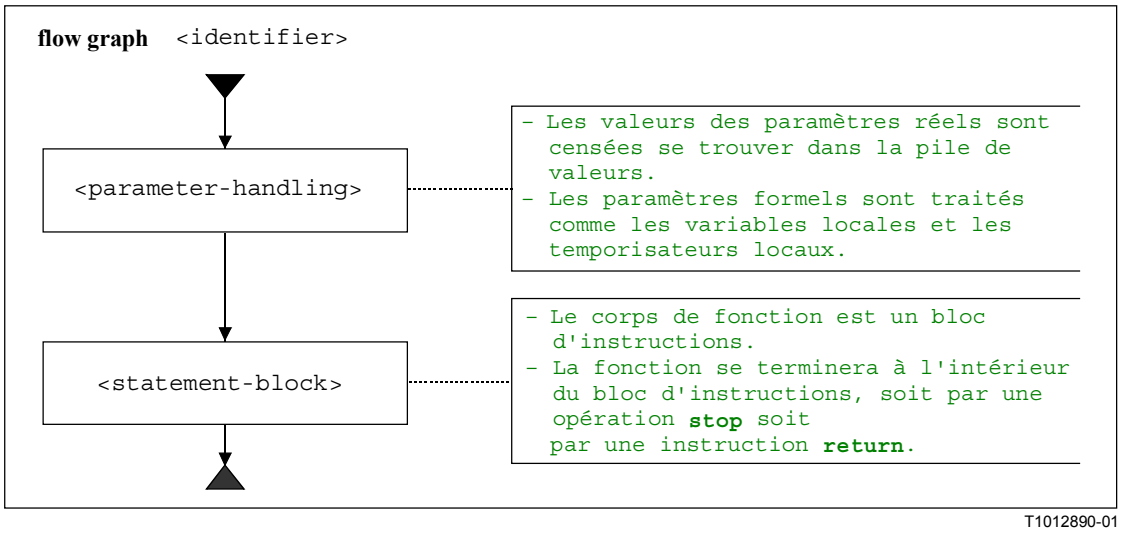


Figure B.10/Z.140 – Représentation de fonctions par graphe orienté

B.3.2.5 Représentation de définitions de type de composant par graphe orienté

Schématiquement, la structure syntaxique d'une définition de type de composant TTCN-3 est la suivante:

```
type component <identifier> <port-constant-variable-timer-declarations>
```

La sémantique accédera aux graphes orientés représentant les types en utilisant les noms de ceux-ci.

Le schéma de la représentation par graphe orienté d'une définition de type de composant est reproduit dans la Figure B.11. Le nom du graphe orienté `<identifier>` se rapporte au nom du type de composant représenté.

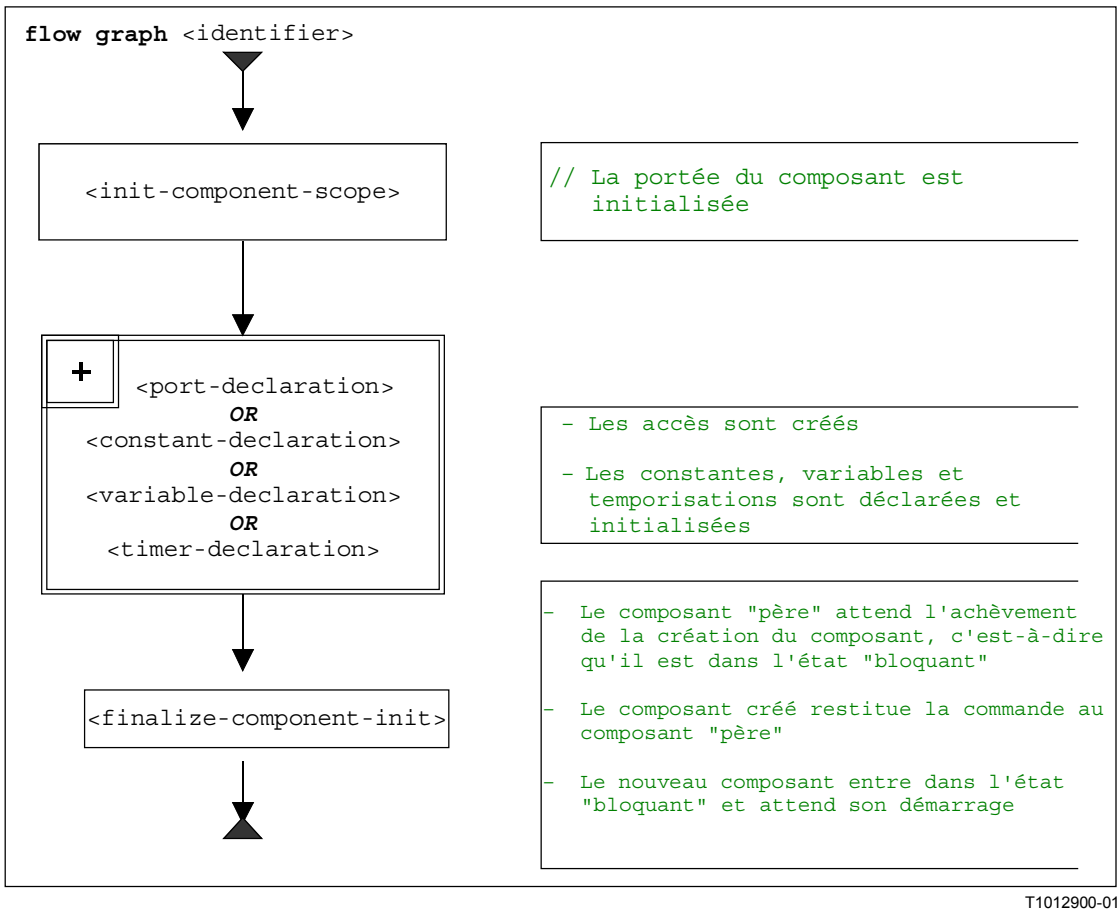


Figure B.11/Z.140 – Représentation par graphe orienté des définitions de type de composant

B.3.2.6 Extraction d'un nœud de début à partir d'un graphe orienté

La fonction suivante est requise pour l'extraction de la référence du nœud de début d'un graphe orienté:

La fonction *GET-FLOW-GRAPH*: *GET-FLOW-GRAPH* (<*flow-graph-identifier*>)

Cette fonction renvoie une référence au nœud de début d'un graphe orienté avec le nom <*flow-graph-identifier*>, qui se rapporte au nom du module pour la commande, aux noms des tests élémentaires, aux noms de fonction et aux définitions de type de composant.

B.3.3 Définitions des états des modules TTCN-3

Au cours de l'interprétation des graphes orientés représentant le comportement en notation TTCN-3, l'on manipule des *états de module*. Un état de module est un état structuré qui se compose de plusieurs sous-états décrivant les états des composants de test et des accès. Les états de module, les états de composant et les états d'accès sont présentés dans ce paragraphe. L'on définit par ailleurs les fonctions permettant d'extraire des informations à partir d'états et de manipuler ceux-ci.

B.3.3.1 Etat de module

Comme indiqué sur la Figure B.12, un état de module se compose d'une *liste d'états d'entité,* d'une *liste d'états d'accès*, d'une référence à un composant *MTC* et d'un verdict de type *TC-VERDICT*. La *liste d'états d'entité* décrit l'état de la commande de module et, durant l'exécution d'un test élémentaire les états des composantes de test instanciées. La *liste d'états d'accès*, la référence *MTC* et le verdict *TC-VERDICT* ne sont applicables qu'au cours de l'exécution d'un test élémentaire. La liste des états d'accès décrit ces états. La partie *MTC* fournit une référence au composant MTC. La partie *TC-VERDICT* mémorise le verdict de test global proprement dit d'un test élémentaire et la partie DONE est un compteur du nombre de mises à jour de la partie *TC-VERDICT*.

NOTE 1 – Le nombre de mises à jour de TC-VERDICT est identique au nombre de composants de test qui se sont terminés.

Le comportement de la partie commande d'un module (M-CONTROL dans la Figure B.12) est traité comme un composant de test normal et son état est le premier élément dans la *liste des états d'entité* d'un état de module.

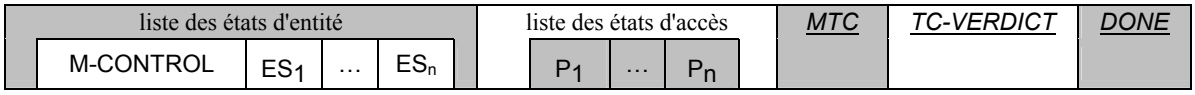


Figure B.12/Z.140 – Structure d'un état de module

NOTE 2 – Les états d'accès peuvent être considérés comme faisant partie des états d'entité. Les accès sont cependant rendus visibles par d'autres composants au moyen des instructions **`connect`** et **`map`**. Ils sont donc traités au niveau sommital d'un état de module.

B.3.3.1.1 Accès aux états de module

Les références *MTC*, *SYSTEM*, *TC-VERDICT* et *DONE* font partie d'un état de module et sont traitées comme des variables globales, c'est-à-dire que les mots clés *MTC* et *TC-VERDICT* peuvent être utilisés pour extraire et pour changer les valeurs de l'état correspondant du module.

NOTE 1 – Il n'existe qu'un seul état de module au cours de l'interprétation d'un module TTCN-3. Donc les mots clés *MTC* et *TC-VERDICT* peuvent être pris en considération en tant qu'identificateurs uniques pour la procédure d'évaluation.

Pour le traitement de la *liste des états d'entité* et de la *liste des états d'accès*, les opérations de liste *append*, *delete*, *first* et *length* peuvent être utilisées.

NOTE 2 – Les opérations de liste *append*, *delete*, *first* et *length* ont la signification suivante:

- <*list*>.*append*(<*item*>) ajoute <*item*> en tant que dernier élément dans la liste <*list*>;
- <*list*>.*delete*(<*item*>) supprime <*item*> de la liste <*list*>;
- <*list*>.*first*() renvoie le premier élément de <*list*>;
- <*list*>.*length*() renvoie la longueur de <*list*>;
- <*list*>.*next*(<*item*>) renvoie l'élément qui suit <item> dans la liste, ou la valeur **`NULL`** si <*item*> est le dernier élément dans la liste.

B.3.3.2 Etats d'entité

Les états d'entité servent à décrire les états réels des parties commande de module et des composants de test. La structure d'un état d'entité est représentée dans la Figure B.13.

<*identifier*>	***STATUS***	*CONTROL-STACK*	**état de données**	**état de temporisateur**	*VALUE-STACK*	*E-VERDICT*

Figure B.13/Z.140 – Structure d'un état d'entité

La partie <*identifier*> est un identificateur unique d'entité, c'est-à-dire d'une commande de module ou d'un composant de test, contenu dans le système de test. De tels identificateurs uniques sont créés implicitement pour la commande de module, pour le composant **mtc** et pour le système de test lorsqu'un module commence à s'exécuter ou lorsqu'un test élémentaire est exécuté au moyen de l'instruction **execute**. L'identificateur sert à désigner et à adresser des entités dans le système de test, par exemple en cas d'opérations **send** avec des clauses **to** ou d'opérations **receive** avec des clauses **from**.

La partie *STATUS* indique si la commande de module ou le composant de test est dans l'état **ACTIVE** ou **BLOCKED**. La commande de module est bloquée pendant l'exécution d'un test élémentaire. Des composants de test peuvent être bloqués pendant la création d'autres composants de test, c'est-à-dire pendant l'exécution d'une opération **create**.

La partie *CONTROL-STACK* est une pile de références à des nœuds de graphe orienté. L'élément sommital d'une pile *CONTROL-STACK* est le prochain nœud de graphe orienté qui doit être interprété. La pile est appelée à modéliser correctement les appels de fonction.

La partie *état des données* est considérée comme étant une énumération des listes de corrélations de variable. La structure de cette énumération de listes reflète les unités de portée imbriquées qui correspondent à des appels de fonction imbriqués. Chaque liste de l'énumération des listes de corrélations de variable décrit les corrélations de variable contenues dans chaque unité de portée. L'entrée ou la sortie d'une unité de portée correspond à l'adjonction ou à la suppression d'une liste de corrélations de variable à partir de la partie *état des données*. Une description plus détaillée de la partie *état des données* d'un état d'entité peut être consultée dans le § B.3.3.2.2.

La partie *état de temporisateur* est considérée comme étant une énumération des listes d'états de temporisateur. La structure de cette énumération de listes reflète les unités de portée imbriquées qui correspondent à des appels de fonction imbriqués. Chaque liste de l'énumération des listes d'états de temporisateur décrit les corrélations de temporisateur (les temporisateurs connus et leur état) contenues dans chaque unité de portée. L'entrée ou la sortie d'une unité de portée correspond à l'adjonction ou à la suppression d'une liste d'états de temporisateur à partir de la partie *état de temporisateur*. Une description plus détaillée de la partie *état de temporisateur* d'un état d'entité peut être consultée dans le § B.3.3.2.3.

La partie *VALUE-STACK* est une pile de valeurs de tous les types possibles, qui permet une mémorisation intermédiaire de résultats finals ou intermédiaires d'opérations, de fonctions et d'instructions. Par exemple, le résultat de l'évaluation d'une expression ou le résultat de la fonction **mtc** sera inscrit dans la partie *VALUE-STACK*. En plus des valeurs de tous les types de données connues dans un module, l'on définit la valeur spéciale **MARK** comme faisant partie de l'alphabet de la pile. Lors de la sortie d'une unité de portée, la valeur **MARK** sert à nettoyer la partie *VALUE-STACK*.

La partie *E-VERDICT* mémorise le verdict local réel d'un composant de test. Cette partie est négligée si un état d'entité représente la commande de module.

B.3.3.2.1 Accès aux états d'entité

Les parties *STATUS* et *E-VERDICT* d'un état d'entité sont traitées comme des variables globales, c'est-à-dire que les valeurs de *STATUS* et *E-VERDICT* peuvent être extraites ou modifiées au moyen de la notation 'par point' <*identifier*>.*STATUS* et <*identifier*>.*E-VERDICT*. La partie <*identifier*> dans la notation 'par point' se rapporte à l'identificateur unique d'une entité.

Les parties *CONTROL-STACK* et *VALUE-STACK* d'un état d'entité peuvent être adressées au moyen de la notation 'par point' <*identifier*>.*CONTROL-STACK* et <*identifier*>.*VALUE-STACK*.

Les parties *CONTROL-STACK* et *VALUE-STACK* peuvent être consultées et manipulées au moyen des opérations de pile *push*, *pop*, *top*, *clear* et *clear-until*.

NOTE – Les opérations de pile *push*, *pop*, *top*, *clear* et *clear-until* ont la signification suivante:

- <*stack*>.*push*(<*item*>) fait passer <*item*> à <*stack*>;
- <*stack*>.*pop*() désempile l'élément sommital de la pile <*stack*>;
- <*stack*>.*top*() renvoie l'élément sommital de <*stack*> ou **NULL** si la pile <*stack*> est vide;
- <*stack*>.*clear*() supprime <*stack*>, c'est-à-dire désempile tous les éléments de <*stack*>;
- <*stack*>.*clear-until*(<*item*>) désempile les éléments de <*stack*> jusqu'à ce que <*item*> soit trouvé ou que <*stack*> soit vide.

Pour la création d'un nouvel état d'entité la fonction *NEW-ENTITY* est supposée disponible:

NEW-ENTITY (<entity-identifier>, <*flow-graph-node-reference*>)

crée un nouvel état d'entité et renvoie sa référence. Les composants du nouvel état d'entité ont les valeurs suivantes:

- <entity-identifier> est l'identificateur unique;
- *STATUS* est mis à **ACTIVE**;
- <flow-graph-node-reference> est le seul élément (sommital) dans *CONTROL-STACK*;
- *data state* et *timer state* sont des listes vides;
- *VALUE-STACK* est une pile vide;
- *E-VERDICT* est mis à **none**.

Au cours de la traversée d'un graphe orienté, la pile *CONTROL-STACK* change souvent sa valeur de la même façon: l'élément sommital est désempilé de cette pile et le nœud successeur du nœud désempilé est transféré dans la pile *CONTROL-STACK*. Cette série d'opérations de pile est encapsulée dans la fonction *NEXT-CONTROL* comme suit:

```
<identifier>.NEXT-CONTROL(boolean <bool>) {
    FlowGraphNodeType successorNode := <identifier>.CONTROL-
    STACK.NEXT(<bool>).top();
    <identifier>.CONTROL-STACK.pop();
    <identifier>.CONTROL-STACK.push(successorNode).
}
```

B.3.3.2.2 Association d'état de données et de variable

Comme indiqué dans la Figure B.14, un *état de données* est une énumération de listes de corrélations de variable. Chaque liste de corrélations de variable définit les corrélations de variable se trouvant dans une certaine unité de portée. L'adjonction d'une nouvelle liste de corrélations de variable correspond à l'entrée dans une nouvelle unité de portée, par exemple, lors de l'appel d'une fonction. La suppression d'une liste de corrélations de variable correspond à la sortie d'une unité de portée, par exemple, lorsqu'une fonction effectue une instruction **return**.

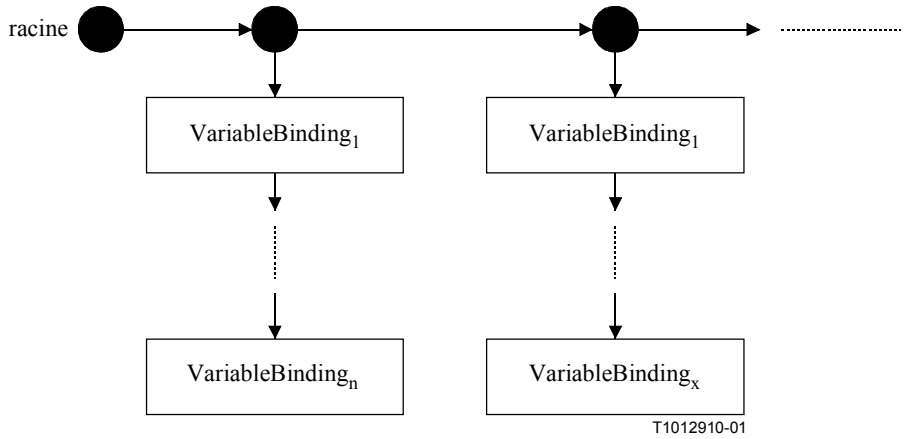


Figure B.14/Z.140 – Structure de la partie "état de données" d'un état d'entité

La structure d'une corrélation de variable est représentée dans la Figure B.15. Une variable possède un nom <*var-name*>, un *emplacement* et une *valeur*. Le nom <*var-name*> identifie une variable dans une unité de portée. L'*emplacement* est un identificateur unique de l'emplacement de mémorisation de la valeur de la variable. La partie *valeur* d'une corrélation de variable décrit la valeur réelle d'une variable.

NOTE – Les identificateurs uniques d'emplacement doivent être fournis automatiquement lors de la déclaration d'une variable.

<*var-name*>	**emplacement**	**valeur**

Figure B.15/Z.140 – Structure d'une corrélation de variable

La distinction entre nom de variable et emplacement de variable a été établie afin de modéliser correctement des appels de fonction et l'exécution de tests élémentaires avec paramétrisation en valeur et en référence, comme suit:

a) un paramètre transmis par valeur est traité comme la déclaration d'une nouvelle variable, c'est-à-dire qu'une nouvelle corrélation de variable est ajoutée à la liste des corrélations de variable de l'unité de portée de la fonction appelée ou du test élémentaire exécuté. La nouvelle corrélation de variable utilise le nom du paramètre formel en tant que nom de variable <*var-name*>, reçoit un nouvel emplacement et obtient la valeur qui est insérée dans la fonction ou dans le test élémentaire;

b) un paramètre transmis par référence conduit également à une nouvelle corrélation de variable dans l'unité de portée de la fonction appelée ou du test élémentaire exécuté. La nouvelle corrélation de variable utilise également le nom du paramètre formel en tant que nom de variable <*var-name*>, mais ne reçoit aucun nouvel emplacement ni aucune nouvelle valeur. La nouvelle corrélation de variable obtient une copie de *l'emplacement* et de la *valeur* de la variable qui sont transmis par référence.

Lors de la mise à jour d'une valeur de variable, par exemple, dans le cas d'une affectation à une variable, le nom de la variable est utilisé pour désigner un emplacement et toutes les corrélations de variable possédant le même emplacement sont mises à jour en même temps. Ainsi, lors de la sortie de l'unité de portée, la liste des variables appartenant à cette unité de portée peut être supprimée sans autre mise à jour. En raison de la procédure de mise à jour, les variables transmises par référence possèdent automatiquement la valeur correcte.

B.3.3.2.3 Association d'état de temporisateur et de temporisateur

Comme indiqué dans les Figures B.16 et B.17, un *état de temporisateur* et un *état de données* sont comparables dans un état d'entité. L'un et l'autre sont une énumération de listes de corrélations. Chaque liste de corrélations définit les corrélations valides dans une certaine unité de portée. L'adjonction d'une nouvelle liste correspond à l'entrée dans une nouvelle unité de portée et la suppression d'une liste de corrélations correspond à la sortie d'une unité de portée.

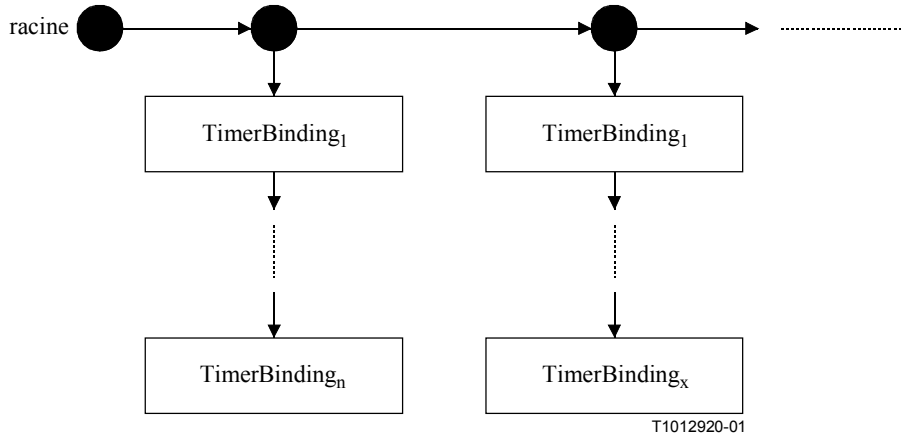


Figure B.16/Z.140 – Structure de la partie état de temporisateur d'un état d'entité

La structure d'une corrélation de temporisateur est représentée dans la Figure B.17. La signification des termes <*timer-name*> et *emplacement* est similaire à la signification des termes <*var-name*> et *emplacement* pour une corrélation de variable (Figure B.15).

<*timer-name*>	***emplacement***	*STATUS*	*DEF-DURATION*	*ACT-DURATION*	*TIME-LEFT*

Figure B.17/Z.140 – Structure d'une corrélation de temporisateur

La partie *STATUS* indique si un temporisateur est actif, inactif ou arrivé à expiration. Les valeurs *STATUS* sont **IDLE**, **RUNNING** et **TIMEOUT**. La partie *DEF-DURATION* décrit la durée par défaut d'un temporisateur. La partie *ACT-DURATION* mémorise la durée réelle avec laquelle un temporisateur a été armé. La partie *TIME-LEFT* décrit la durée réelle pendant laquelle un temporisateur actif doit rester armé avant d'arriver à expiration.

NOTE – La durée *DEF-DURATION* est indéfinie si un temporisateur est déclaré sans durée par défaut. Les durées *ACT-DURATION* et *TIME-LEFT* sont mises à 0,0 si un temporisateur est arrêté ou arrive à expiration. Si un temporisateur est armé sans durée, la valeur de l'élément *DEF-DURATION* est copiée dans l'élément *ACT-DURATION*. Une erreur dynamique se produit si un temporisateur est armé sans durée définie.

Un temporisateur ne peut être transmis à des fonctions que par référence, c'est-à-dire que le mécanisme est semblable au mécanisme pour les variables qui est décrit au § B.3.3.2.2. Cela signifie qu'une nouvelle corrélation de temporisateur (avec le nom de paramètre formel de type <*timer-name*>) est créée afin d'obtenir des copies des éléments *emplacement*, *STATUS*, *DEF-DURATION*, *ACT-DURATION* et *TIME-LEFT* à partir du temporisateur qui est transmis par référence. Lors de la mise à jour <*timer-name*> toutes les corrélations de temporisateur ayant le même *emplacement* sont mises à jour en même temps.

B.3.3.2.4 Accès aux états de temporisateur et de données

La *valeur* d'une variable peut être extraite au moyen de la notation à points <*identifier*>.<*var-name*> où <*identifier*> se rapporte à l'identificateur unique d'une entité. Pour modifier la valeur d'une variable, la fonction *VAR-SET* doit être utilisée:

<*identifier*>.*VAR-SET* (<*var-name*>, <*value*>)

met la valeur de variable <*var-name*> dans la portée réelle d'une entité avec l'identificateur unique <*identifier*>. En outre, la valeur de toutes les variables avec le même emplacement que la variable <*var-name*> sera également mise à <*value*>.

Les valeurs de *STATUS*, *DEF-DURATION*, *ACT-DURATION* et *TIME-LEFT* d'un temporisateur <*timer-name*> peuvent être extraites au moyen de la notation à points:

<*identifier*>.<*timer-name*>.*STATUS*;

<*identifier*>.<*timer-name*>.*DEF-DURATION*;

<*identifier*>.<*timer-name*>.*ACT-DURATION*;

<*identifier*>.<*timer-name*>.*TIME-LEFT*.

Pour modifier les valeurs de *STATUS*, *DEF-DURATION*, *ACT-DURATION* et *TIME-LEFT* d'un temporisateur <*timer-name*>, une opération générique TIMER-SET doit être utilisée, par exemple:

<*identifier*>.*TIMER-SET*(<*timer-name*>, *STATUS*, <*value*>)

met la valeur *STATUS* du temporisateur <*timer-name*> contenu dans la portée réelle d'une entité ayant l'identificateur unique <*identifier*> à la valeur <*value*>. En outre, la valeur de l'élément *STATUS* de tous les temporisateurs ayant le même emplacement que le temporisateur <*timer-name*> sera également mise à <*value*>. La fonction TIMER-SET peut aussi être utilisée pour changer les valeurs de *DEF-DURATION*, *ACT-DURATION* et *TIME-LEFT*.

Pour le traitement de variables, temporisateurs et unités de portée, les fonctions suivantes doivent être définies:

a) La fonction *INIT-VAR*: <*identifier*>.*INIT-VAR* (<*var-name*>, <*value*>)

crée une nouvelle corrélation de variable pour une variable <*var-name*> avec la valeur initiale <*value*> dans l'unité de portée réelle d'une entité avec l'identificateur unique <*identifier*>. L'utilisation du mot clé **NONE** en tant que <*value*> signifie qu'une variable avec une valeur initiale indéfinie est créée.

b) La fonction *INIT-TIMER*: <*identifier*>.INIT-TIMER (<*timer-name*>,<*duration*>)

crée une nouvelle corrélation de temporisateur pour un temporisateur <*timer-name*> avec la durée par défaut <*duration*> contenue dans la portée réelle d'une entité avec l'identificateur unique <*identifier*>. L'utilisation du mot clé **NONE** en tant que <*duration*> signifie qu'un temporisateur sans durée par défaut est créé.

c) La fonction *GET-VAR-LOC*: <*identifier*>.*GET-VAR-LOCATION* (<*var-name*>)

extrait l'emplacement de la variable <*var-name*> appartenant à une entité avec l'identificateur unique <*identifier*>

d) La fonction *GET-TIMER-LOC*: <*identifier*>.*GET-TIMER-LOCATION* (<*timer-name*>)

extrait l'emplacement du temporisateur <*timer-name*> appartenant à une entité avec l'identificateur unique <*identifier*>

e) La fonction *INIT-VAR-LOC*: <*identifier*>.*INIT-VAR-LOC* (<*var-name*>, <*location*>)

crée une nouvelle corrélation de variable pour une variable <*var-name*> avec l'emplacement <*location*> dans l'unité de portée réelle d'une entité avec l'identificateur unique <*identifier*>. La variable sera initialisée avec la valeur d'une autre variable avec l'emplacement <*location*>.

NOTE 1 – Les variables avec le même emplacement résultent d'une paramétrisation par référence. En raison du traitement des paramètres de référence comme décrit au § B.3.3.2.2, toutes les variables avec le même emplacement auront des valeurs identiques au cours de leur durée de vie.

f) La fonction *INIT-TIMER-LOC*: <*identifier*>.*INIT-TIMER-LOC* (<*timer-name*>, <*location*>)

crée une nouvelle corrélation de temporisateur pour un temporisateur <*timer-name*> avec l'emplacement <*location*> dans l'unité de portée réelle d'une entité avec l'identificateur unique <*identifier*>. Le temporisateur sera initialisé avec les valeurs de *STATUS*, *DEF-DURATION*, *ACT-DURATION* et *TIME-LEFT* d'un autre temporisateur avec l'emplacement <*location*>.

NOTE 2 – Les temporisateurs avec le même emplacement résultent d'une paramétrisation par référence. En raison du traitement des paramètres de référence de temporisateur comme décrit au § B.3.3.2.3, tous les temporisateurs avec le même emplacement auront des valeurs identiques pour *STATUS*, *DEF-DURATION*, *ACT-DURATION* et *TIME-LEFT* au cours de leur durée de vie.

g) La fonction *INIT-VAR-SCOPE*: <*identifier*>.*INIT-VAR-SCOPE* ()

initialise une nouvelle portée de variable dans l'état de données de l'entité avec l'identificateur unique <identifier>, c'est-à-dire qu'une liste vide est ajoutée en tant que première liste de l'énumération de listes de corrélations de variables.

h) La fonction *INIT-TIMER-SCOPE*: <*identifier*>.*INIT-TIMER-SCOPE* ()

initialise une nouvelle portée de temporisateur dans l'état de temporisateur de l'entité avec l'identificateur unique <*identifier*>, c'est-à-dire qu'une liste vide est ajoutée en tant que première liste de l'énumération des listes de corrélations de temporisateur.

i) La fonction *DEL-VAR-SCOPE*: <*identifier*>.*DEL-VAR-SCOPE* ()

supprime une portée de variable de l'état de données de l'entité avec l'identificateur unique <*identifier*>, c'est-à-dire que le première liste de l'énumération de listes de corrélation de variables est supprimée.

j) La fonction *DEL-TIMER-SCOPE*: <*identifier*>.*DEL-TIMER-SCOPE* ()

supprime une portée de temporisateur de l'état de temporisateur de l'entité avec l'identificateur unique <*identifier*>, c'est-à-dire que le première liste de l'énumération des listes de corrélations de temporisateur est supprimée.

B.3.3.3 Etats d'accès

Les états d'accès sont utilisés pour décrire les états réels des accès. La structure d'un état d'accès est décrite dans la Figure B.18. L'élément <*port-name*> se rapporte au nom d'accès qui est utilisé pour identifier l'accès par le composant de test <*owner*> qui possède cet accès. L'élément *STATUS* fournit l'état réel de l'accès. Un accès peut être dans l'état **`STARTED`** ou **`STOPPED`**.

NOTE – Un accès dans un système de test est identifié de manière unique par le composant de test détenteur <*owner*> et par le nom d'accès <*port-name*> contenu dans <*owner*>.

La partie *liste des connexions* d'un état d'accès conserve la trace des connexions entre les différents accès dans le système de test. Le mécanisme est expliqué au § B.3.3.2.1.

La partie *file de valeurs* d'un état d'accès contient les éléments de données qui sont reçus à cet accès mais pas encore consommés.

<*port-name*>	<*owner*>	*STATUS*	**liste des connexions**	**file de valeurs**

Figure B.18/Z.140 – Structure d'un état d'accès

B.3.3.3.1 Traitement des connexions entre accès

Une connexion entre deux composants de test est établie par raccordement de deux de leurs accès au moyen d'une opération **`connect`**. Un composant peut ainsi utiliser ultérieurement son nom d'accès local pour s'adresser à la file distante. Comme indiqué dans la Figure B.19, une *connexion*

est représentée dans les états des deux files connectées par une paire d'éléments <*remote-entity*> et <*remote-port-name*>. L'élément <*remote-entity*> est l'identificateur unique du composant de test qui possède l'accès distant. L'élément <*remote-port-name*> se rapporte au nom local utilisé par l'élément <*remote-entity*> pour s'adresser à la file. La notation TTCN-3 prend en charge les connexions d'accès en mode point à multipoint. Toutes les connexions d'un accès sont donc organisées dans une liste.

NOTE 1 – Les connexions établies par opérations **`map`** sont également traitées dans la liste des connexions. L'opération **`map`**(*PTC1*:*MyPort*, **`system`**.*PCO1*) conduit à une nouvelle connexion (**`system`**, *PCO1*) dans l'état de l'accès *MyPort* détenu par *PTC1*. Le côté distant auquel *PCO1* est connecté réside à l'intérieur du SUT. Son comportement est hors du domaine de la présente sémantique.

NOTE 2 – La sémantique opérationnelle traite le mot clé **`system`** en tant qu'adresse symbolique. Une connexion (**`system`**, <*port-name*>) contenue dans la liste des connexions d'un accès indique que l'accès est mappé sur l'accès <*port-name*> dans l'interface avec le système de test.

Figure B.19/Z.140 – Structure d'une connexion

B.3.3.3.2 Traitement des états d'accès

Le traitement des états d'accès est assuré par les méthodes suivantes.

a) La fonction *NEW-PORT*: *NEW-PORT(<owner>,<port-name>)*

crée un nouvel accès et renvoie sa référence. Le nouvel accès est détenu par <*owner*> et possède le nom <*port-name*> à l'accès identifié par le composant de test <*owner*> et le nom d'accès <*port-name*>. L'état du nouvel accès est armé et les deux éléments liste des connexions et file de valeurs sont vides.

b) La fonction *GET-PORT*: *GET-PORT(<owner>, <port-name>)*

renvoie une référence au port identifié par le composant de test <*owner*> qui possède l'accès et le nom d'accès <*port-name*>.

c) La fonction *GET-REMOTE-PORT*: *GET-REMOTE-PORT(<owner>, <port-name>, <remote-entity>)*

renvoie la référence à l'accès qui est détenu par composant de test <*remote-entity*> et connecté à un accès identifié par <*owner*> et <*port-name*>. L'adresse symbolique **SYSTEM** est renvoyée si l'accès distant est mappé sur un accès dans l'interface avec le système de test.

NOTE 1 – *GET-REMOTE-PORT* renvoie **`NULL`** s'il n'y a pas d'accès distant or si l'accès distant ne peut être identifié de manière univoque. La valeur spéciale **`NONE`** peut être utilisée en tant que valeur pour le paramètre <*remote-entity*> si l'entité distante n'est pas connue ou n'est pas requise, c'est-à-dire qu'il n'existe qu'une connexion point à point pour cet accès.

d) La fonction *STATUS* d'un accès est traitée comme une variable. Elle peut être adressée par qualification de la partie *STATUS* avec un appel *GET-PORT*:

GET-PORT(<*owner*>, <*port-name*>).*STATUS*

e) La fonction *ADD-CON*: *ADD-CON*(<*owner*>, <*port-name*>, <*remote-entity*>, <*remote-port-name*>)

ajoute une connexion (<*remote-entity*>, <*remote-port-name*>) à la liste des connexions de l'accès <*port-name*> détenu par <*owner*>.

f) La fonction *DEL-CON*: *DEL-CON*(<*owner*>, <*port-name*>, <*remote-entity*>, <*remote-port-name*>)

supprime la connexion (<*remote-entity*>, <*remote-port-name*>) de la liste des connexions de l'accès <*port-name*> détenu par <*owner*>.

La file d'attente des valeurs contenues dans un état d'accès peut être obtenue et manipulée au moyen des opérations de file connues sous les appellations suivantes: *enqueue*, *dequeue*, *first* et *clear*. L'utilisation d'une fonction *GET-PORT* ou *GET-REMOTE* fait référence à la file d'attente à laquelle l'accès est requis.

NOTE 2 – Les opérations de file d'attente *enqueue*, *dequeue*, *first* et *clear* ont la signification suivante:

- <*queue*>.*enqueue*(<*item*>) met <*item*> en tant que dernier élément dans <*queue*>;
- <*queue*>.*dequeue*() supprime le premier élément de <*queue*>;
- <*queue*>.*first*() renvoie le premier élément dans <*queue*> ou **`NULL`** si <*queue*> est vide;
- <*queue*>.*clear*() supprime tous les éléments de <*queue*>.

B.3.3.4 Fonctions générales pour le traitement des états de module

La sémantique opérationnelle part du principe que les fonctions suivantes existent pour le traitement des états de module.

NOTE – Au cours de l'interprétation d'un module TTCN-3, il n'existe qu'un seul état de module. L'on part du principe que les composants de l'état du module sont mémorisés dans des variables globales et non pas dans un objet complexe contenant des données. Les fonctions suivantes sont donc censées s'appliquer à des variables globales. Elle ne s'adressent pas à un objet spécifique d'état de module.

a) La fonction *DEL-ENTITY*: *DEL-ENTITY*(<*entity-identifier*>)

supprime une entité avec l'identificateur unique <*entity-identifier*>. La suppression implique:

- la suppression de l'état d'entité désigné par <*entity-identifier*>;
- la suppression de tous les accès détenus par <*entity-identifier*>;
- la suppression de toutes les connexions mettant en jeu <*entity-identifier*>.

b) La fonction *EXISTING*: *EXISTING*(<*entity-identifier*>)

renvoie *true* si un état d'entité existe pour l'entité identifiée par <*entity-identifier*>. Sinon il renvoie la valeur "*false*".

c) La fonction *UPDATE-REMOTE-REFERENCES*:

UPDATE-REMOTE-REFERENCES(<*source-entity*>, <*target-entity*>)

met à jour variables et temporisateurs ayant le même emplacement dans les deux entités. Les valeurs qui seront utilisées pour la mise à jour sont les valeurs de variables et temporisateurs détenues par <*source-entity*>.

B.3.4 Messages, appels de procédure, réponses et exceptions

L'échange d'informations entre composants de test et entre ceux-ci et le système SUT est associé à des *messages, appels de procédure, réponses à des appels de procédure* et à des *exceptions*. Aux fins des communications, ces éléments doivent être construits, codés et décodés. Le codage concret, c'est-à-dire le mappage des types de données TTCN-3 sur des bits et sur des octets, ainsi que le décodage, c'est-à-dire le mappage de bits et d'octets sur des types de données TTCN-3, sont hors du domaine d'application de la sémantique opérationnelle. Dans la présente Recommandation, les *messages, appels de procédure, réponses à des appels de procédure* et *exceptions* sont traités au niveau théorique.

B.3.4.1 Messages

Les messages sont associés à des communications asynchrones. Les valeurs de tous les types de données (prédéfinis et définis par l'utilisateur) peuvent être échangées entre les entités communicantes. Comme indiqué dans la Figure B.20, la sémantique opérationnelle traite un message comme un objet structuré composé d'une partie *expéditeur* et d'une partie *valeur*. La partie *expéditeur* désigne l'entité qui envoie le message tandis que la partie *valeur* définit la valeur du message.

Figure B.20/Z.140 – Structure d'un message

NOTE – La sémantique opérationnelle ne présente qu'un modèle pour les concepts de notation TTCN-3. La question de savoir si et comment les informations relatives à l'*expéditeur* doivent être ou ont été envoyées et/ou reçues dépend de l'implémentation du système de test. Par exemple, les informations d'*expéditeur* peuvent parfois être contenues dans la partie *valeur* d'un message et donc faire partie intégrante de sa structure.

B.3.4.2 Appels de procédure et réponses

Les appels de procédure et les réponses correspondantes sont associés à des appels synchrones. Ils sont définis comme les valeurs d'un enregistrement avec des composants représentant les paramètres. La sémantique opérationnelle traite également les appels de procédure et les réponses correspondantes comme des valeurs contenues dans des types structurés. La structure d'un appel de procédure et la structure d'une réponse à un tel appel sont présentées dans les Figures B.21 et B.22.

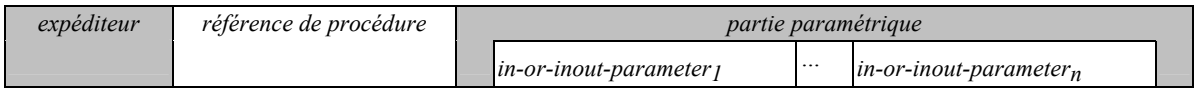


Figure B.21/Z.140 – Structure d'un appel de procédure

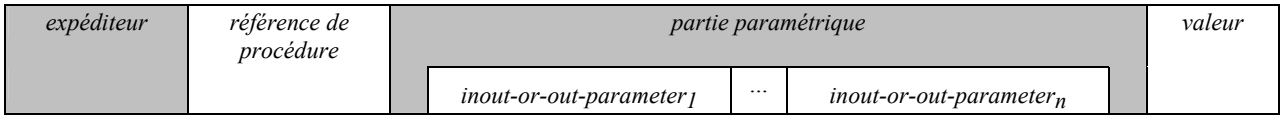


Figure B.22/Z.140 – Structure d'une réponse à un appel de procédure

La partie *expéditeur* et la partie *référence de procédure* ont la même signification dans les deux figures. La partie *expéditeur* se rapporte à l'entité expéditrice d'un appel ou d'une réponse à un appel de procédure. La partie *référence de procédure* se rapporte à la procédure à laquelle l'appel et la réponse appartiennent. La partie *partie paramétrique* de l'appel de procédure dans la Figure B.21 se rapporte aux paramètres **`in`** et **`inout`**. La partie *partie paramétrique* de la réponse dans la Figure B.22 se rapporte aux paramètres **`in`** et **`inout`** de la procédure à laquelle l'appel et la procédure appartiennent. Par ailleurs, la réponse contient une partie *valeur* pour les valeurs de retour contenues dans la réponse à un appel de procédure.

NOTE 1 – Comme indiqué dans la note précédente, la sémantique opérationnelle ne présente qu'un modèle pour les concepts de notation TTCN-3. La question de savoir si et comment les informations décrites dans les Figures B.21 et B.22 doivent être ou ont été envoyées et/ou reçues dépend de l'implémentation du système de test.

NOTE 2 – Pour un appel de procédure, les paramètres de type **`out`** ne sont pas applicables et sont omis de la Figure B.21. Pour une réponse à un appel de procédure, les paramètres de type **`in`** ne sont pas applicables et sont omis de la Figure B.22.

B.3.4.3 Exceptions

Les exceptions sont également associées à des communications synchrones. La structure d'une exception est décrite dans la Figure B.23. Elle se compose de trois parties. La partie *expéditeur* désigne l'entité expéditrice de l'exception. La partie *référence de procédure* se rapporte à la procédure à laquelle l'exception appartient. La partie *valeur* donne la valeur de l'exception. Le type de valeur d'une exception est défini dans la signature de la procédure à laquelle la partie *référence de procédure* fait référence. En général, il peut s'agir d'un quelconque type de données TTCN-3 prédéfini ou défini par l'utilisateur.

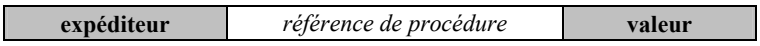


Figure B.23/Z.140 – Structure d'une exception

B.3.4.4 Construction de messages, d'appels de procédure, de réponses et d'exceptions

Les opérations d'expédition d'un message, d'un appel de procédure, d'une réponse à un appel de procédure ou d'une exception sont les suivantes: **`send`**, **`call`**, **`reply`** et **`raise`**. Toutes ces opérations d'envoi sont construites de la même façon:

```
<port-name>.<sending-operation>(<send-specification>) [to <receiver>]
```

Les termes <*port-name*> et <*sending-operation*> définissent l'accès et l'opération utilisés pour envoyer un élément. En cas de connexions point à multipoint, une entité <*receiver*> doit être spécifiée. L'élément à envoyer est construit au moyen de la spécification <*send-specification*>, laquelle peut utiliser des valeurs concrètes, des références de modèle, des valeurs de variable, des constantes, des expressions, des fonctions, etc., afin de construire et de coder l'élément à envoyer.

La sémantique opérationnelle part du principe qu'il existe une fonction générique *CONSTRUCT-ITEM*:

- *CONSTRUCT-ITEM*(<*sender*>, <*sending-operation*>, <*send-specification*>)

  renvoie un message, un appel de procédure, une réponse à un appel de procédure ou une exception selon les termes <*sending-operation*> et <*send-specification*>. Les informations <*sender*> sont également censées faire partie de l'élément à expédier (Figures B.20 à B.23).

B.3.4.5 Appariement de messages, d'appels de procédure, de réponses et d'exceptions

Les opérations de réception d'un message, d'un appel de procédure, d'une réponse à un appel de procédure ou d'une exception sont les suivantes: **`receive`**, **`getcall`**, **`getreply`** et **`catch`**. Toutes ces opérations de réception sont construites de la même façon:

```
<port-name>.<receiving-operation>(<matching-part>) [from <sender>]
                                                   [<assignment-part>]
```

Les parties <*port-name*> et <*receiving-operation*> définissent l'accès et l'opération utilisés pour la réception d'un élément. En cas de connexions point à multipoint, une clause **`from`** peut être utilisée afin de sélectionner une entité expéditrice spécifique <*sender*>. L'élément à recevoir doit remplir les conditions spécifiées dans la partie <*matching-part*>, c'est-à-dire qu'il doit correspondre. La partie <*matching-part*> peut utiliser des valeurs concrètes, des références de modèle, des valeurs de variable, des constantes, des expressions, des fonctions, etc., afin de spécifier les conditions d'appariement.

La sémantique opérationnelle part du principe qu'il existe une fonction générique *MATCH-ITEM*:

- *MATCH-ITEM*(<*item-to-check*>, <*matching-part*>, <*sender*>)

  renvoie *true* si <*item-to-check*> répond aux conditions de l'élément <*matching-part*> et si l'élément <*item-to-check*> a été envoyé par <*sender*>; sinon il renvoie la valeur "*false*".

B.3.4.6 Extraction d'informations à partir d'éléments reçus

Les informations obtenues par la réception de messages, d'appels de procédure, de réponses à des appels de procédure et d'exceptions peuvent être extraites de la partie <*assignment-part*> (voir § B.3.4.3) des fonctions réceptrices **`receive`**, **`getcall`**, **`getreply`** et **`catch`**. L'élément <*assignment-part*> décrit la façon dont les paramètres d'appel et de réponse de procédure, les valeurs de retour codées dans les réponses, les messages, les exceptions et l'identificateur de l'entité <*sender*> sont affectés à des variables.

La sémantique opérationnelle part du principe qu'il existe une fonction générique *RETRIEVE-INFO*:

- *RETRIEVE-INFO*(<*item-received*>, <*assignment-part*>, <*receiver*>)

  toutes les valeurs à extraire selon l'élément <*assignment-part*> sont extraites et affectées aux variables énumérées dans la partie affectation. Les affectations sont effectuées au moyen de l'opération VAR-SET, c'est-à-dire que les variables ayant le même emplacement sont mises à jour en même temps.

B.3.5 Fichiers de communication pour fonctions et tests élémentaires

Les fonctions et les tests élémentaires sont appelés (ou exécutés) au moyen de leur nom et d'une liste de paramètres réels. Ces derniers fournissent des références pour le paramètre de référence ainsi que des valeurs concrètes pour le paramètre de valeur, comme défini par les paramètres formels dans la définition de fonction ou de test élémentaire. La sémantique opérationnelle traite les appels de fonction et les appels de tests élémentaires au moyen d'*enregistrements d'appel* comme indiqué dans la Figure B.24. La valeur de l'élément *BEHAVIOUR-ID* est le nom d'une fonction ou d'un test élémentaire. Les paramètres de valeur fournissent des valeurs concrètes <$parId_1$> … <$parId_n$> pour les paramètres formels <$parId_1$> … <$parId_n$>. Les paramètres de référence fournissent des références relatives aux emplacements de variables et temporisateurs existants. Avant qu'une fonction ou un test élémentaire puisse être exécuté, un fichier de communication approprié doit être construit.

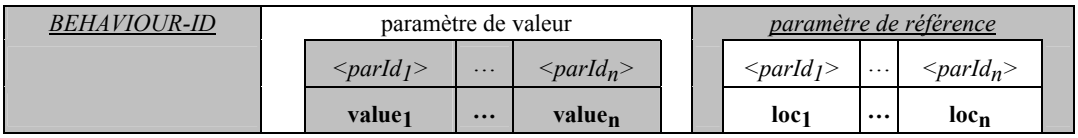


Figure B.24/Z.140 – Structure d'un fichier de communication

B.3.5.1 Traitement des fichiers de communication

Le nom de la fonction ou du test élémentaire ainsi que les valeurs de paramètre réel peuvent être extraits au moyen de la notation par points. Par exemple, <*myRecord*>.<$parId_n$> ou <*myRecord*>.*BEHAVIOUR-ID* où <*myRecord*> est un pointeur vers un fichier de communication.

Pour la construction d'un fichier de communication, la fonction *NEW-CALL-RECORD* est supposée disponible:

- *NEW-CALL-RECORD*(<*behaviour-name*>)

  crée un nouveau fichier de communication pour la fonction ou le test élémentaire nommé <*behaviour-name*> et renvoie un pointeur vers le nouveau fichier. Les champs paramétriques du nouveau fichier de communication ont des valeurs indéfinies.

- <*call-record*>.*INIT-CALL-RECORD()*

  crée des variables et des temporisateurs pour le traitement des paramètres de valeur et de référence dans la portée réelle d'une fonction ou d'un test élémentaire. Les variables pour le

traitement des paramètres de valeur sont initialisées avec les valeurs correspondantes qui sont fournies dans le fichier de communication. Les variables et les temporisateurs pour le traitement des paramètres de référence obtiennent l'emplacement fourni. Ils obtiennent par ailleurs une valeur de variable ou de temporisateur existant dans une autre unité de portée du composant dans lequel le fichier de communication a été créé.

B.3.6 La procédure d'évaluation pour un module de notation TTCN-3

B.3.6.1 Phases d'évaluation

La procédure d'évaluation relative à un module de notation TTCN-3 est structurée en quatre phases comme suit:

1) *phase d'initialisation;*
2) *phase de mise à jour*;
3) *phase de sélection*;
4) *phase d'exécution*.

Les phases 2), 3) et 4) sont répétées jusqu'à ce que la partie commande du module soit terminée. La procédure d'évaluation est décrite au moyen d'une association de texte informel, de pseudo-code et des fonctions introduites dans les paragraphes précédents.

B.3.6.1.1 Phase I: initialisation

La phase d'initialisation comporte les actions suivantes:

a) **Déclaration et initialisation de variables**

INIT-FLOW-GRAPHS(); // Initialisation du traitement par graphe orienté.
// L'action *INIT-FLOW-GRAPHS* est expliquée au § B.3.5.1.

– *Entity*:= **NULL**; // La variable *Entity* sera utilisée pour renvoyer vers un état
// d'entité. Un état d'entité représente soit une commande de
// module soit un composant de test.

– *AllEntities*:= **NULL**; // La variable *AllEntities* sera une liste d'états d'entité

– *AllPorts*:= **NULL**; // La variable *AllPorts* sera une liste d'états d'accès

– *MTC*:= **NULL**; // *MTC* renverra au composant MTC lorsqu'un test élémentaire
// est en cours

– *TC-VERDICT*:= **none**; // *TC-VERDICT* mémorisera le verdict réel de test élémentaire
// lorsqu'un test élémentaire est en cours

– *DONE*:= 0; // Au cours de l'exécution d'un test élémentaire,
// *DONE* décompte le nombre de composant de tests qui
// se sont terminés.

NOTE – Les variables globales *AllEntities*, *AllPorts*, *MTC*, *TC-VERDICT* et *DONE* forment l'état de module qui est manipulé au cours de l'interprétation d'un module TTCN-3.

b) **Création et initialisation de commande de module**

Entity:= *NEW-ENTITY* (*GET-UNIQUE-ID*(),*GET-FLOW-GRAPH* (<*moduleId*>));
// Un nouvel état d'entité est créé et initialisé
// avec le nœud de début du graphe orienté
// représentant le comportement de la
// commande du module avec le nom
// <*moduleId*>. La fonction *GET-UNIQUE-ID*
// sera expliquée au § B.3.5.1.

Entity.*INIT-VAR-SCOPE*(); // Nouvelle portée de variable

Entity.*INIT-TIMER-SCOPE*(); // Nouvelle portée de temporisateur

Entity.*VALUE-STACK*.*push*(**MARK**); // Une marque est insérée dans la pile de
// valeurs

AllEntities.*append*(*Entity*); // La nouvelle entité est mise dans l'état
// de module.

B.3.6.1.2 Phase II: mise à jour

La phase de mise à jour concerne toutes les actions qui sont à l'extérieur de la portée de la sémantique opérationnelle mais influence l'interprétation d'un module TTCN-3. La phase de mise à jour implique les actions suivantes:

a) **progression du temps**: tous les temporisateurs actifs sont mis à jour, c'est-à-dire que les valeurs *TIME-LEFT* des temporisateurs armés sont (le cas échéant) diminuées, et si en raison de la mise à jour un temporisateur arrive à expiration, les corrélations de temporisateur correspondantes sont mises à jour, c'est-à-dire que la variable *TIME-LEFT* est mise à 0,0 et que la variable *STATUS* est mise à `TIMEOUT`;

b) **comportement du SUT**: les messages, les appels de procédure distante, les réponses à des appels de procédure distante et les exceptions (le cas échéant) reçus en provenance du système SUT sont mis dans les files d'attente d'accès dans lesquelles les réceptions correspondantes doivent avoir lieu.

NOTE – Cette sémantique opérationnelle ne fait aucune hypothèse quant à la progression du temps et quant au comportement du SUT.

B.3.6.1.3 Phase III: sélection

La phase de sélection se compose des deux actions suivantes:

a) **sélection**: sélectionner une entité non bloquée, c'est-à-dire une entité qui a la valeur d'état *STATUS* **ACTIVE**;

b) **mise en instance**: mémoriser l'identificateur de l'entité sélectionnée dans la variable globale *Entity*.

B.3.6.1.4 Phase IV: exécution

La phase d'exécution se compose des deux actions suivantes:

a) **étape d'exécution de l'entité sélectionnée**: exécuter le nœud sommital du graphe orienté contenu dans la pile *CONTROL-STACK* de l'*entité*;

b) **critère de contrôle de terminaison**: arrêter l'exécution si la commande de module s'est terminée, c'est-à-dire que la liste des états d'entité est vide, sinon continuer avec la Phase II.

NOTE – L'étape d'exécution de l'entité sélectionnée peut être vue comme un appel de procédure. La vérification du critère de terminaison est effectuée lorsque l'étape d'exécution se termine, c'est-à-dire renvoie la commande.

B.3.6.2 Fonctions globales

La procédure d'évaluation utilise les fonctions globales *INIT-FLOW-GRAPHS* et *GET-UNIQUE-ID*:

a) *INIT-FLOW-GRAPHS* est supposée être la fonction qui initialise le traitement par graphe orienté. Le traitement peut comprendre la création du graphe orienté et le traitement du pointeur vers le graphe orienté et vers ses nœuds.

b) *GET-UNIQUE-ID* est supposée être une fonction qui renvoie un identificateur unique chaque fois qu'elle est appelée. L'identificateur unique peut être implémenté sous la forme d'une variable de compteur qui est incrémentée et renvoyée chaque fois que la fonction *GET-UNIQUE-ID* est appelée.

Le pseudo-code utilise les paragraphes suivants pour décrire l'exécution de nœuds de graphe orienté utilisant les fonctions *CONTINUE-COMPONENT*, *RETURN*, *****DYNAMIC-ERROR*****:

c) *CONTINUE-COMPONENT* le composant de test réel continue son exécution avec le nœud se trouvant au sommet de la pile de commande, c'est-à-dire que la commande n'est pas rendue à la procédure d'évaluation de module décrite dans ce paragraphe.

d) *RETURN* renvoie la commande à la procédure d'évaluation de module décrite dans cette clause. Le *RETURN* est la dernière action de '*l'étape d'exécution de l'entité sélectionnée*' de la phase d'*exécution*.

e) *****DYNAMIC-ERROR***** se rapporte à l'occurrence d'une erreur dynamique. La procédure de traitement des erreurs elle-même est hors du domaine de la sémantique opérationnelle. Si une erreur dynamique se produit tous les comportements suivants du module sont censés être indéfinis.

NOTE – L'occurrence d'une erreur dynamique concerne le comportement de test. Une erreur dynamique spécifiée par la sémantique opérationnelle indique un problème dans l'utilisation de la notation TTCN-3, par exemple, un usage erroné ou des conditions critiques.

B.3.7 Définitions des segments de graphe orienté pour les créations syntaxiques TTCN-3

La sémantique opérationnelle représente le comportement de la notation TTCN-3 sous la forme de graphes orientés. L'algorithme de construction des graphes orientés représentant le comportement est décrit au § B.3.2.1. Il est fondé sur les modèles de graphe orienté et de segment de graphe orienté qui doivent être utilisés lors de la construction de graphes orientés concrets pour la partie commande d'un module, les tests élémentaires, les fonctions et les définitions de type de composant contenues dans un module TTCN-3. Les définitions des modèles pour les segments de graphe orienté seront décrites dans le présent paragraphe. Elles sont présentées en ordre alphabétique et non pas en ordre logique.

Les définitions de segment de graphe orienté sont représentées sous la forme de figures. Les nœuds de graphe orienté sont indiqués du côté gauche de ces figures et les commentaires associés aux nœuds et aux lignes de flux sont indiqués du côté droit. Des annotations descriptives sont présentées pour les nœuds de référence et des commentaires en pseudo-code sont associés aux nœuds de base. Le pseudo-code décrit comment un nœud de base est interprété, c'est-à-dire les changements de l'état du module. Il fait usage des fonctions définies dans les paragraphes précédents du § B.3 ainsi que des variables globales qui ont été déclarées et initialisées au cours de la procédure d'évaluation pour modules TTCN-3 (§ B.3.6). L'on pourra également trouver dans le présent paragraphe un aperçu général de toutes les fonctions et de tous les mots clés utilisés par le pseudo-code.

B.3.7.1 Instruction d'alternative

La représentation par graphe orienté de l'instruction **`alt`** dans la Figure B.25 établit une distinction entre instructions **`alt`** ayant une branche **`else`** et instructions **`alt`** n'ayant pas de branche **`else`**.

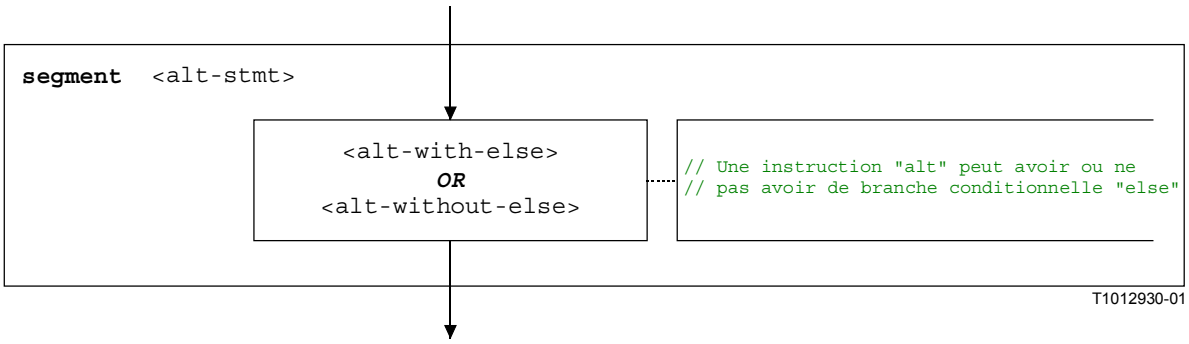


Figure B.25/Z.140 – Segment de graphe orienté <alt-stmt>

Les segments de graphe orienté <alt-with-else> et <alt-without-else> sont indiqués dans les Figures B.26 et B.27. La branche **else** est un bloc d'instructions qui ne nécessite pas d'autre explication. Cependant, les deux segments de graphe orienté sont très similaires à la différence près que la branche **else** fournit une sortie définie pour l'instruction **alt**, tandis qu'une instruction d'alternative sans branche **else** peut se reboucler.

Les deux segments de graphe orienté ont un nœud d'entrée et, en dehors d'une seule ligne de flux entrante, une ligne de flux additionnelle avec une étiquette <altId>. C'est une étiquette symbolique pour l'instruction **alt** qui identifie la cible d'instructions **goto alt** et également définit le rebouclage dans le segment de graphe orienté <alt-without-else>. Les deux segments de graphe orienté également ont un point de sortie défini au moyen de l'étiquette <altIdExit> et du nœud alt-exit.

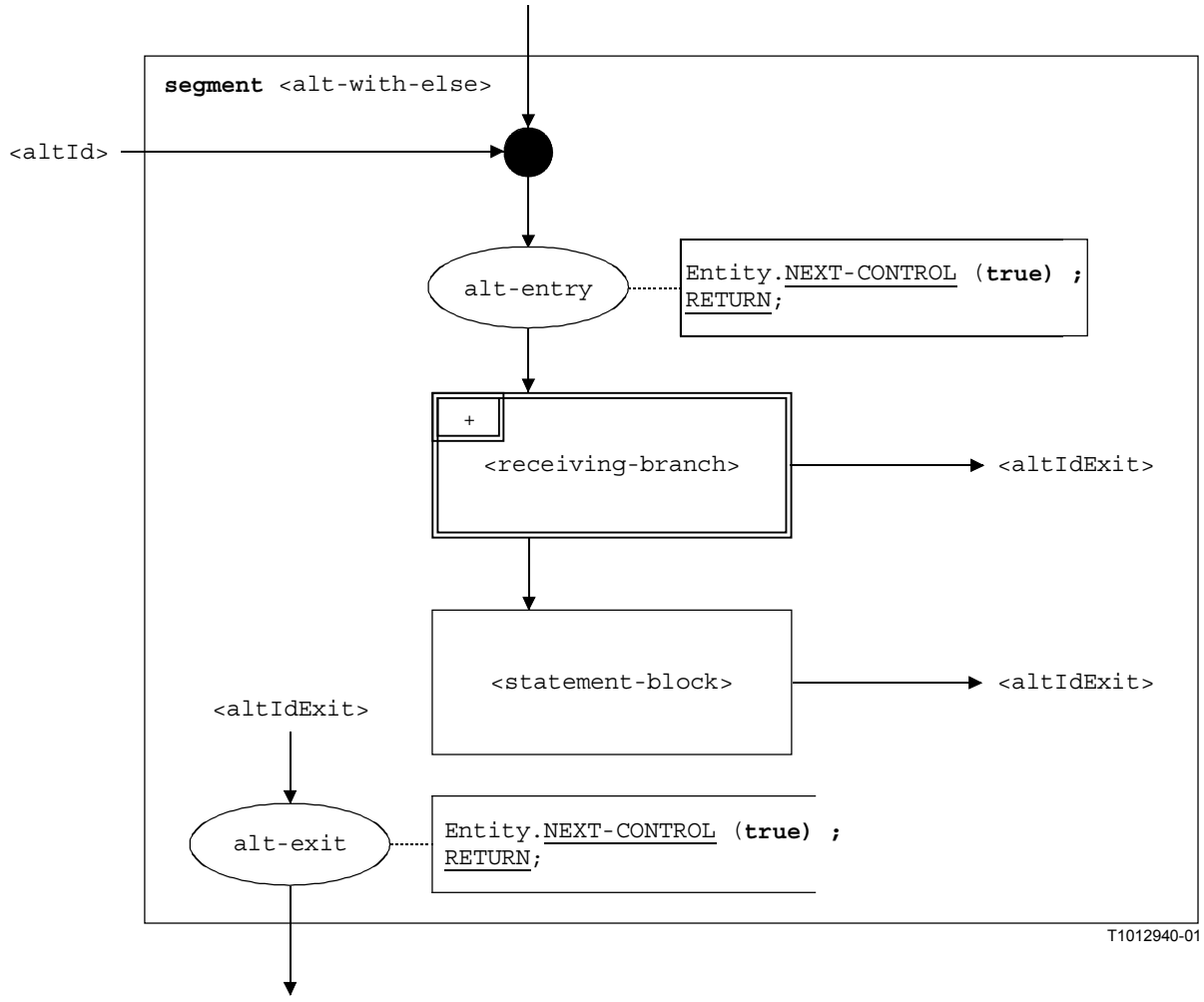


Figure B.26/Z.140 – Segment de graphe orienté <alt-with-else>

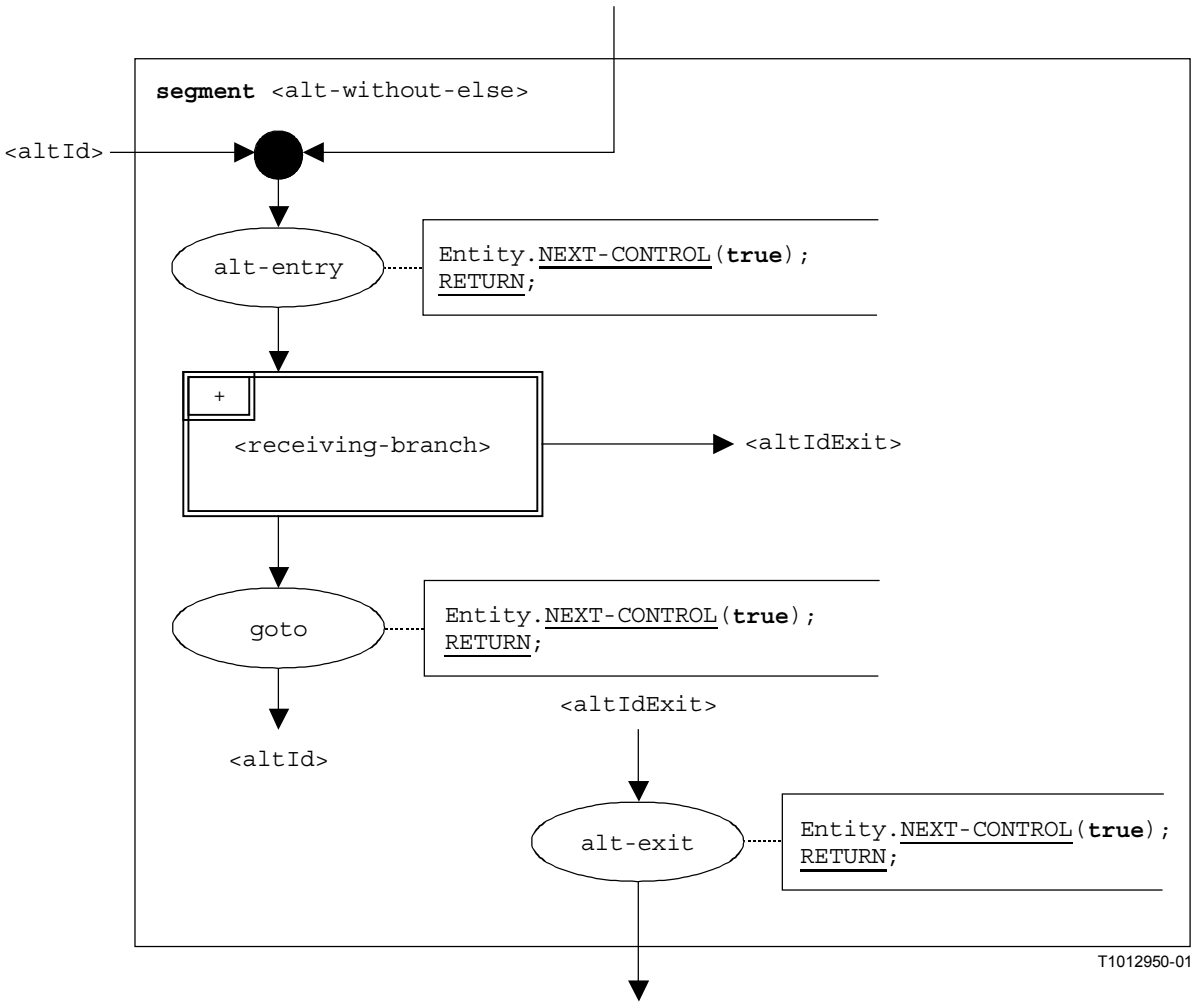


Figure B.27/Z.140 – Segment de graphe orienté <alt-without-else>

B.3.7.1.1 Segment de graphe orienté <receiving-branch>

L'exécution du segment de graphe orienté `<receiving-branch>` est décrite dans la Figure B.28.

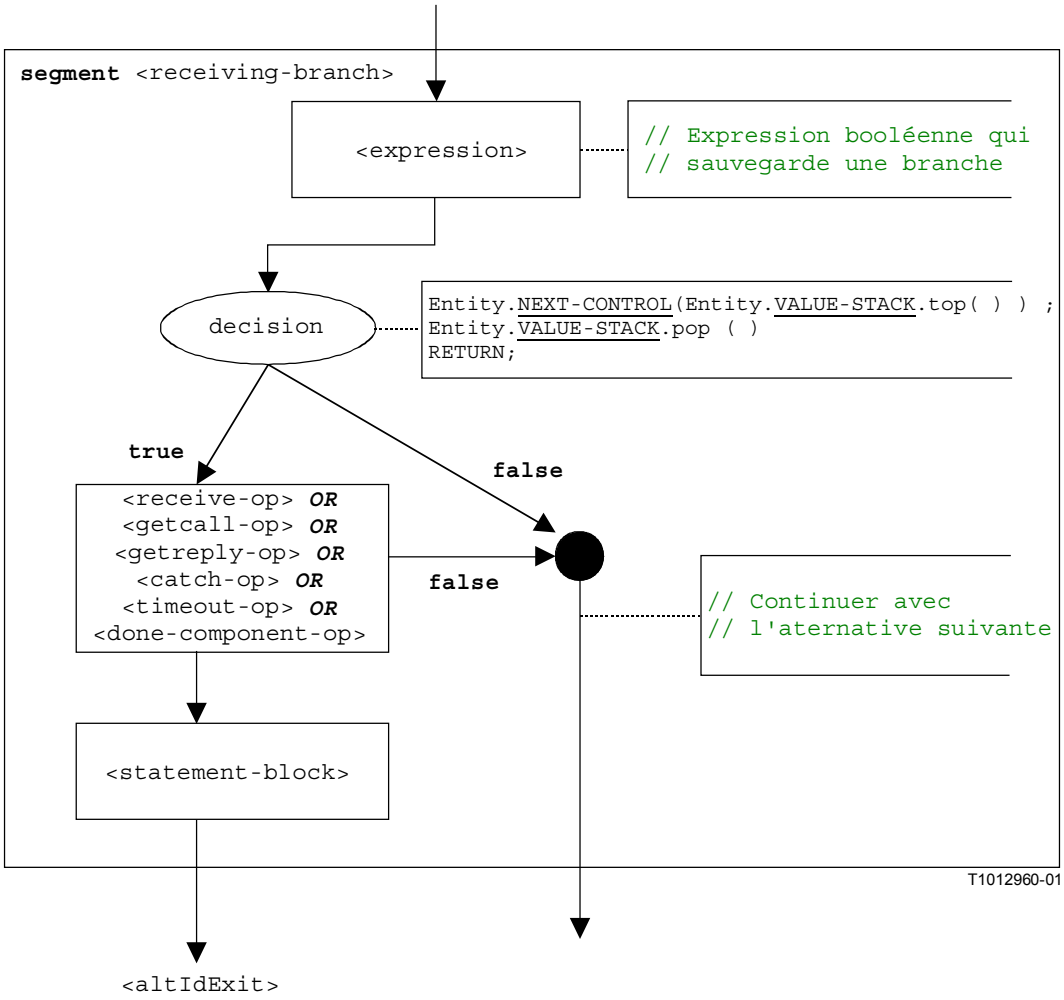


Figure B.28/Z.140 – Segment de graphe orienté <receiving-branch>

B.3.7.2 Instruction d'affectation

La structure syntaxique d'une instruction **`assignment`** est:

```
<varId> := <expression>
```

La valeur de l'expression `<expression>` est affectée à la variable `<varId>`. L'exécution d'une instruction d'affectation est définie par le segment de graphe orienté `<assignment-stmt>` dans la Figure B.29.

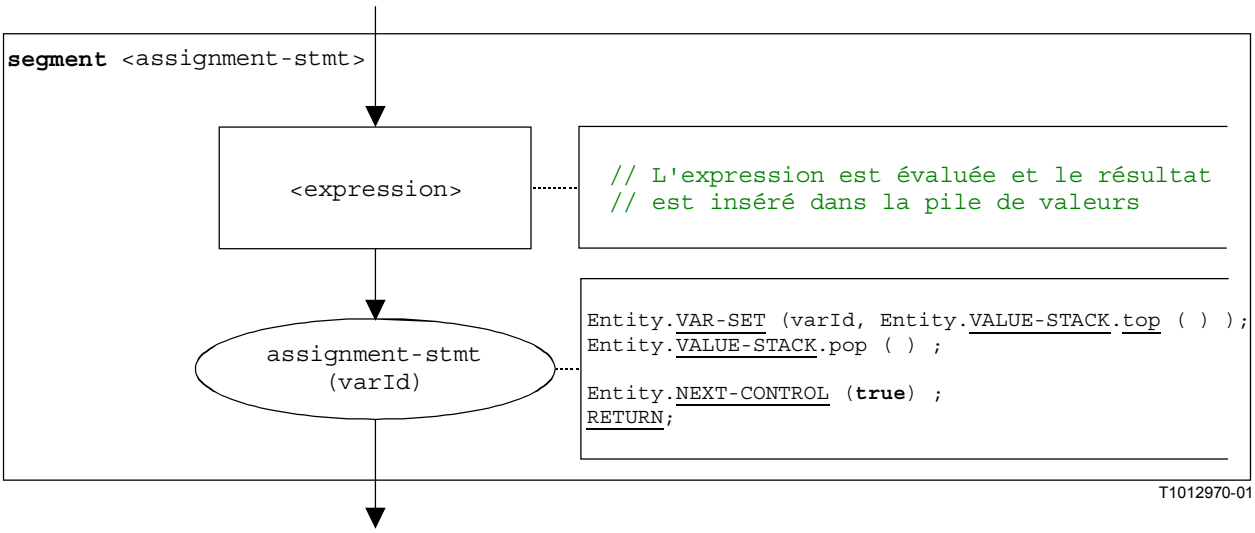


Figure B.29/Z.140 – Segment de graphe orienté <assignment-stmt>

B.3.7.3 Opération d'appel

La structure syntaxique de l'opération **call** est:

```
<portId>.call (<callSpec> [<blocking-info>]) [to <component_expression>]
                                                [<call-reception-part>]
```

L'expression facultative <blocking-info> se compose soit du mot clé **nonblocking** soit d'une durée pour une exception de temporisation. L'expression facultative <component_expression> dans la clause **to** se rapporte à l'entité réceptrice. Elle peut être fournie sous la forme d'une valeur de variable ou de la valeur de retour d'une fonction. L'expression facultative <call-reception-part> indique les réceptions en variante en cas d'opération d'appel bloquant.

La sémantique opérationnelle établit une distinction entre opérations d'**appel** *bloquant* et d'**appel** *non bloquant* (*blocking* et un *non-blocking* **call**). Un **call** est non bloquant s'il n'attend pas de réponses ou si le mot clé **nonblocking** est utilisé. Un **call** bloquant possède une partie <call-reception-part>.

Le segment de graphe orienté <call-op> dans la Figure B.30 définit l'exécution d'une opération **call**. Il reflète la distinction entre appels bloquants et non bloquants.

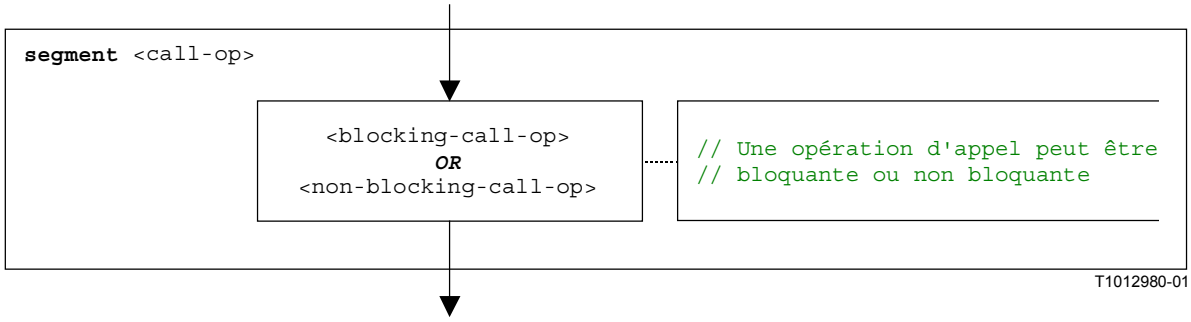


Figure B.30/Z.140 – Segment de graphe orienté <call-op>

Une entité réceptrice peut être spécifiée sous la forme d'une expression pour les opérations d'appel bloquantes et non bloquantes. Les possibilités sont indiquées dans les Figures B.31 et B.32.

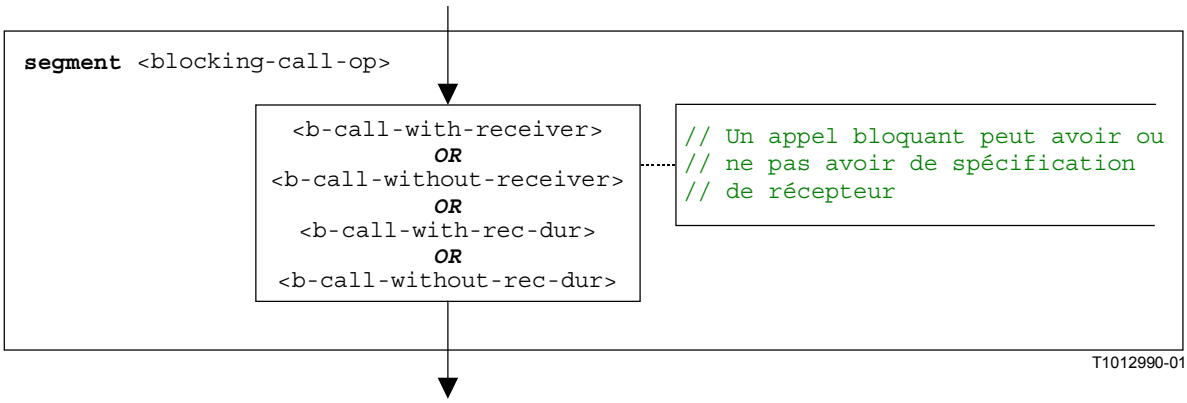


Figure B.31/Z.140 – Segment de graphe orienté <blocking-call-op>

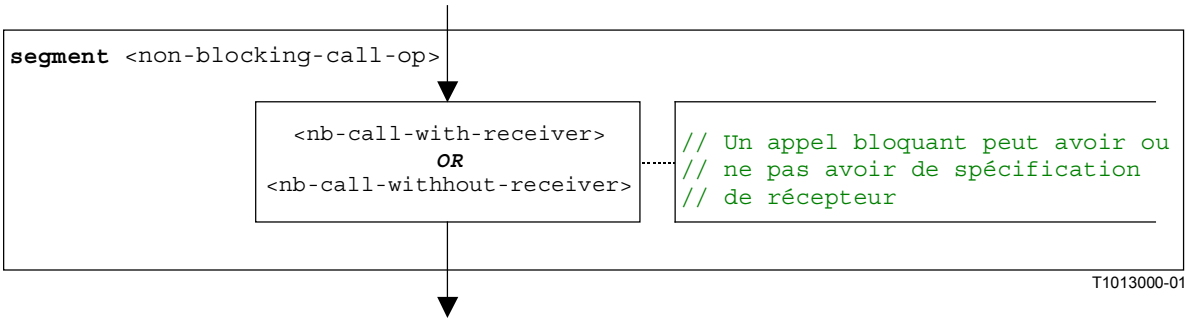


Figure B.32/Z.140 – Segment de graphe orienté <non-blocking-call-op>

B.3.7.3.1 Segment de graphe orienté <nb-call-with-receiver>

Le segment de graphe orienté `<nb-call-with-receiver>` de la Figure B.33 définit l'exécution d'une opération d'appel non bloquant **`call`** où le récepteur est spécifié sous la forme d'une expression.

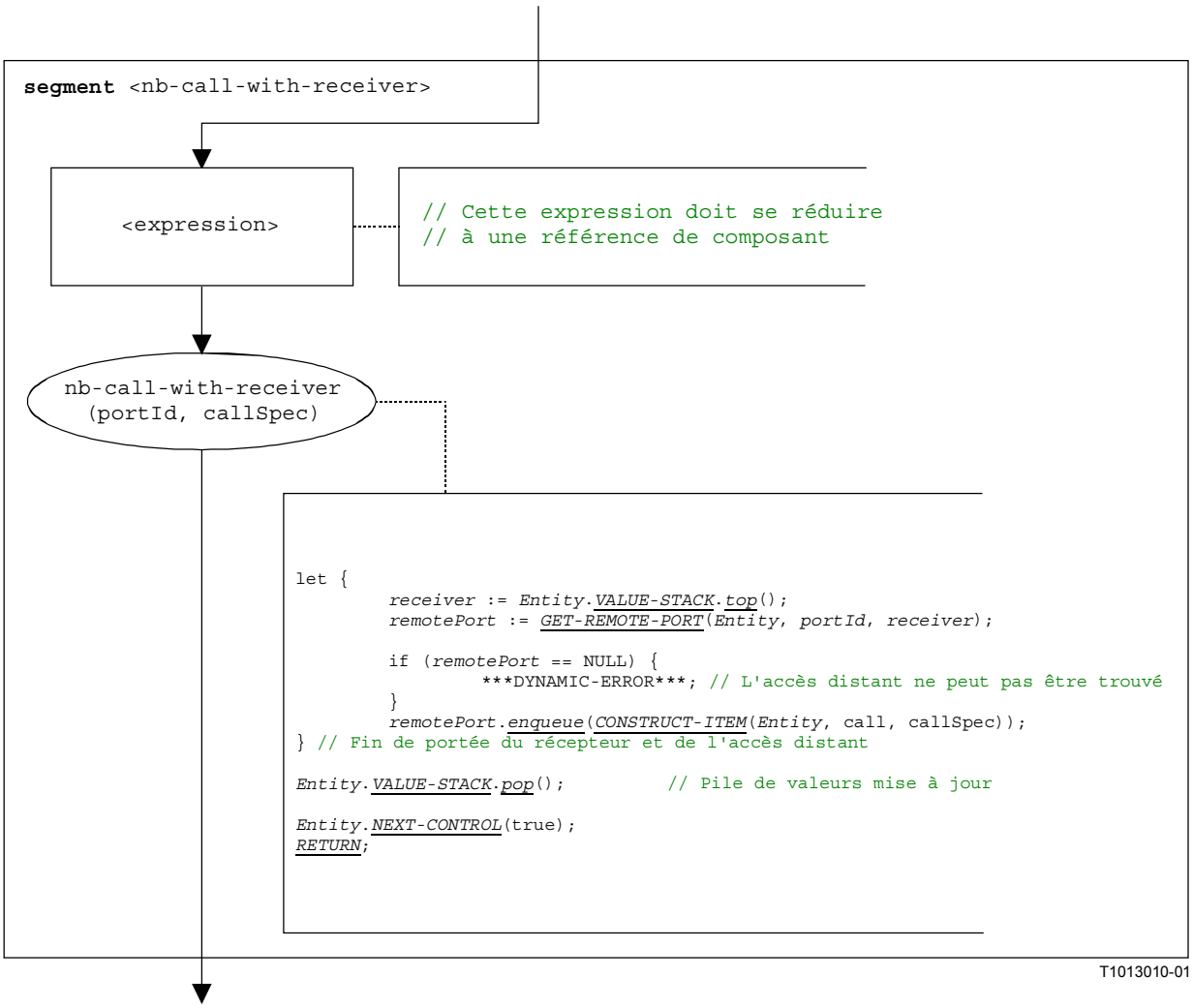


Figure B.33/Z.140 – Segment de graphe orienté <nb-call-with-receiver>

B.3.7.3.2 Segment de graphe orienté <nb-call-without-receiver>

Le segment de graphe orienté `<nb-call-without-receiver>` dans la Figure B.34 définit l'exécution d'une opération d'appel non bloquant **`call`** sans clause **`to`**.

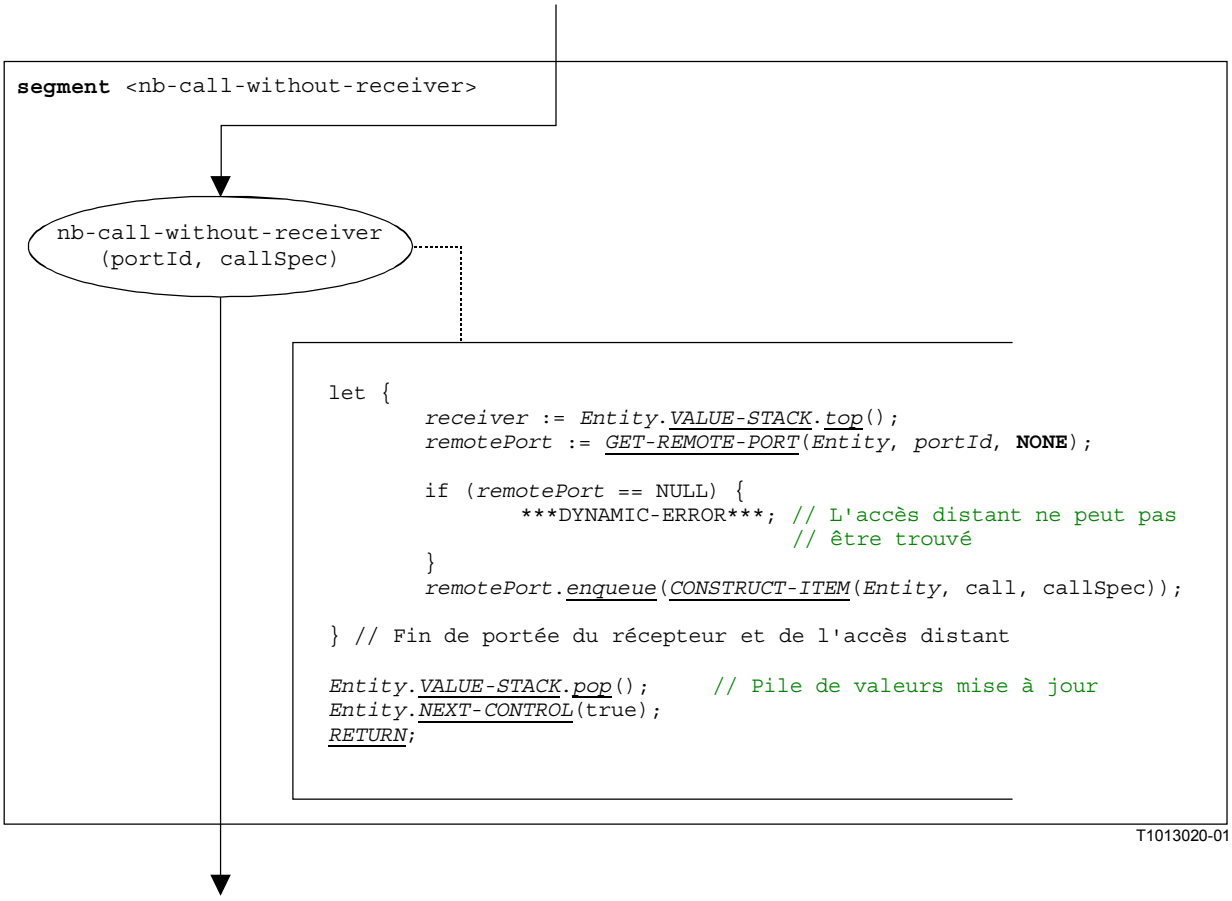


Figure B.34/Z.140 – Segment de graphe orienté <nb-call-without-receiver>

B.3.7.3.3 Segment de graphe orienté <b-call-with-receiver>

Les appels bloquants sont modélisés par un appel non bloquant suivi par une instruction d'alternative. Le segment de graphe orienté `<b-call-with-receiver>` décrit l'exécution d'un appel bloquant, sans durée en tant que mise en instance par temporisateur, mais avec une description de récepteur pour l'appel. Le segment de graphe orienté est décrit dans la Figure B.35.

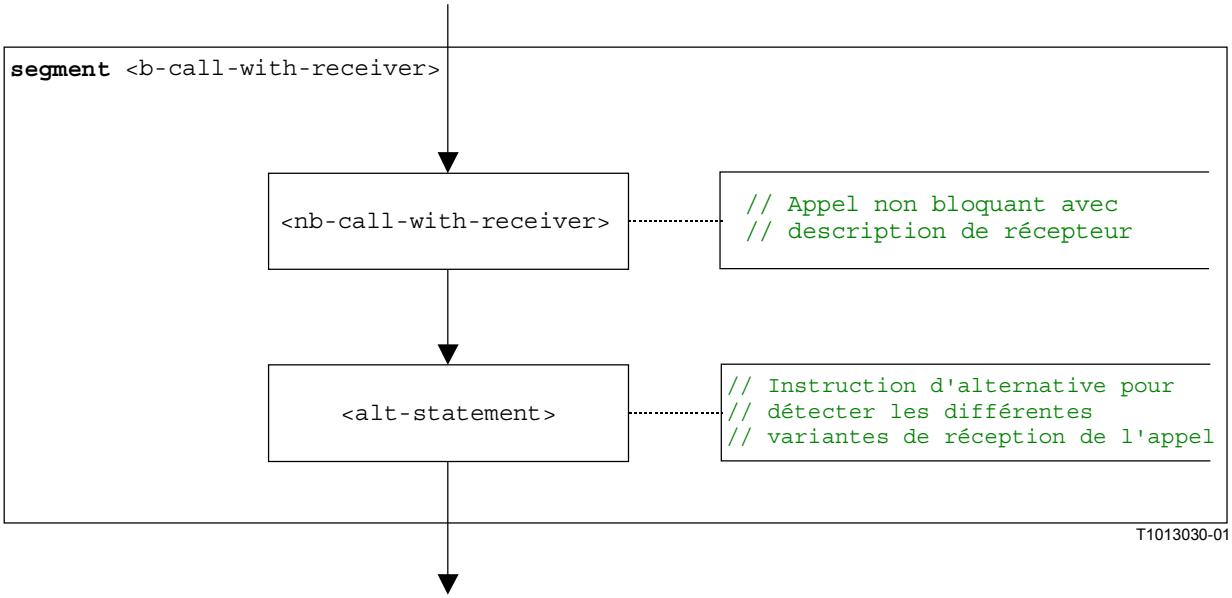


Figure B.35/Z.140 – Segment de graphe orienté <b-call-with-receiver>

B.3.7.3.4 Segment de graphe orienté <b-call-without-receiver>

Le segment de graphe orienté `<b-call-without-receiver>` décrit l'exécution d'un appel bloquant, sans durée en tant que mise en instance par temporisateur et sans description de récepteur pour l'appel. Le segment de graphe orienté est décrit dans la Figure B.36.

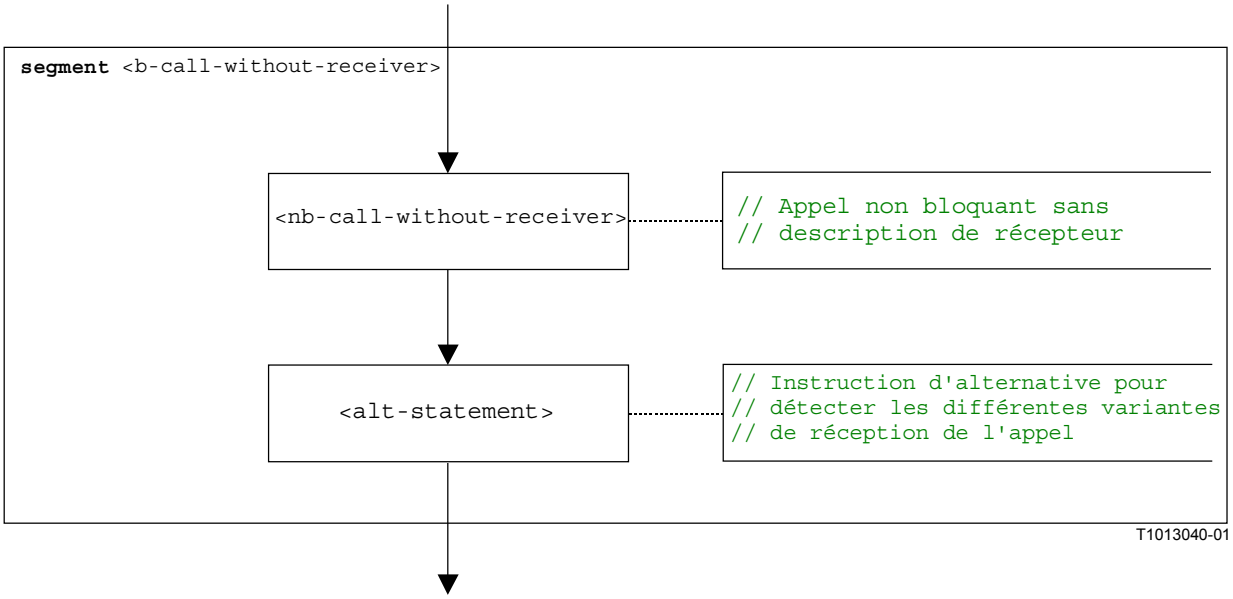


Figure B.36/Z.140 – Segment de graphe orienté <b-call-without-receiver>

B.3.7.3.5 Segment de graphe orienté <b-call-with-rec-dur>

Les appels bloquants mis en instance par temporisateurs sont modélisés par un appel non bloquant suivi par une instruction d'alternative. Pour la durée, un temporisateur système spécial SYS-TI est armé. La branche de temporisation de détection contenue dans l'instruction **`alt`** se rapporte au temporisateur système. Le segment de graphe orienté `<b-call-with-rec-dur>` décrit l'exécution d'un appel bloquant, avec une durée en tant que mise en instance par temporisateur et à description de récepteur pour l'appel. Le segment de graphe orienté est décrit dans la Figure B.37.

NOTE – Le traitement du temporisateur système n'est traité que de manière informelle. L'implémentation relève de l'équipement de test.

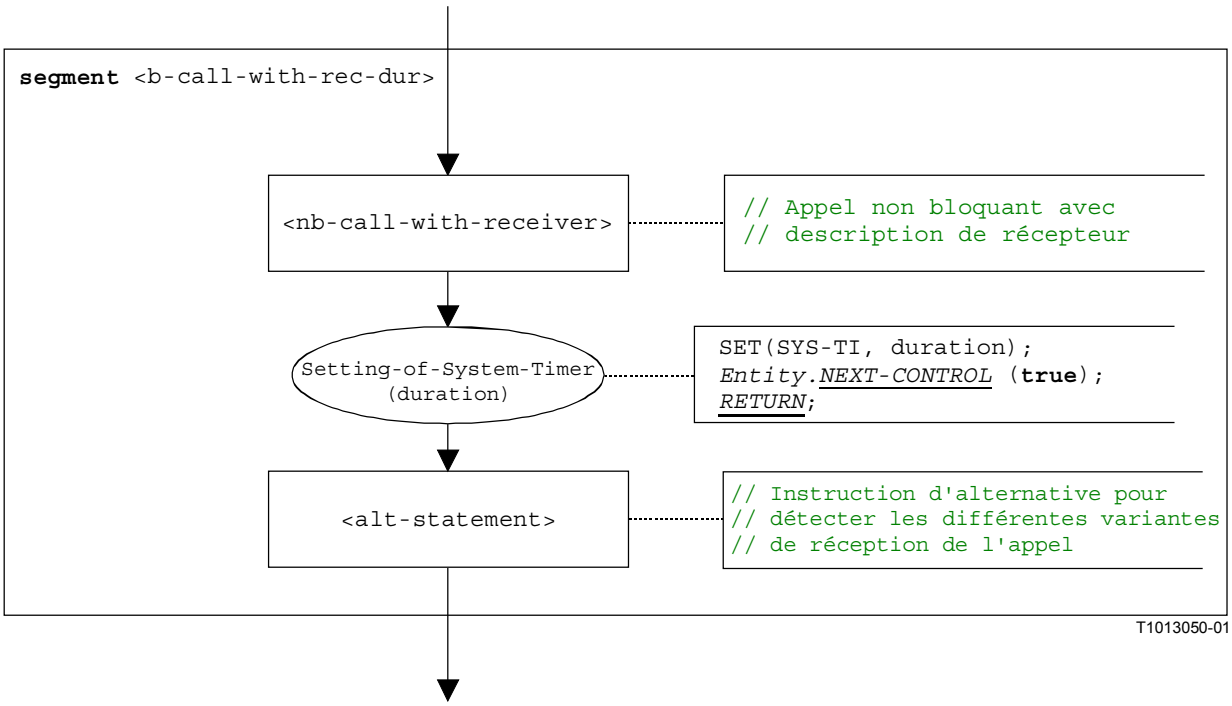


Figure B.37/Z.140 – Segment de graphe orienté <b-call-with-rec-dur>

B.3.7.3.6 Segment de graphe orienté <b-call-without-rec-dur>

Le segment de graphe orienté <b-call-without-rec-dur> décrit l'exécution d'un appel bloquant, avec une durée en tant que mise en instance par temporisateur et sans description de récepteur pour l'appel. Le segment de graphe orienté est décrit dans la Figure B.38.

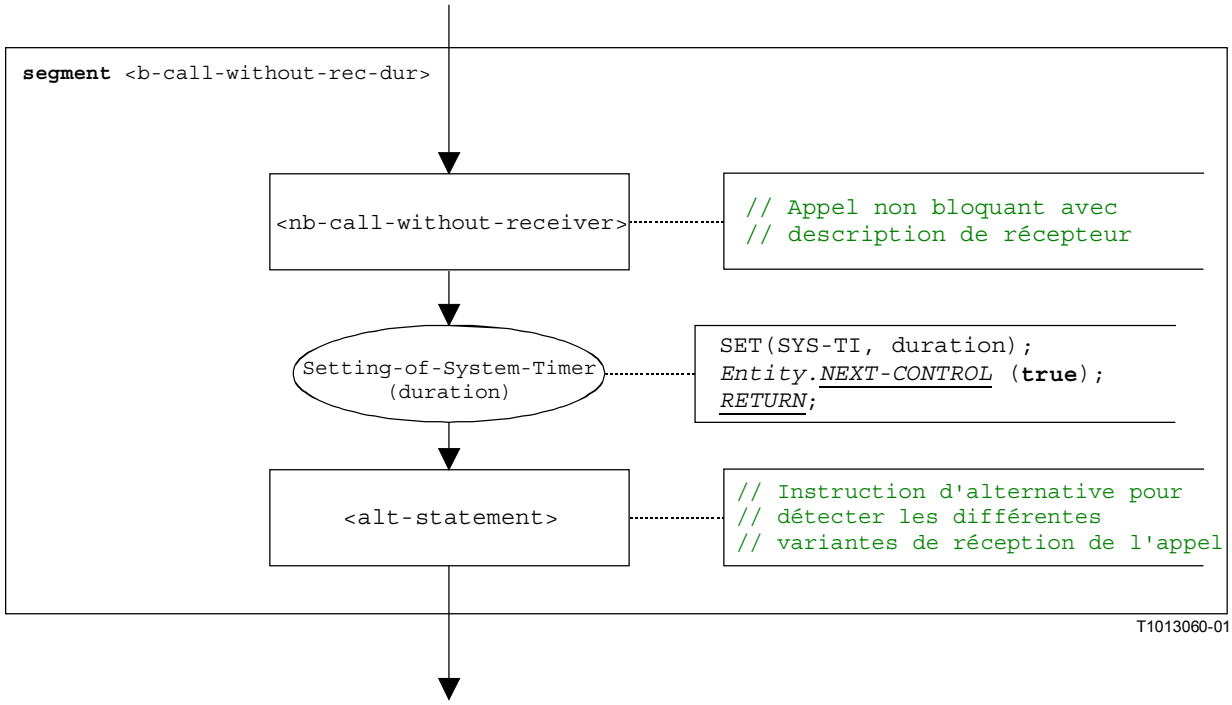


Figure B.38/Z.140 – Segment de graphe orienté <b-call-without-rec-dur>

B.3.7.4 Opération d'acquisition

La structure syntaxique de l'opération **catch** est:

```
<portId>.catch (<matchingSpec>) [from <component_expression>] ->
                                                    [<assignmentPart>]
```

L'expression facultative <component_expression> dans la clause **from** se rapporte à l'expéditeur de l'exception. Elle peut être fournie sous la forme d'une valeur de variable ou de la valeur de retour d'une fonction, c'est-à-dire que l'on part de l'hypothèse qu'il s'agit d'une expression. L'expression facultative <assignmentPart> indique l'affectation d'informations acquises si l'exception acquise s'apparie avec la spécification d'appariement <matchingSpec> et avec la clause (facultative) **from**.

Le segment de graphe orienté <catch-op> dans la Figure B.39 définit l'exécution d'une opération **catch**.

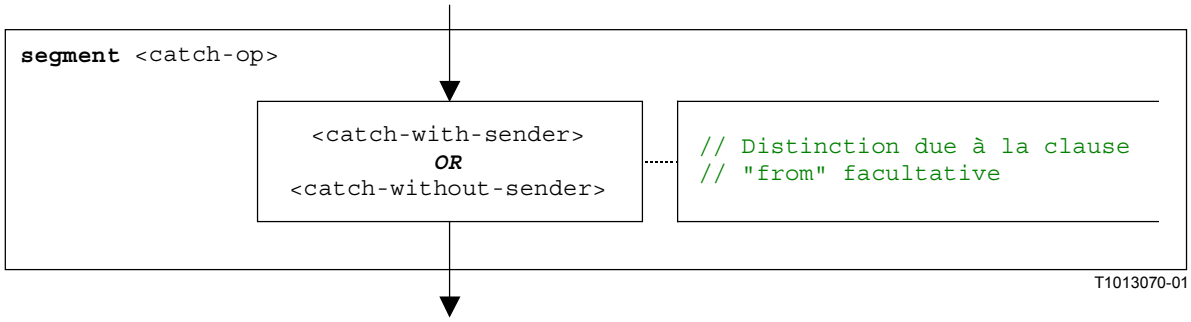


Figure B.39/Z.140 – Segment de graphe orienté <catch-op>

B.3.7.4.1 Segment de graphe orienté <catch-with-sender>

Le segment de graphe orienté <catch-with-sender> dans la Figure B.40 définit l'exécution d'une opération **catch** où l'expéditeur est spécifié sous la forme d'une expression.

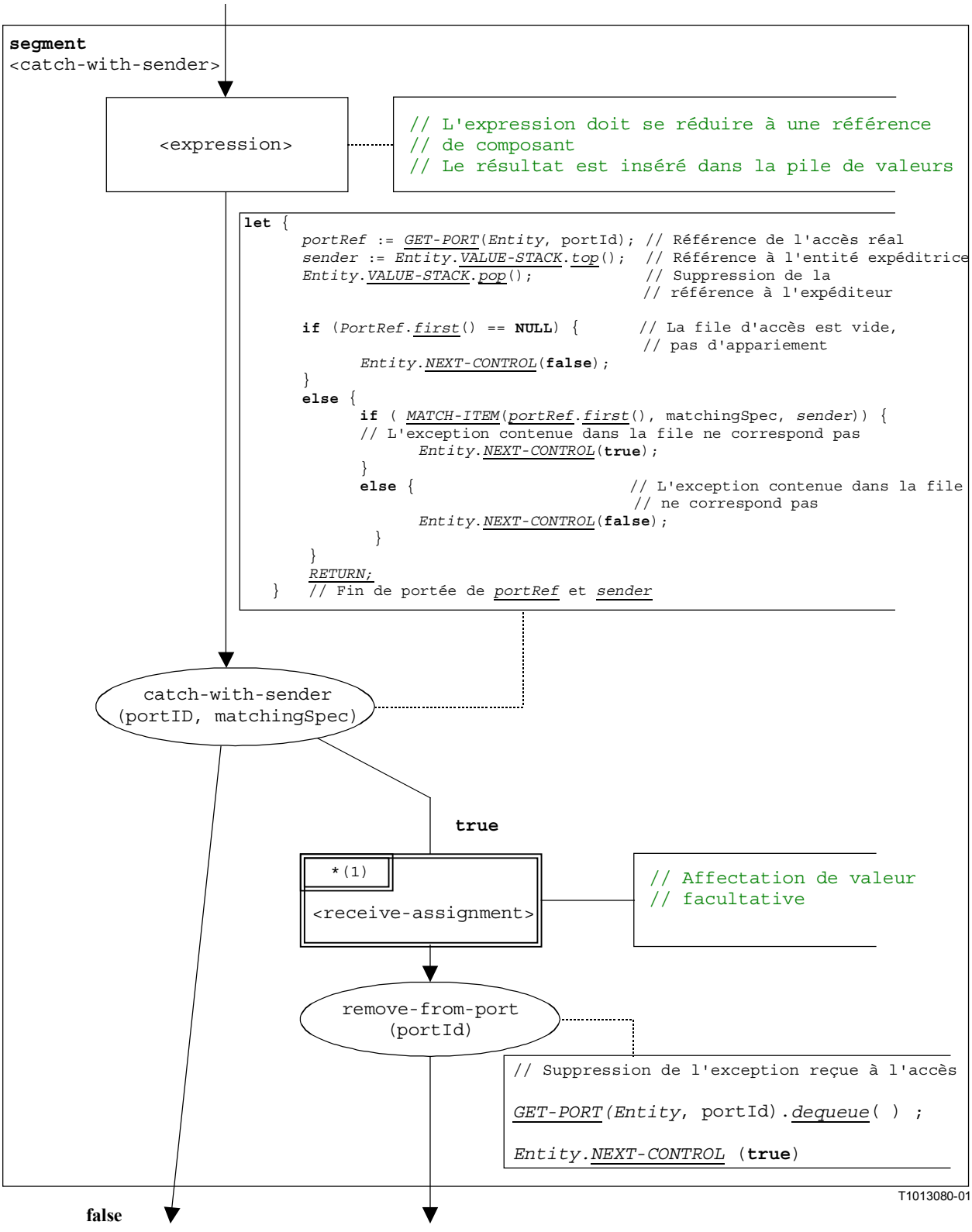


Figure B.40/Z.140 – Segment de graphe orienté <catch-with-sender>

B.3.7.4.2 Segment de graphe orienté <catch-without-sender>

Le segment de graphe orienté `<catch-without-sender>` dans la Figure B.41 définit l'exécution d'une opération **`catch`** sans clause **`from`**.

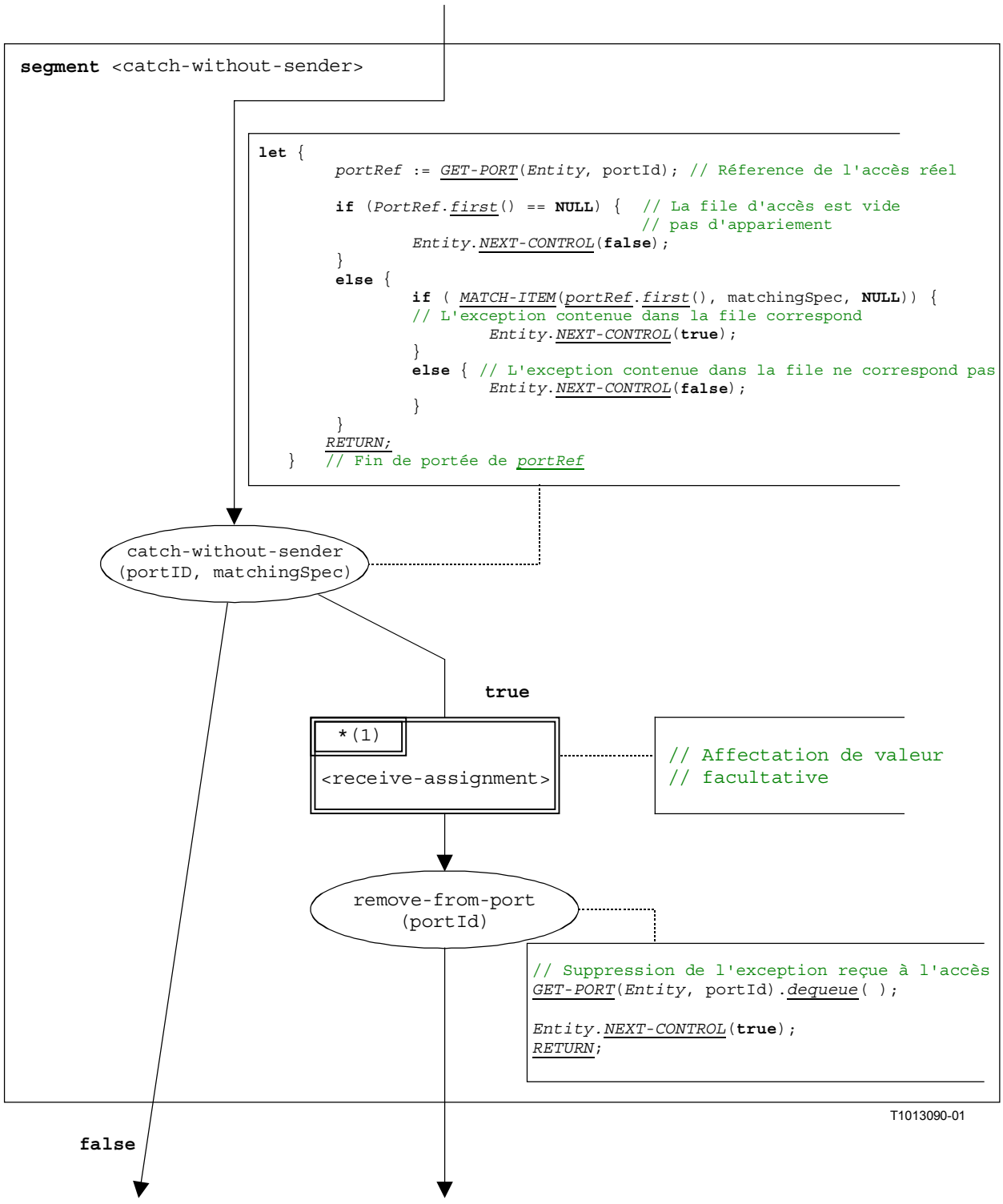


Figure B.41/Z.140 – Segment de graphe orienté <catch-without-sender>

B.3.7.5 Opération de libération d'accès

La structure syntaxique de l'opération **clear** est:

<portId>**.clear**

Le segment de graphe orienté <clear-port-op> dans la Figure B.42 définit l'exécution de l'opération **clear**.

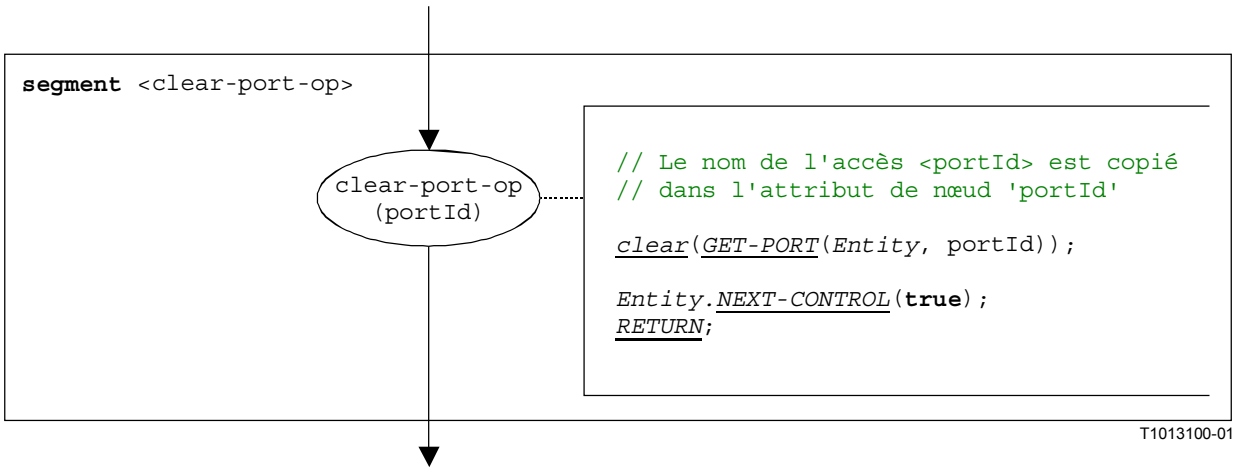


Figure B.42/Z.140 – Segment de graphe orienté <clear-port-op>

B.3.7.6 Opération de connexion

La structure syntaxique de l'opération **connect** est:

connect(<component_expression$_1$>.<portId1>,<component_expression$_2$>.<portId2>)

Les identificateurs <portId1> et <portId2> sont considérés comme étant des identificateurs d'accès des composants de test correspondants. Les composants auxquels l'accès appartient sont référencés au moyen des références de composant <component_expression$_1$> et <component_expression$_2$>. Les références peuvent être mémorisées dans des variables ou sont renvoyées par une fonction. Par souci de simplicité nous les considérerons comme des expressions qui se réduisent à une référence de composant. Donc, la pile de valeurs est utilisée pour mémoriser la référence de composants.

L'exécution de l'opération **connect** est définie par le segment de graphe orienté <connect-op> indiqué dans la Figure B.43. Dans le graphe orienté décrit, la première expression à réduire se rapporte à <component_expression$_1$> et la deuxième expression se rapporte à <component_expression$_2$>, c'est-à-dire que l'expression <component_expression$_2$> est au sommet de la pile de valeurs lorsque le nœud connect-op est exécuté.

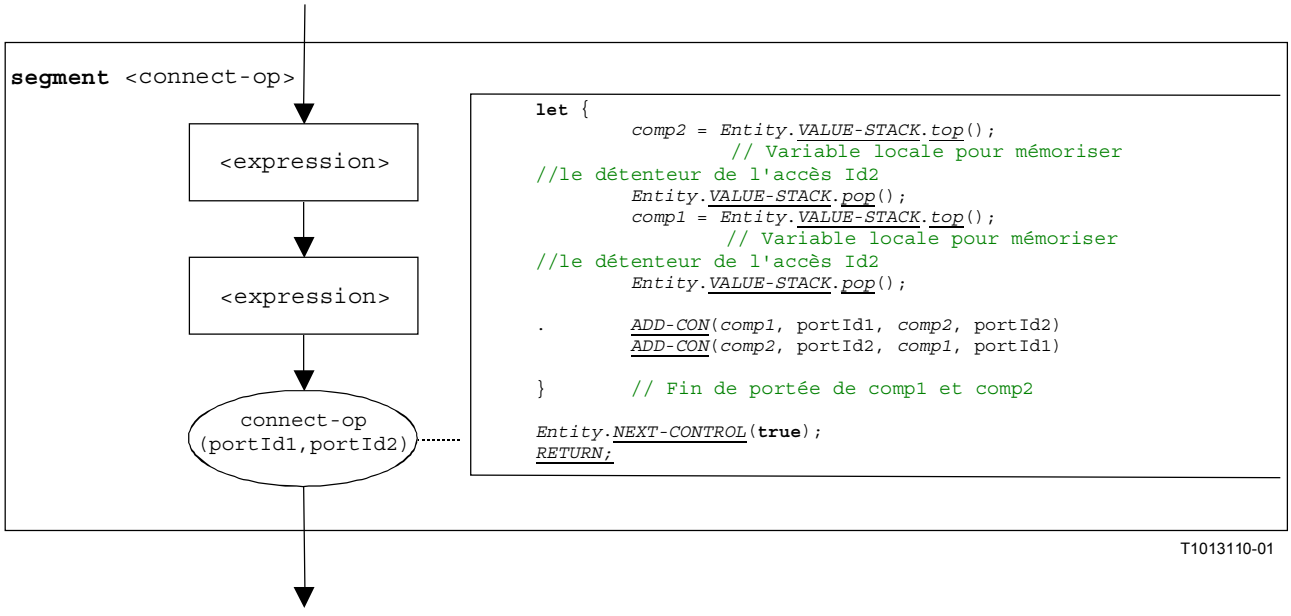


Figure B.43/Z.140 – Segment de graphe orienté <connect-op>

B.3.7.7 Déclaration d'une constante

La structure syntaxique d'une déclaration de constante est:

```
const <constType> <constId>:= <constType-expression>
```

La valeur d'une constante est considérée comme étant une expression qui donne une valeur du type de la constante.

NOTE – Des constantes globales sont remplacées par leurs valeurs dans une étape de prétraitement avant que cette sémantique soit appliquée (§ B.2.3). Des constantes locales sont traitées comme des déclarations de variable avec initialisation. L'utilisation correcte des constantes, c'est-à-dire que des constantes ne doivent jamais apparaître à gauche d'une affectation, doit être vérifiée au cours de l'analyse sémantique statique d'un module TTCN-3.

Le segment de graphe orienté <constant-déclaration> dans la Figure B.44 définit l'exécution d'une déclaration de constante où la valeur de la constante est fournie sous la forme d'une expression.

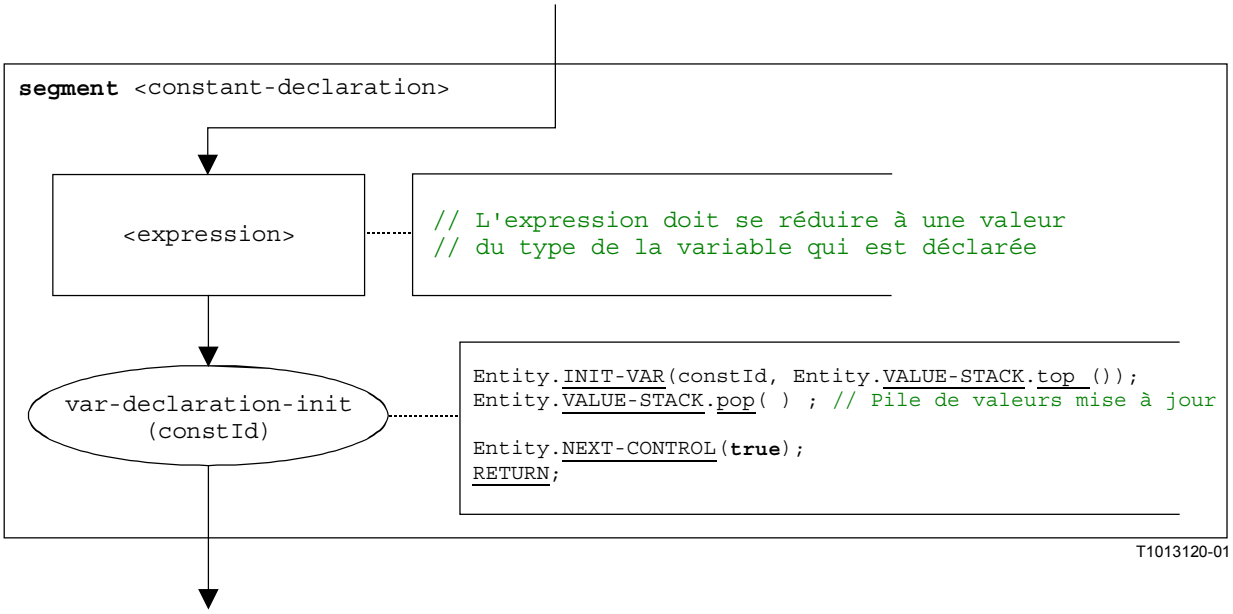


Figure B.44/Z.140 – Segment de graphe orienté <constant-declaration>

B.3.7.8 Opération de création

La structure syntaxique de l'opération **create** est:

```
<componentTypeId>.create
```

Le segment de graphe orienté <create-op> dans la Figure B.45 définit l'exécution de l'opération **create**.

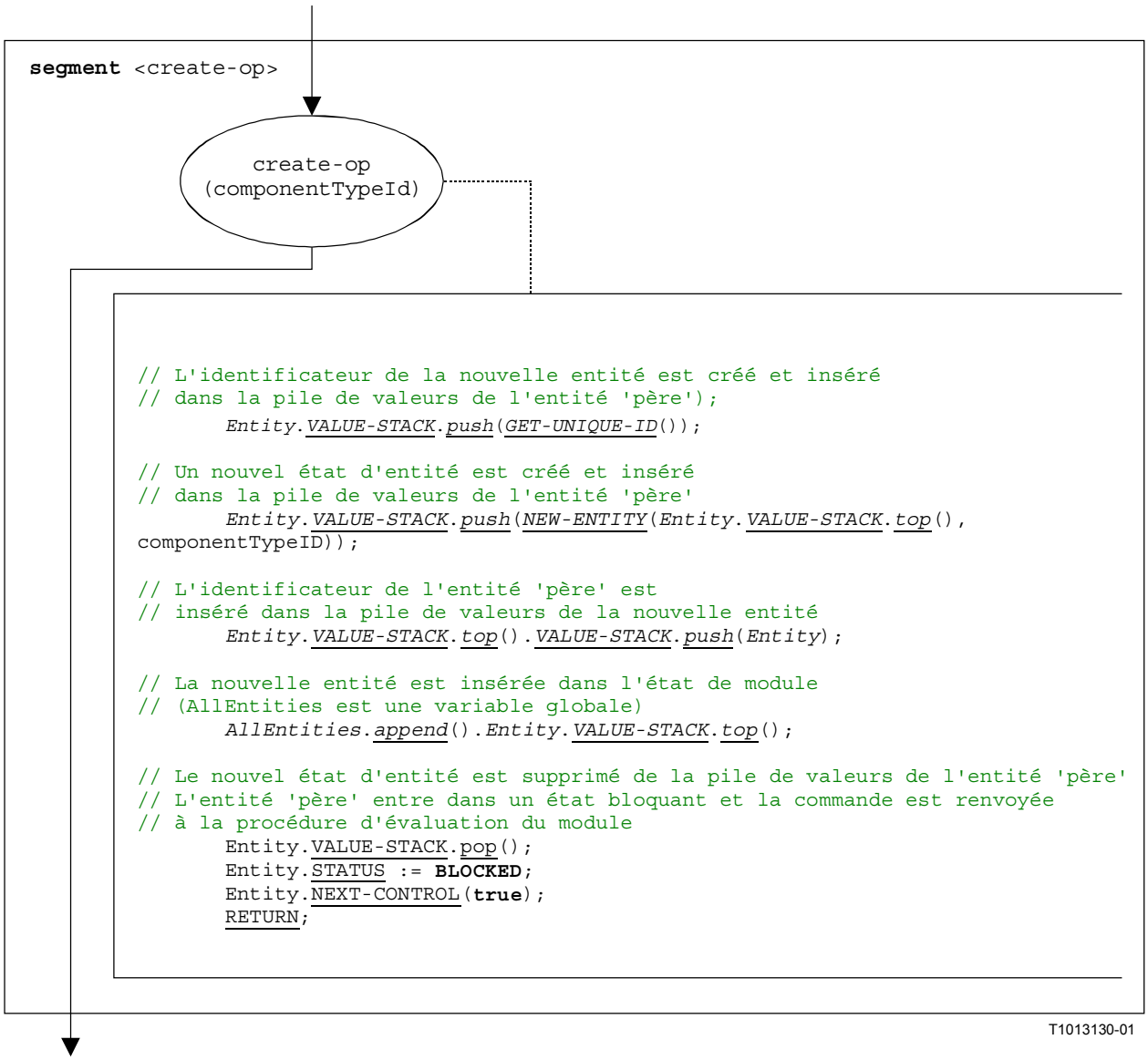


Figure B.45/Z.140 – Segment de graphe orienté <create-op>

B.3.7.9 Déclaration d'un accès

La structure syntaxique d'une déclaration d'accès est:

```
<portType> <portName>
```

Les déclarations d'accès peuvent être trouvées dans les définitions de type de composant. L'effet d'une déclaration d'accès est la création d'un nouvel accès. Le segment de graphe orienté <port-declaration> dans la Figure B.46 définit l'exécution d'une déclaration d'accès.

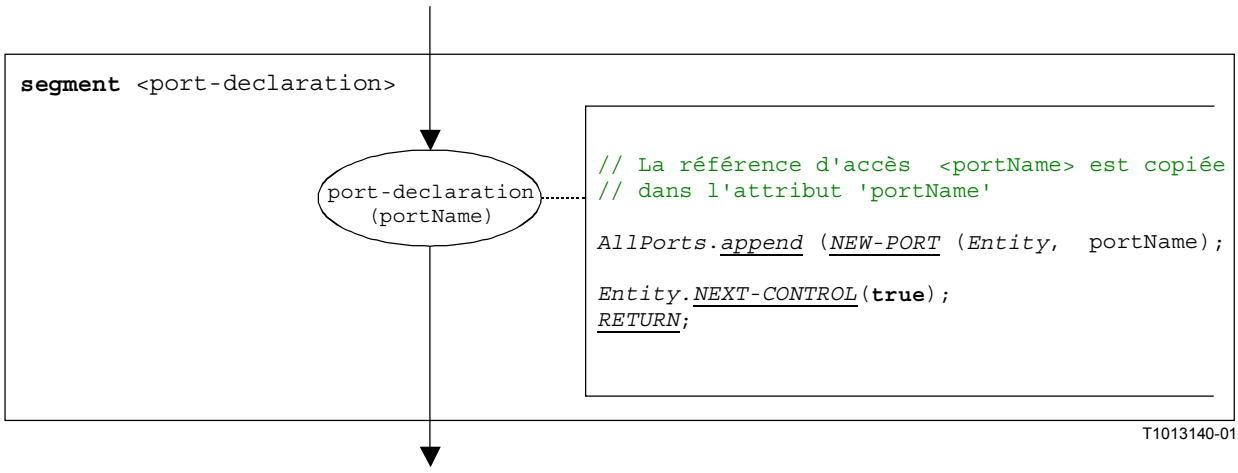


Figure B.46/Z.140 – Segment de graphe orienté <port-declaration>

B.3.7.10 Déclaration d'un temporisateur

La structure syntaxique d'une déclaration de temporisateur est:

timer <timerId> [:= <float_expression>]

L'effet d'une déclaration de temporisateur est la création d'une nouvelle corrélation de temporisateur. La déclaration d'une variable avec une durée par défaut est facultative. La valeur par défaut est considérée comme étant une expression qui donne une valeur du type **float**.

Le segment de graphe orienté <timer-declaration> dans la Figure B.47 définit l'exécution de l'opération de déclaration d'un temporisateur.

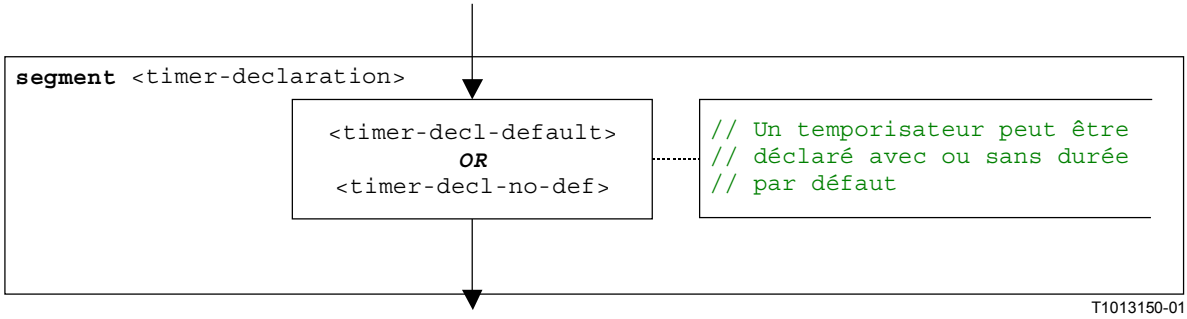


Figure B.47/Z.140 – Segment de graphe orienté <timer-declaration>

B.3.7.10.1 Segment de graphe orienté <timer-decl-default>

Le segment de graphe orienté <timer-decl-default> dans la Figure B.48 définit l'exécution d'une déclaration de temporisateur où une durée par défaut est fournie sous la forme d'une expression.

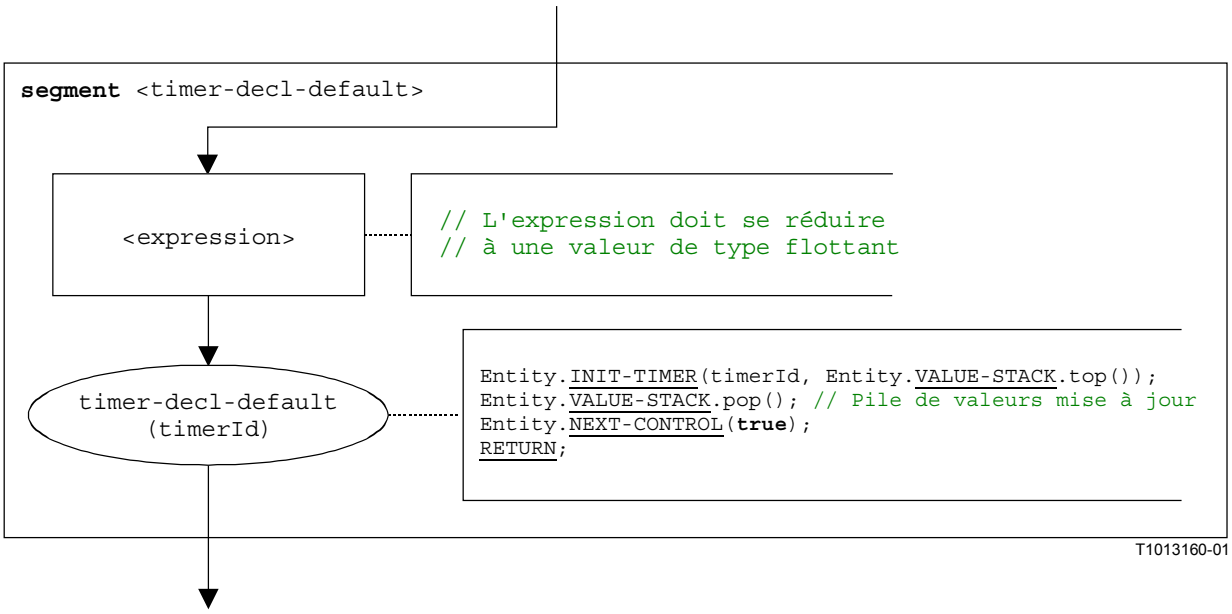


Figure B.48/Z.140 – Segment de graphe orienté <timer-decl-default>

B.3.7.10.2 Segment de graphe orienté <timer-decl-no-def>

Le segment de graphe orienté <timer-decl-no-def> dans la Figure B.49 définit l'exécution d'une déclaration de temporisateur où aucune durée par défaut n'est fournie, c'est-à-dire que la durée par défaut du temporisateur est indéfinie.

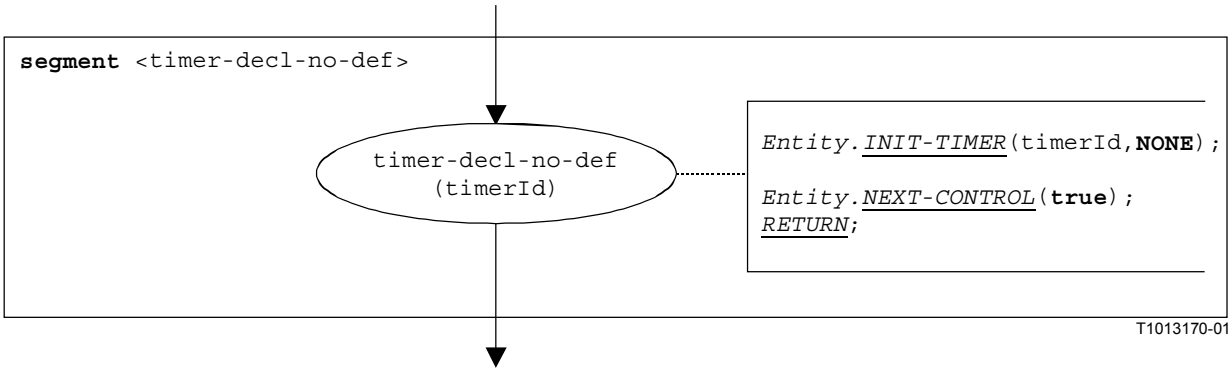


Figure B.49/Z.140 – Segment de graphe orienté <timer-decl-no-def>

B.3.7.11 Déclaration d'une variable

La structure syntaxique d'une déclaration de variable est:

var <varType> <varId> [:= <varType_expression>]

L'initialisation d'une variable par fourniture d'une valeur initiale est facultative. La valeur initiale est considérée comme étant une expression qui donne une valeur du type variable.

Le segment de graphe orienté <variable-declaration> dans la Figure B.50 définit l'exécution de l'opération de déclaration d'une variable.

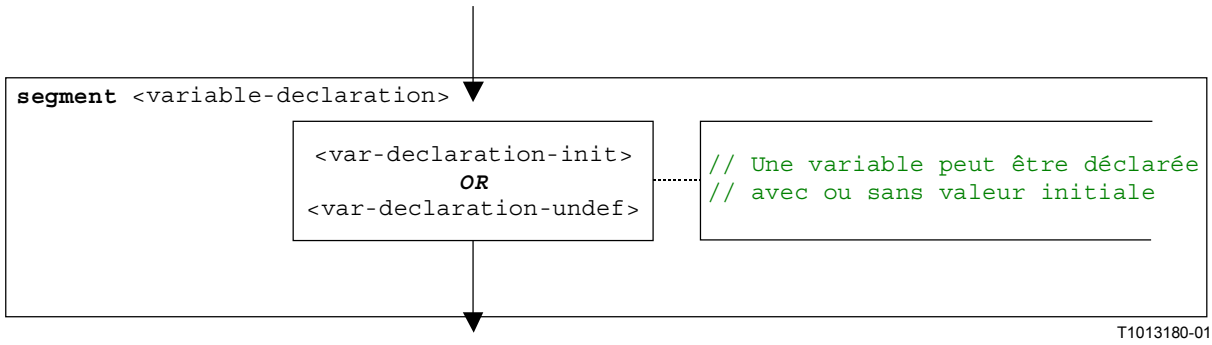


Figure B.50/Z.140 – Segment de graphe orienté <variable-declaration>

B.3.7.11.1 Segment de graphe orienté <var-declaration-init>

Le segment de graphe orienté <var-declaration-init> dans la Figure B.51 définit l'exécution d'une déclaration de variable où une valeur initiale est fournie sous la forme d'une expression.

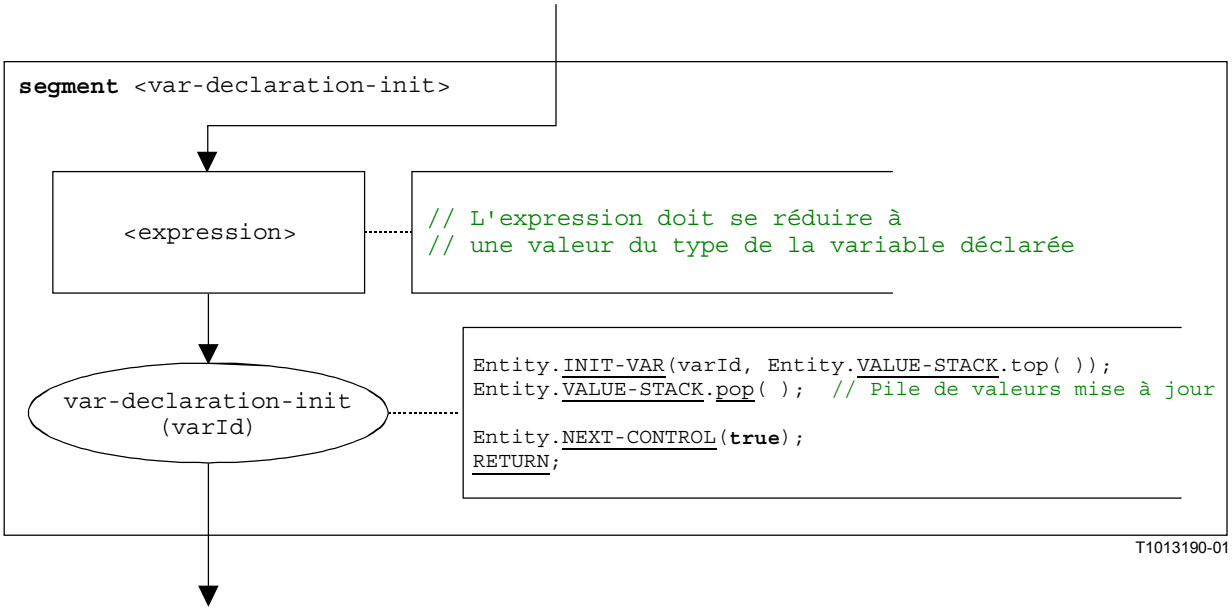


Figure B.51/Z.140 – Segment de graphe orienté <var-declaration-init>

B.3.7.11.2 Segment de graphe orienté <var-declaration-undef>

Le segment de graphe orienté <var-declaration-undef> dans la Figure B.52 définit l'exécution d'une déclaration de variable où aucune valeur initiale n'est fournie, c'est-à-dire que la valeur de la variable est indéfinie.

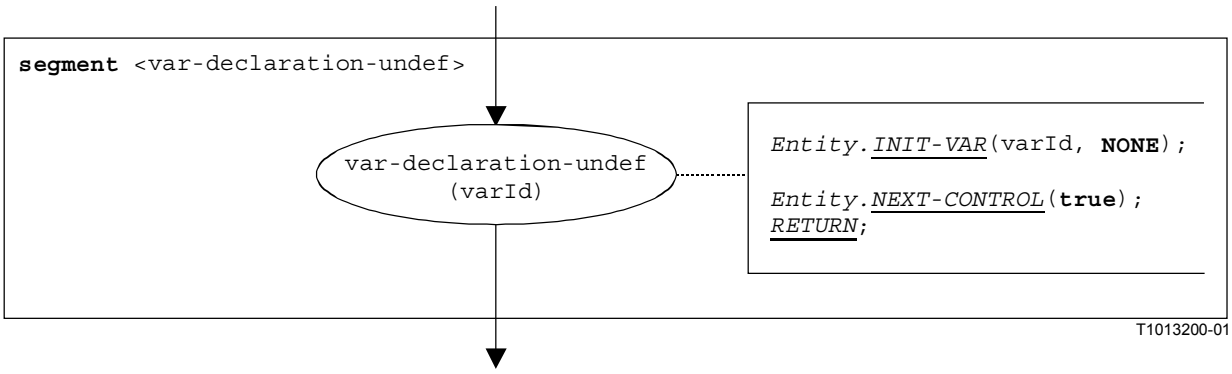


Figure B.52/Z.140 – Segment de graphe orienté <var-declaration-undef>

B.3.7.12 Opération de déconnexion

La structure syntaxique de l'opération **disconnect** est:

disconnect(<component_expression$_1$>.<portId1>,<component_expression$_2$>.<portId2>)

Les identificateurs <portId1> et <portId2> sont considérés comme étant des identificateurs d'accès des composants de test correspondants. Les composants auxquels l'accès appartient sont référencés au moyen des références de composant <component_expression$_1$> et <component_expression$_2$>. Les références peuvent être mémorisées dans des variables ou sont renvoyées par une fonction. Par souci de simplicité, nous les considérerons comme des expressions qui se réduisent à une référence de composant. Donc, la pile de valeurs est utilisée pour mémoriser la référence de composants.

L'exécution de l'opération **disconnect** est définie par le segment de graphe orienté <disconnect-op> indiqué dans la Figure B.53. Dans le segment de graphe orienté, la première expression à réduire se rapporte à <component_expression$_1$> et la deuxième expression se rapporte à <component_expression$_2$>, c'est-à-dire que l'expression <component_expression$_2$> est au sommet de la pile de valeurs lorsque le nœud disconnect-op est exécuté.

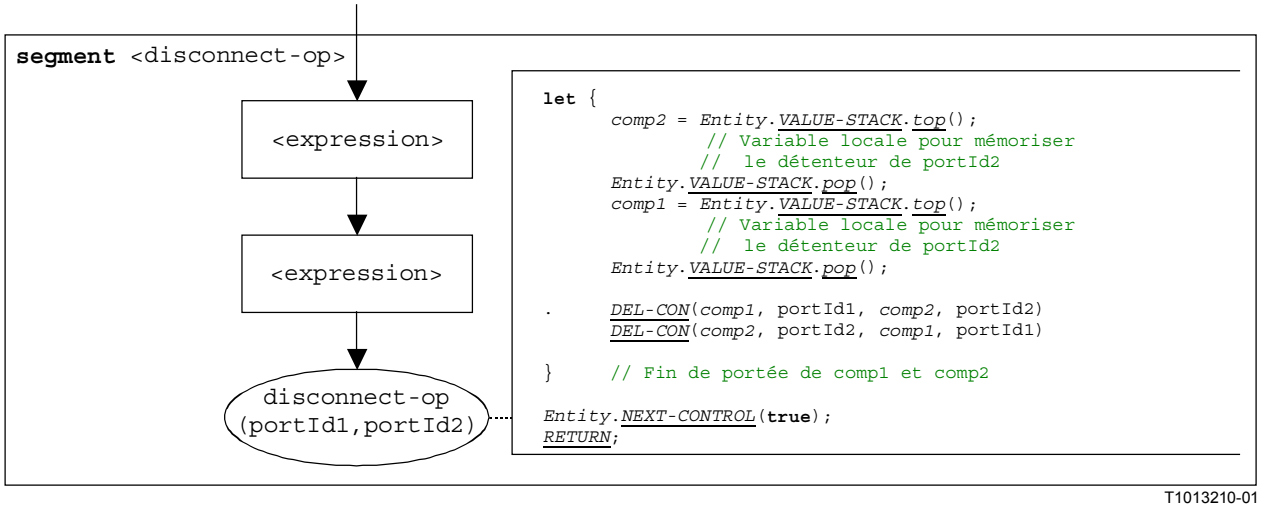


Figure B.53/Z.140 – Segment de graphe orienté <disconnect-op>

B.3.7.13 Instruction d'exécution tant que

La structure syntaxique de l'opération **do-while** est:

```
do <statement-block>
while (<boolean_expression>)
```

L'exécution d'une instruction **do-while** est définie par le segment de graphe orienté <do-while-stmt> indiqué dans la Figure B.54.

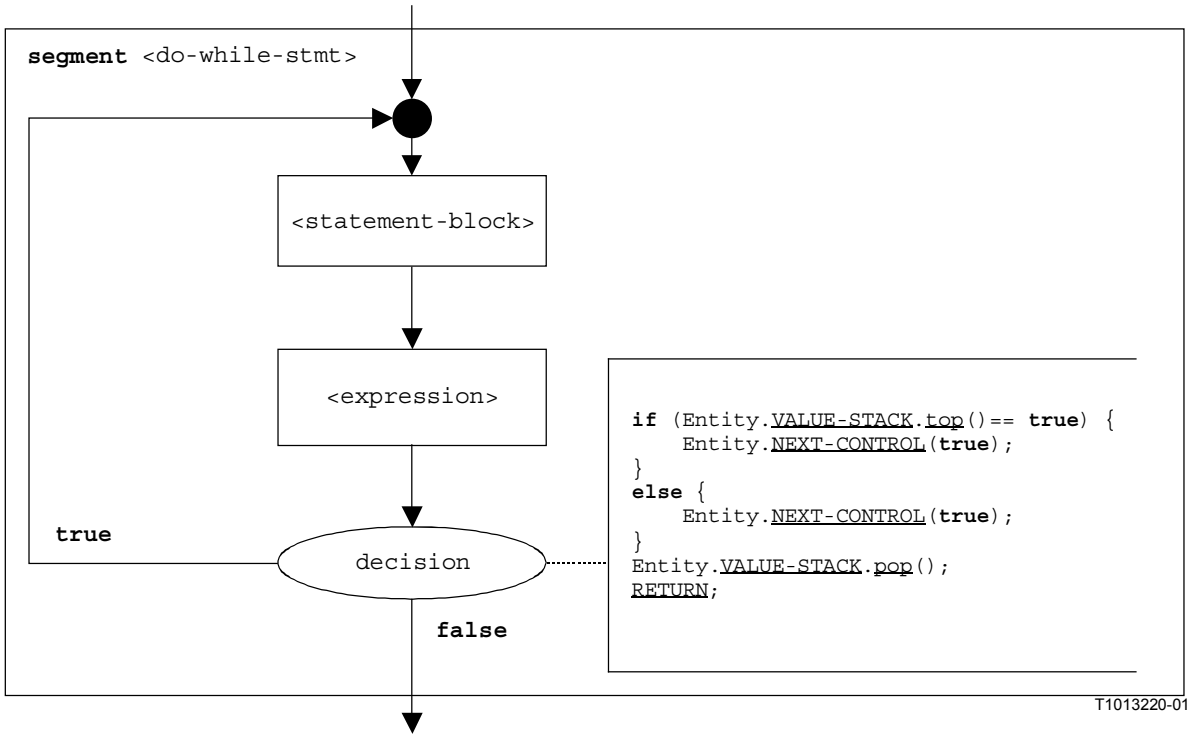


Figure B.54/Z.140 – Segment de graphe orienté <do-while-stmt>

B.3.7.14 Opération d'achèvement de tous les composants

L'opération **done-all-components** se rapporte à l'usage du mot clé **all component** dans l'opération **done** (§ B.7.16). L'opération **done-all-components** ne peut être appelée que par le composant **mtc**. Elle permet de vérifier si tous les composants de test parallèles d'un test élémentaire se sont terminés. La structure syntaxique de l'opération **done-all-components** est:

```
all component.done;
```

L'exécution de l'opération **done-all-components** est définie par le segment de graphe orienté <done-all-comp-op> dans la Figure B.55.

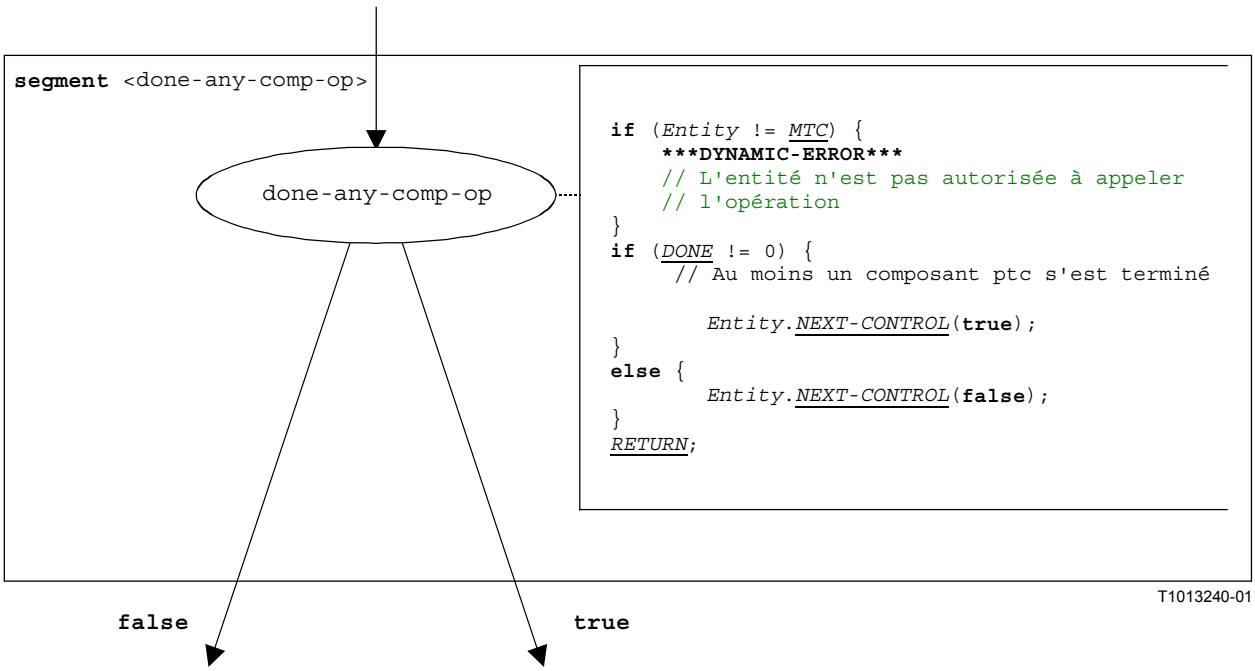


Figure B.55/Z.140 – Segment de graphe orienté <done-all-comp-op>

B.3.7.15 Opération d'achèvement d'un composant quelconque

L'opération **done-any-component** se rapporte à l'usage du mot clé **any component** dans l'opération **done** (§ B.7.16). L'opération **done-any-component** ne peut être appelée que par le composant **mtc**. Elle permet de vérifier si un composant de test parallèle d'un test élémentaire est déjà terminé. La structure syntaxique de l'opération **done-any-component** est:

```
any component.done;
```

L'exécution de l'opération **done-any-component** est définie par le segment de graphe orienté <done-any-comp-op> dans la Figure B.56.

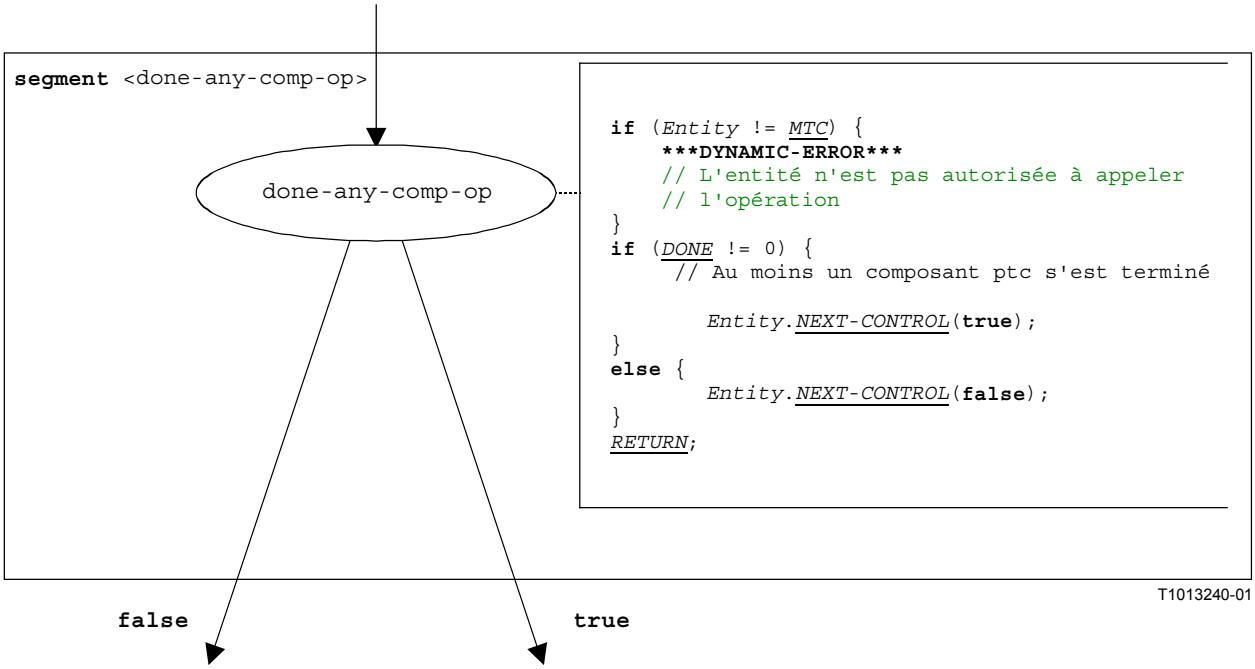


Figure B.56/Z.140 – Segment de graphe orienté <done-any-comp-op>

B.3.7.16 Opération d'achèvement de composant

La structure syntaxique de l'opération "**done** component" est:

<component_expression>.**done**

L'opération "**done** component" vérifie si un composant est en cours ou s'est arrêté. Selon qu'un composant vérifié est en cours ou s'est arrêté l'opération **done** décide la façon dont le flux de commande continue. L'utilisation d'une référence de composant identifie le composant à vérifier. La référence peut être mémorisée dans une variable ou être renvoyée par une fonction. Par souci de simplicité, cela est considéré comme étant une expression qui se réduit à une référence de composant.

Le segment de graphe orienté <**done-component-op**> dans la Figure B.57 définit l'exécution de l'opération "**done** component".

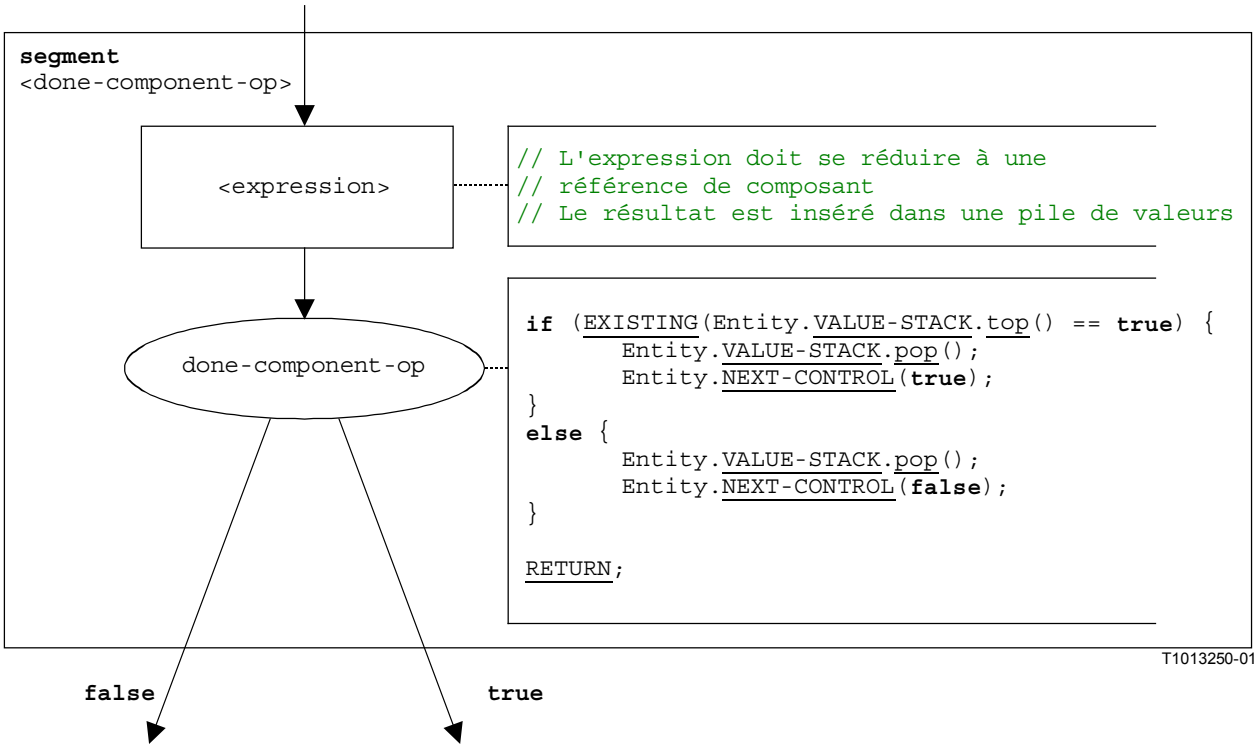


Figure B.57/Z.140 – Segment de graphe orienté <done-component-op>

B.3.7.17 Instruction d'exécution

La structure syntaxique de l'opération **execute** est:

execute(<testCaseId>([<act-par$_1$>, ... , <act-par$_n$>)]) [, <float_expression>])

L'instruction **execute** décrit l'exécution d'un test élémentaire <testCaseId> avec les paramètres réels (facultatifs) <act-par$_1$>, ... , <act-par$_n$>. Facultativement, l'instruction d'exécution peut être mise en instance pendant une durée fournie sous la forme d'une expression qui donne une valeur en virgule flottante. Si au cours de la durée spécifiée le test élémentaire ne renvoie pas de verdict, une exception de temporisation se produit, le test élémentaire est arrêté et un verdict de type **error** est renvoyé. Cependant, la notation TTCN-3 n'a pas de sémantique en temps réel et, donc, la décision quant à la question de savoir si une exception de temporisation se produit ou non est modélisée sous la forme d'un choix non déterministe.

NOTE – La sémantique opérationnelle ne modélise que le choix non déterministe. L'expression <float_expression> n'est pas évaluée.

Si, en raison du choix non déterministe, aucune exception de temporisation ne se produit, le composant **mtc** est créé, l'instance de commande (représentant la partie commande du module TTCN-3) est bloquée jusqu'à ce que le test élémentaire se termine et, pour le prochain test élémentaire à exécuter, le flux de commande est transmis au composant **mtc**. Le flux de commande est rendu à l'instance de commande lorsque le composant **mtc** se termine.

Le segment de graphe orienté <execute-stmt> dans la Figure B.58 définit l'exécution d'une instruction **execute**.

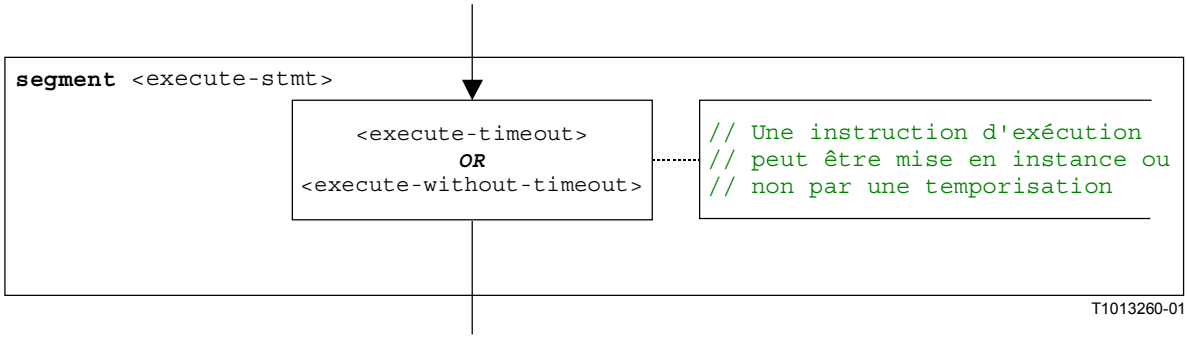


Figure B.58/Z.140 – Segment de graphe orienté <execute-stmt>

B.3.7.17.1 Segment de graphe orienté <execute-timeout>

Le segment de graphe orienté <execute-timeout> dans la Figure B.59 définit l'exécution d'une instruction **execute** qui est mise en instance par une valeur de temporisation.

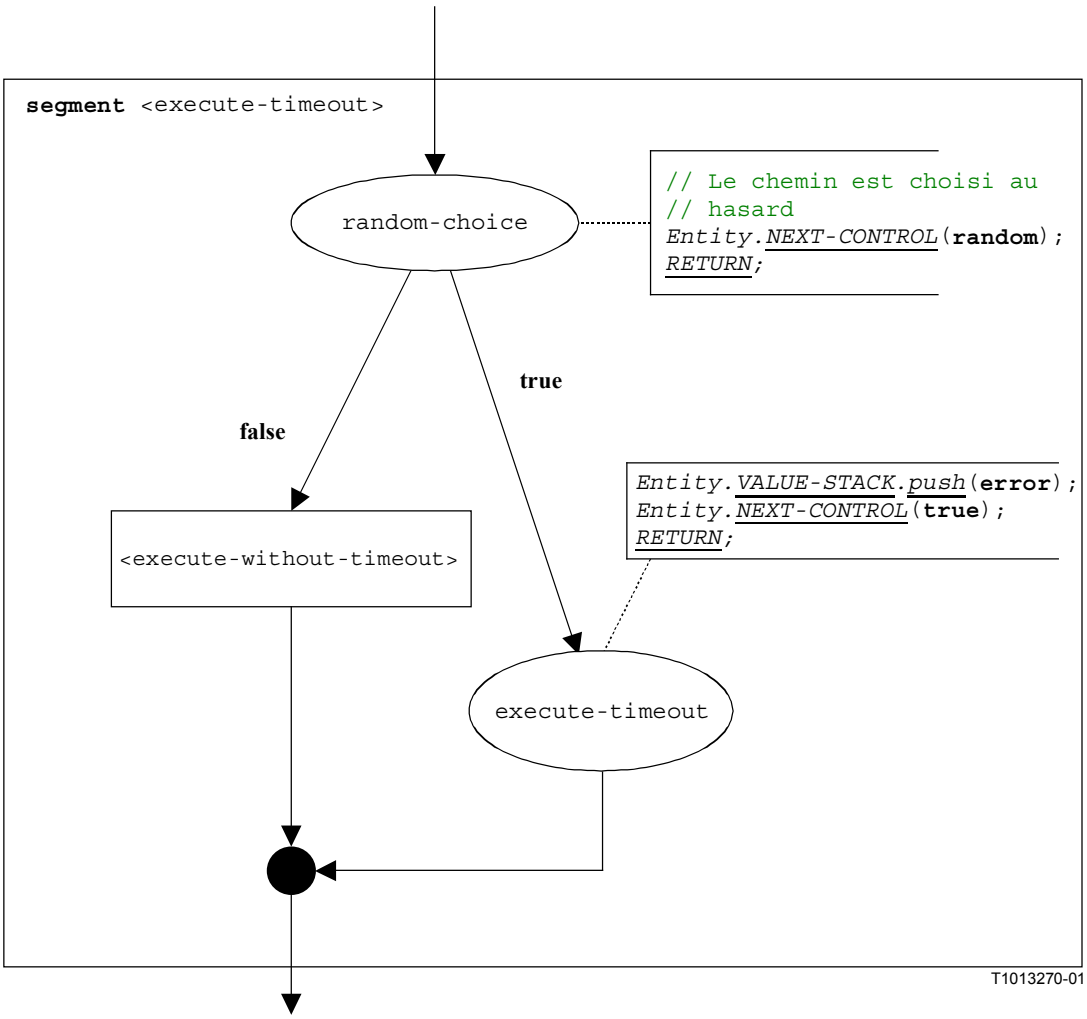


Figure B.59/Z.140 – Segment de graphe orienté <execute-timeout>

B.3.7.17.2 Segment de graphe orienté <execute-without-timeout>

L'exécution d'un test élémentaire commence par la création du composant **mtc**. Ensuite, le composant mtc est lancé avec le comportement défini dans la définition de test élémentaire. Ensuite, la commande de module attend que le test élémentaire se termine. La création et le démarrage du composant mtc peuvent être décrits au moyen des instructions **create** et **start**:

```
mtcType MyMTC:= mtcType.create;
MyMTC.start(TestCaseName(P1…Pn);
```

Le segment de graphe orienté <execute-without-timeout> dans la Figure B.60 définit l'exécution d'une instruction **execute** sans l'occurrence d'une exception de temporisation au moyen des segments de graphe orienté des opérations **create** et **start**.

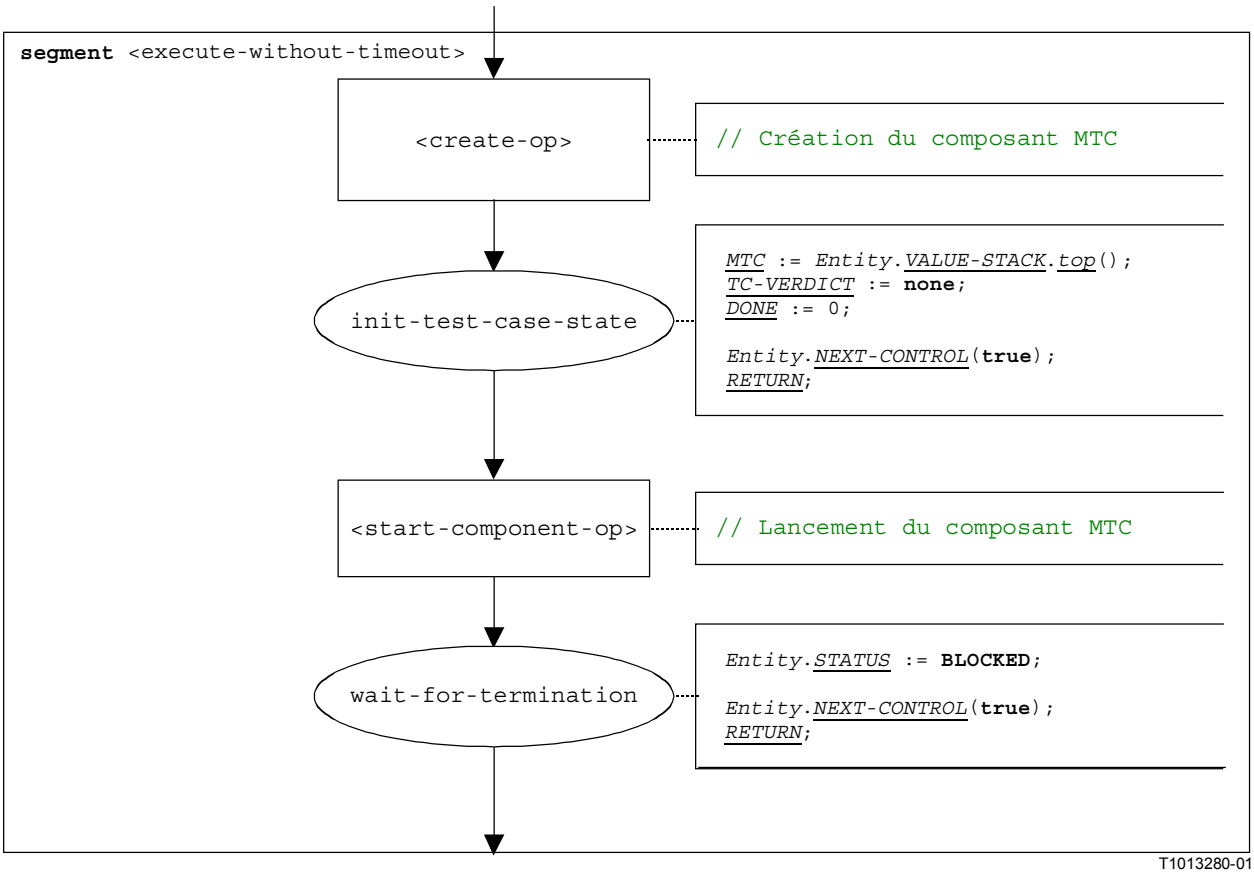


Figure B.60/Z.140 – Segment de graphe orienté <execute-without-timeout>

B.3.7.18 Expression

Pour le traitement d'expressions, les quatre cas suivants doivent être distingués:

a) L'expression est une valeur de littéral (ou une constante);

b) L'expression est une variable;

c) L'expression est un opérateur appliqué à un ou plusieurs opérandes;

d) L'expression est un appel de fonction ou d'opération.

La structure syntaxique d'une expression est:

`<lit-val> | <var-val> | <func-op-call> | <operand-appl>`

où:

`<lit-val>`	indique une valeur de littéral;
`<var-val>`	indique une valeur de variable;
`<func-op-call>`	indique un appel de fonction ou d'opération;
`<operator-appl>`	indique l'application d'opérateurs arithmétiques comme +, -, not, etc.

L'exécution d'une expression est définie par le segment de graphe orienté `<expression>` indiqué dans la Figure B.61.

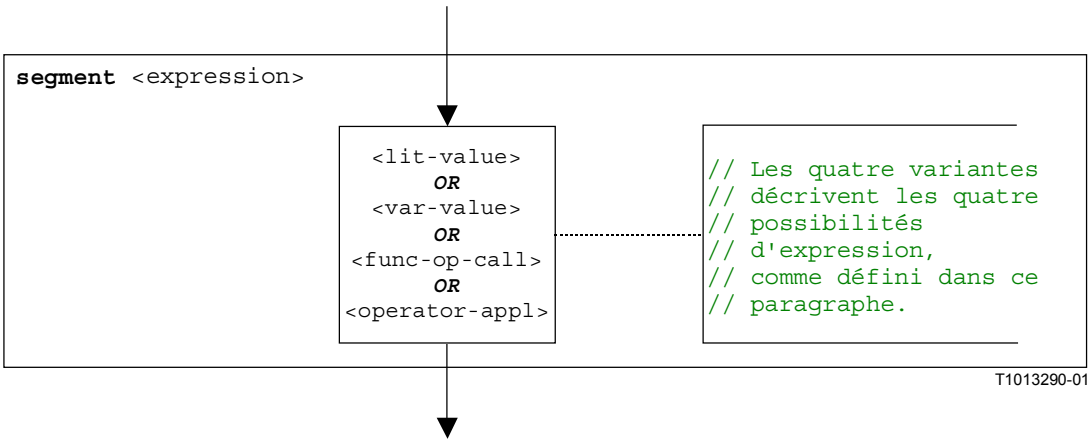


Figure B.61/Z.140 – Segment de graphe orienté <expression>

B.3.7.18.1 Segment de graphe orienté <lit-value>

Le segment de graphe orienté <lit-value> dans la Figure B.62 fait passer une valeur de littéral à la pile de valeurs d'une entité.

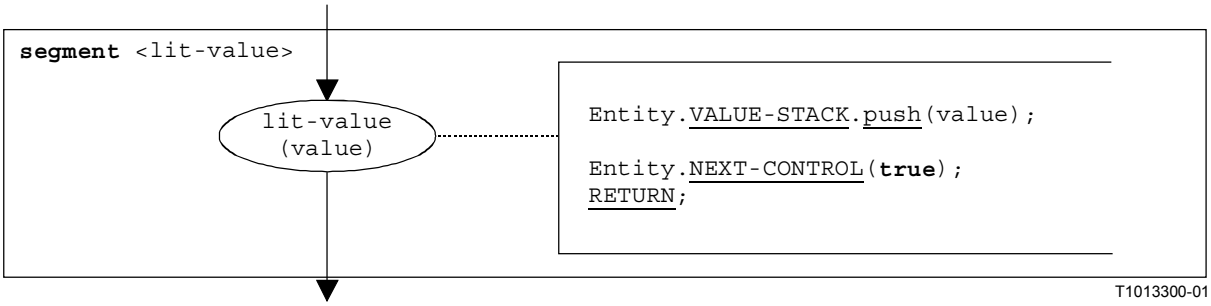


Figure B.62/Z.140 – Segment de graphe orienté <lit-value>

B.3.7.18.2 Segment de graphe orienté <var-value>

Le segment de graphe orienté <var-value> dans la Figure B.63 fait passer la valeur d'une variable à la pile de valeurs d'une entité.

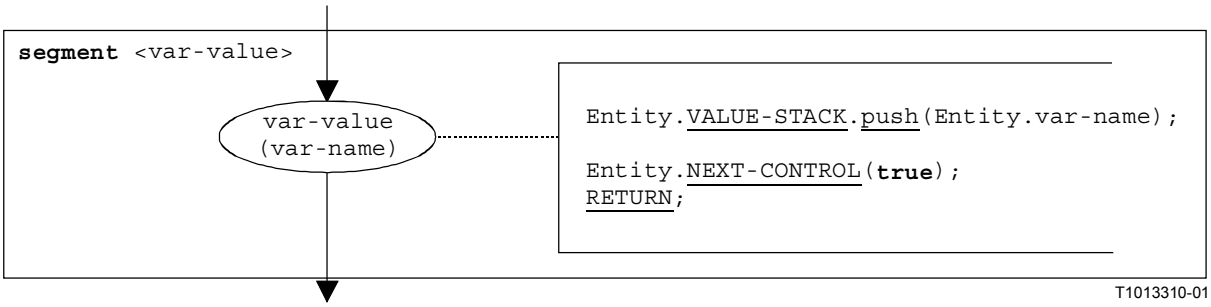


Figure B.63/Z.140 – Segment de graphe orienté <var-value>

B.3.7.18.3 Segment de graphe orienté <func-op-call>

Le segment de graphe orienté <func-op-call> dans la Figure B.64 se rapporte à des appels de fonction et à des opérations, qui renvoient une valeur qui est insérée dans la pile de valeurs d'une entité. Tous ces appels sont considérés comme étant des expressions.

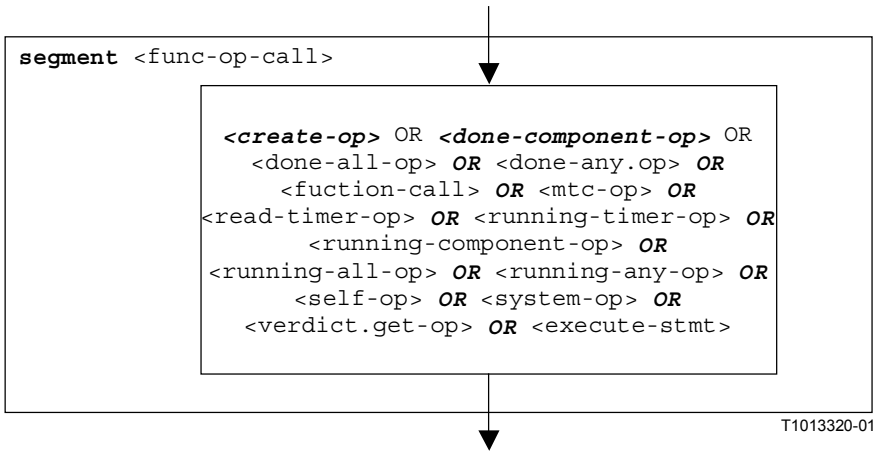


Figure B.64/Z.140 – Segment de graphe orienté <func-op-call>

B.3.7.18.4 Segment de graphe orienté <operator-appl>

La représentation par graphe orienté dans la Figure B.65 se rapporte directement à l'hypothèse que la notation polonaise inverse est utilisée pour évaluer des expressions d'opérateur. Les opérandes de l'opérateur sont calculés et insérés dans la pile d'évaluation. Pour l'application de l'opérateur, les opérandes sont désempilés de la pile d'évaluation et l'opérateur est appliqué. Le résultat de l'application de l'opérateur est finalement inséré dans la pile d'évaluation.

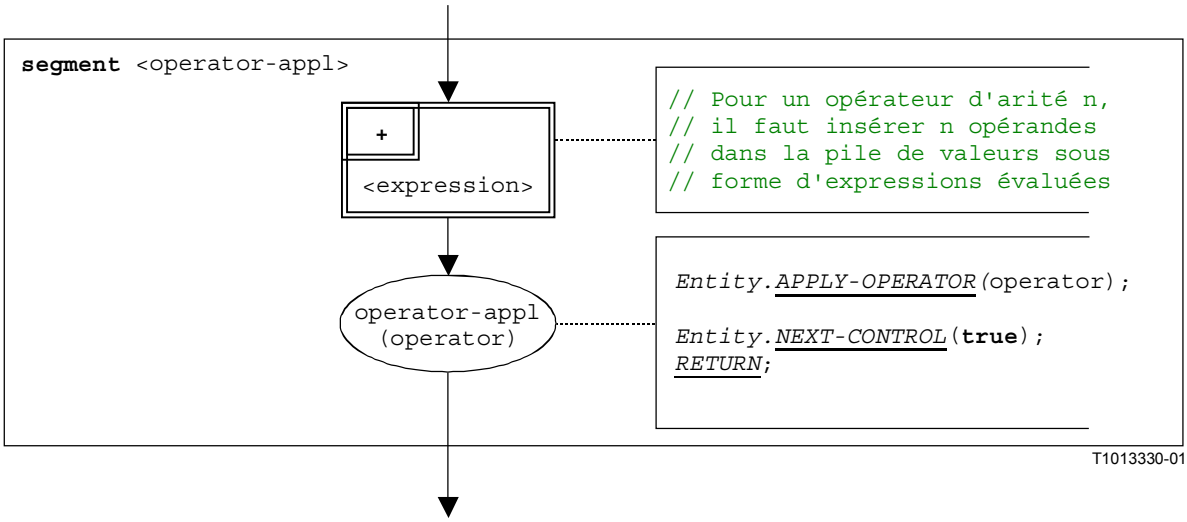


Figure B.65/Z.140 – Segment de graphe orienté <operator-appl>

B.3.7.19 Segment de graphe orienté <finalize-component-init>

Le segment de graphe orienté `<finalize-component-init>` fait partie du graphe orienté représentant le comportement d'une définition de type de composant. Son exécution est définie dans la Figure B.66:

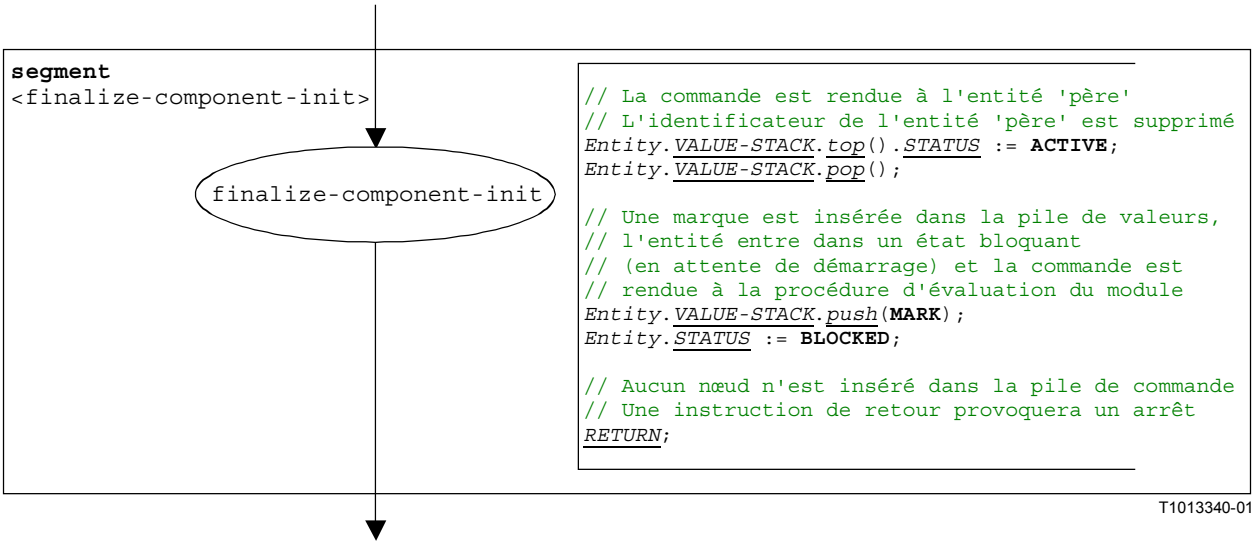


Figure B.66/Z.140 – Segment de graphe orienté <finalize-component-init>

B.3.7.20 Segment de graphe orienté <init-component-scope>

Le segment de graphe orienté `<init-component-scope>` fait partie du graphe orienté représentant le comportement d'une définition de type de composant. Son exécution est définie dans la Figure B.67:

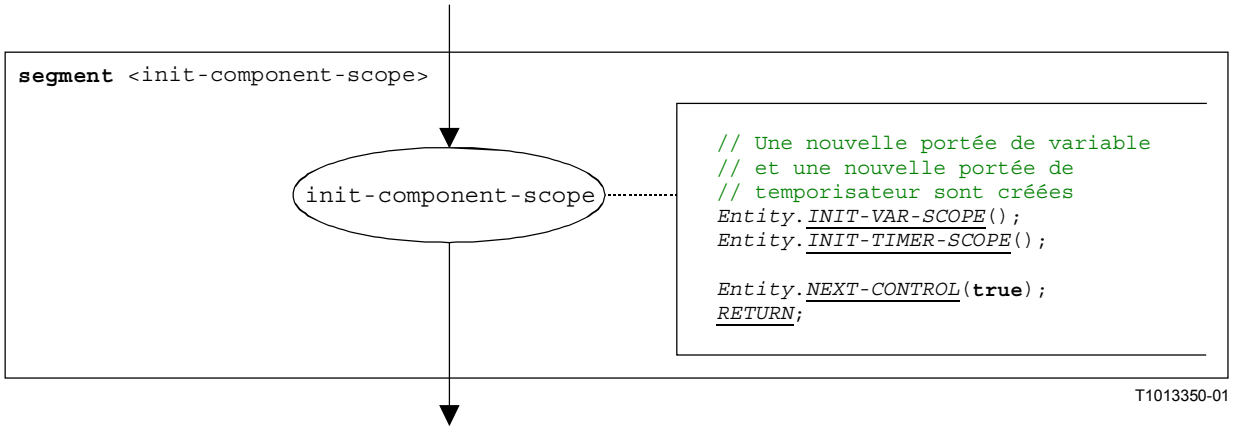


Figure B.67/Z.140 – Segment de graphe orienté <init-component-scope>

B.3.7.21 Instruction "pour" (for)

La structure syntaxique de l'opération **`for-statement`** est:

`for` (<assignment>, <boolean_expression>, <assignment>) <statement-block>

L'initialisation de la variable d'index et la manipulation correspondante de la variable d'index sont considérées comme étant des affectations à la variable d'index. Le segment <boolean_expression> décrit le critère de terminaison de la boucle spécifiée par l'instruction **`for-statement`** et le segment <statement-block> décrit le corps de boucle.

L'exécution de l'instruction "**`for-statement`**" est définie par le segment de graphe orienté <for-stmt> indiqué dans la Figure B.68. La structure <assignment> décrit l'initialisation de la variable d'index. La structure <assignment> contenue dans la branche **`true`** du nœud decision décrit la manipulation de la variable d'index.

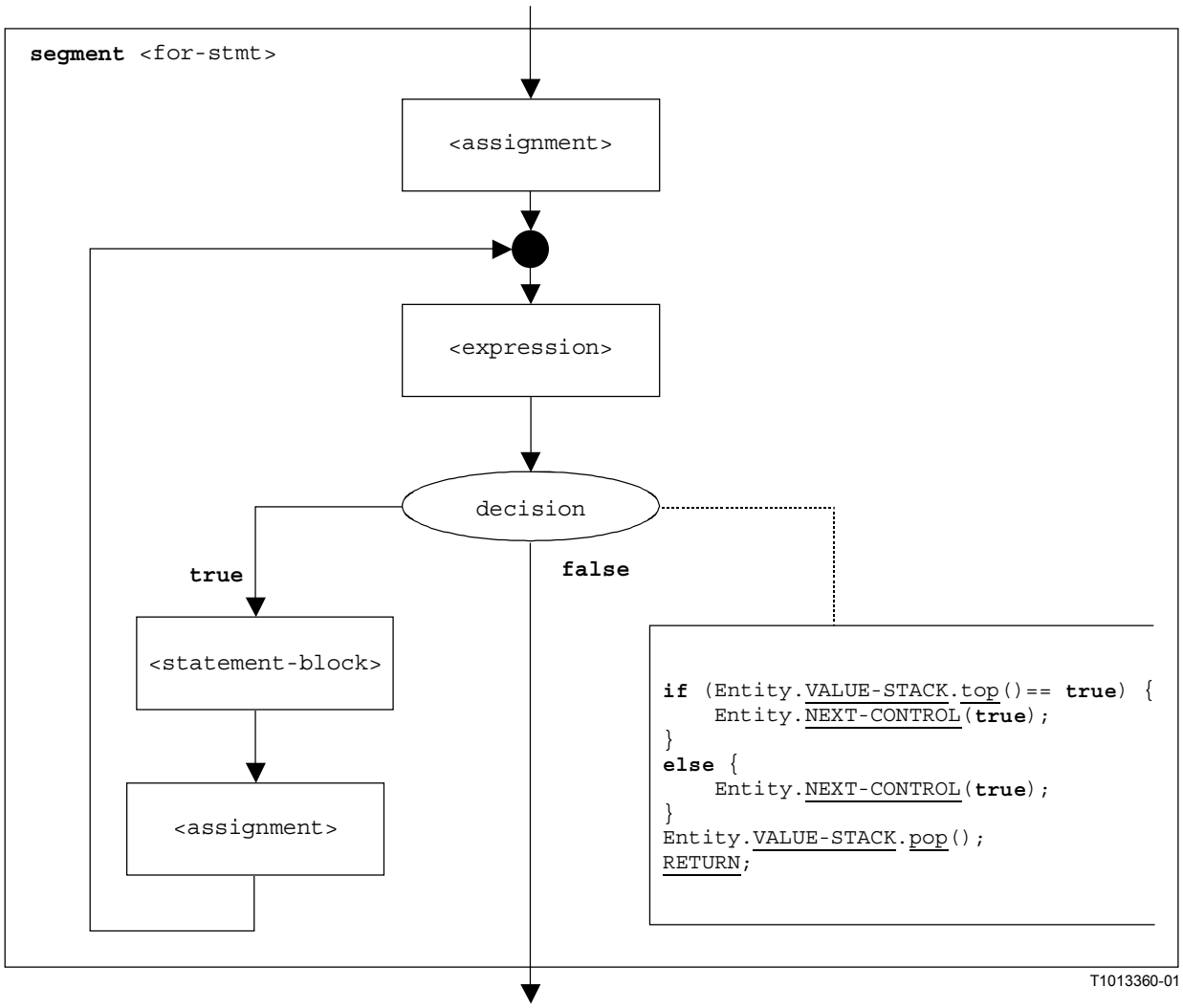


Figure B.68/Z.140 – Segment de graphe orienté <for-stmt>

B.3.7.22 Appel de fonction

La structure syntaxique d'un appel de fonction est:

<function-name>([<act-par-desc$_1$>, ... , <act-par-desc$_n$>])

La structure <function-name> indique le nom d'une fonction et la structure <act-par-descr$_1$>, ... , <act-par-descr$_n$> décrit les valeurs paramétriques réelles de l'appel de fonction. En cas de paramètre de valeur, la description d'un paramètre réel peut être fournie sous la forme d'une expression qui doit être évaluée avant que l'appel puisse être exécuté.

L'on part de l'hypothèse que pour chaque <act-par-desc$_1$> l'identificateur des paramètres formels correspondants <f-par-Id$_1$> est connu, c'est-à-dire que l'on peut étendre la structure syntaxique ci-dessus comme suit:

<function-name>((<f-par-Id$_1$>, <act-par-desc$_1$>), ... , (<f-par-Id$_n$>, <act-par-desc$_n$>))

Le segment de graphe orienté <function-call> dans la Figure B.69 définit l'exécution d'un appel de fonction. L'exécution est structurée en trois étapes. Dans la première étape, un fichier de communication pour la fonction <function-name> est créé. Dans la deuxième étape les valeurs du paramètre réel sont calculées et affectées au champ correspondant dans le fichier de

communication. Dans la troisième étape, la commande du comportement qui appelle la fonction est transférée.

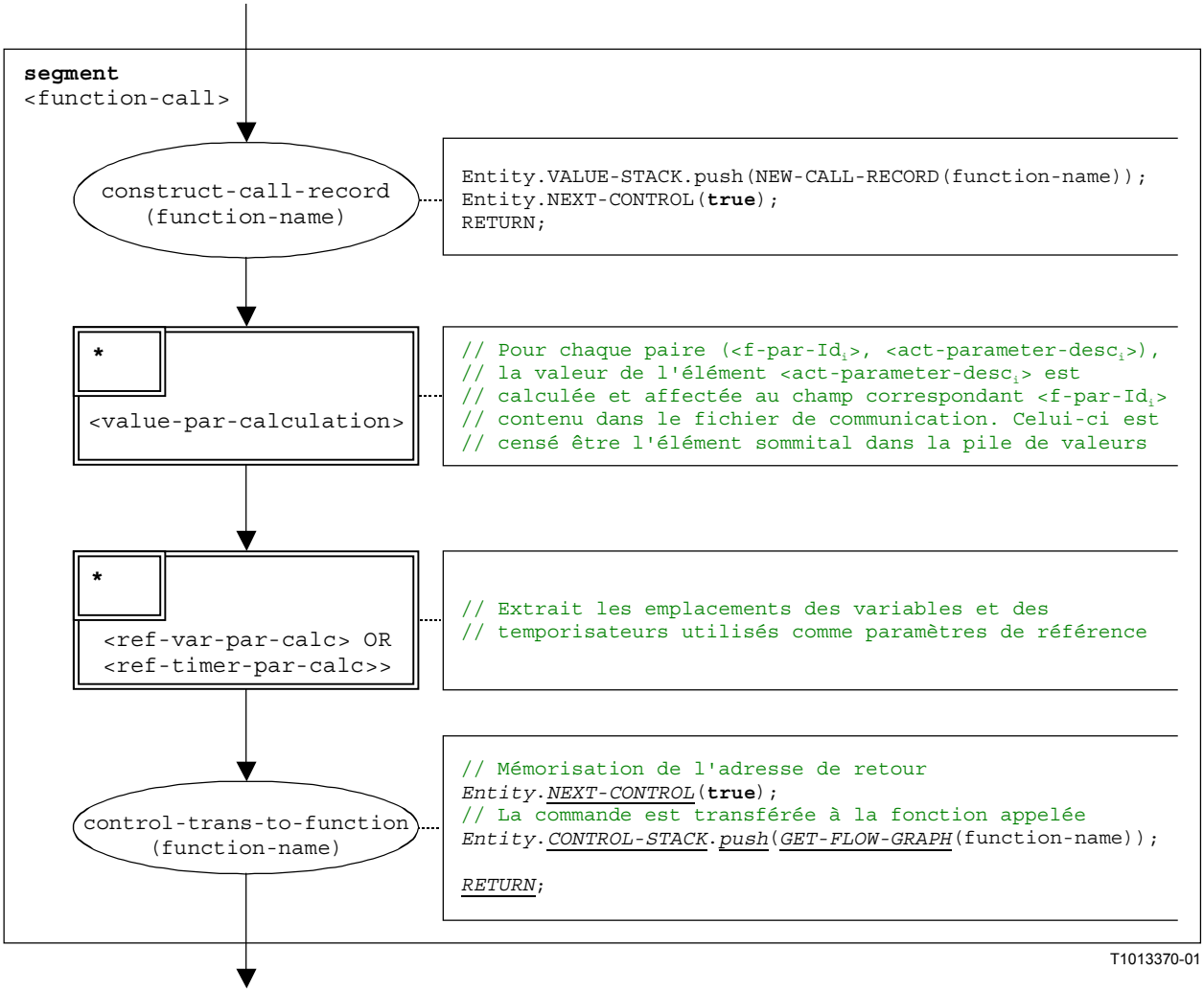


Figure B.69/Z.140 – Segment de graphe orienté <function-call>

B.3.7.23 Segment de graphe orienté <value-par-calculation>

Le segment de graphe orienté <value-par-calculation> est utilisé pour calculer les valeurs paramétriques réelles et pour les affecter aux champs correspondants des fichiers de communication pour fonctions et tests élémentaires.

L'on part de l'hypothèse qu'un fichier de communication est l'élément sommital de la pile de valeurs et qu'une paire de:

(<f-par-Id$_i$>, <act-parameter-desc$_i$>)

doit être traitée. La structure <act-parameter-desc$_i$> doit être évaluée et la structure <f-par-Id$_i$> est l'identificateur d'un paramètre formel qui a un champ correspondant dans le fichier de communication dans la pile de valeurs.

L'exécution du segment de graphe orienté <value-par-calculation> est décrite dans la Figure B.70.

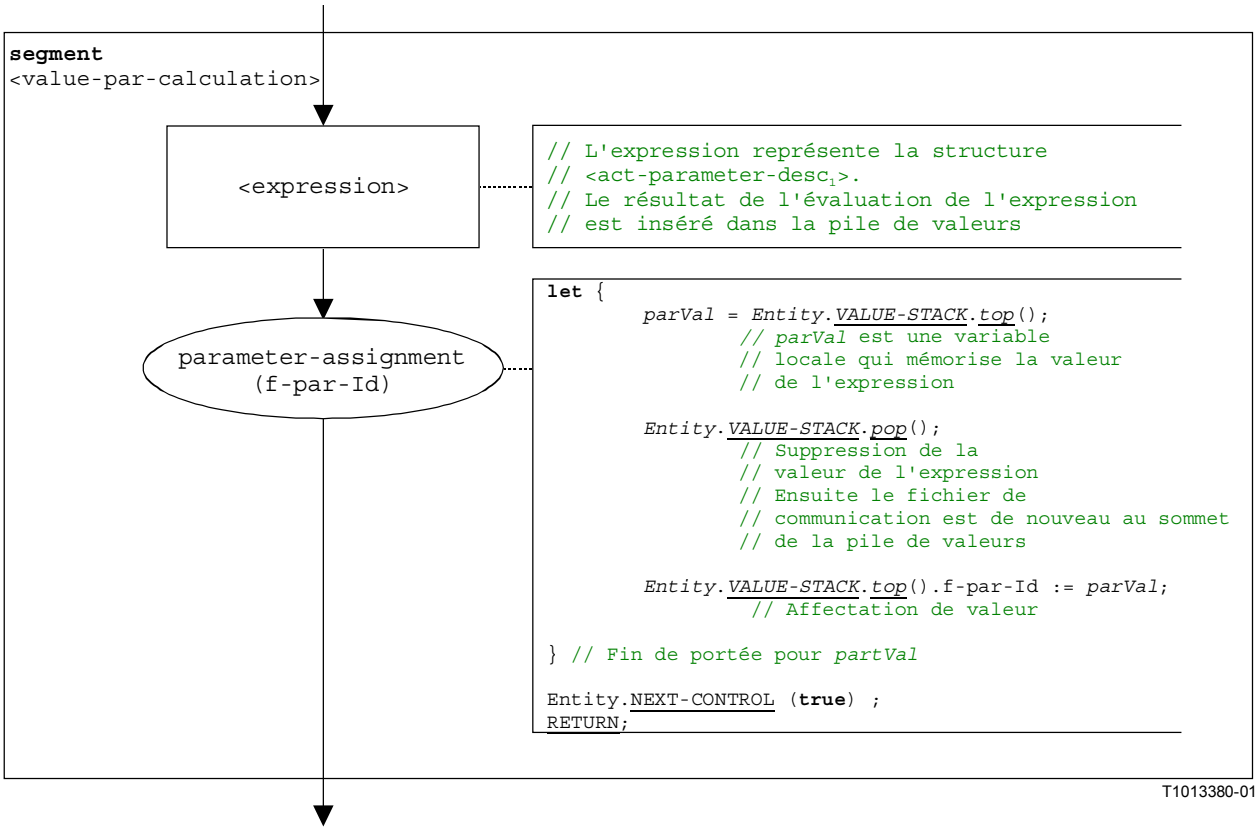


Figure B.70/Z.140 – Segment de graphe orienté <value-par-calculation>

B.3.7.24 Segment de graphe orienté <ref-par-var-calc>

Le segment de graphe orienté <ref-par-var-calc> est utilisé pour extraire les emplacements de variables utilisées en tant que paramètres de référence réels et pour les affecter aux champs correspondants des fichiers de communication pour fonctions et tests élémentaires.

L'on part de l'hypothèse qu'un fichier de communication est l'élément sommital de la pile de valeurs et qu'une paire de:

(<f-par-Id$_i$>, <act-par$_i$>)

doit être traitée. La structure <act-par$_i$> est le paramètre réel pour lequel l'emplacement doit être extrait et <f-par-Id$_i$> est l'identificateur d'un paramètre formel qui a un champ correspondant dans le fichier de communication dans la pile de valeurs.

L'exécution du segment de graphe orienté <ref-par-var-calc> est décrite dans la Figure B.71.

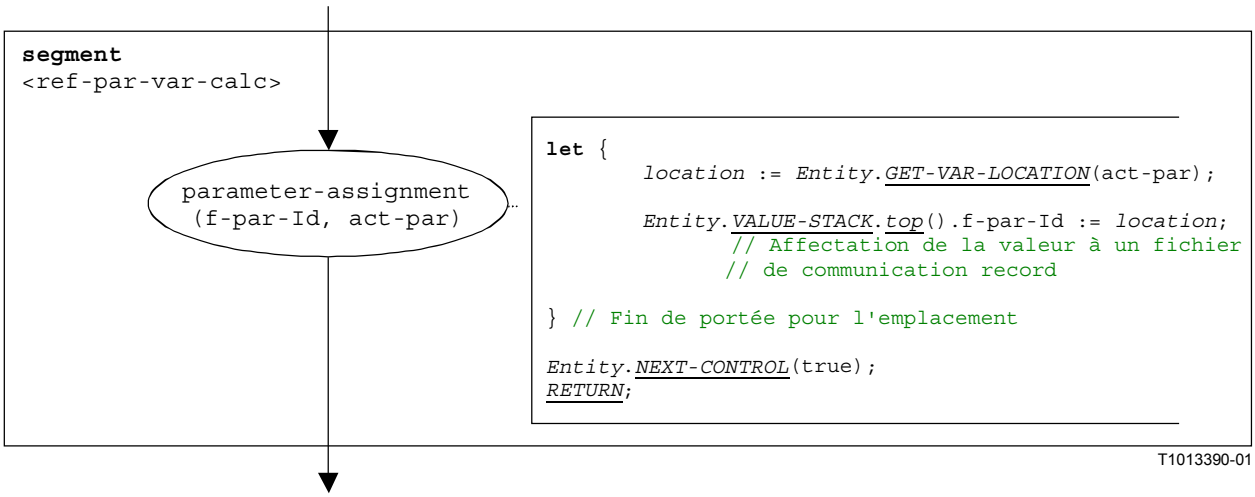


Figure B.71/Z.140 – Segment de graphe orienté <ref-par-var-calc>

B.3.7.25 Segment de graphe orienté <ref-par-timer-calc>

Le segment de graphe orienté <ref-par-timer-calc> est utilisé pour extraire les emplacements de temporisateurs utilisés en tant que paramètres de référence réels et pour les affecter aux champs correspondants du fichier de communication pour fonctions et tests élémentaires.

L'on part de l'hypothèse qu'un fichier de communication est l'élément sommital de la pile de valeurs et qu'une paire de:

(<f-par-Id$_i$>, <act-par$_i$>)

doit être traitée. <act-par$_i$> est le paramètre réel pour lequel l'emplacement doit être extrait et <f-par-Id$_i$> est l'identificateur d'un paramètre formel qui a un champ correspondant dans le fichier de communication de la pile de valeurs.

L'exécution du segment de graphe orienté <ref-par-timer-calc> est décrite dans la Figure B.72.

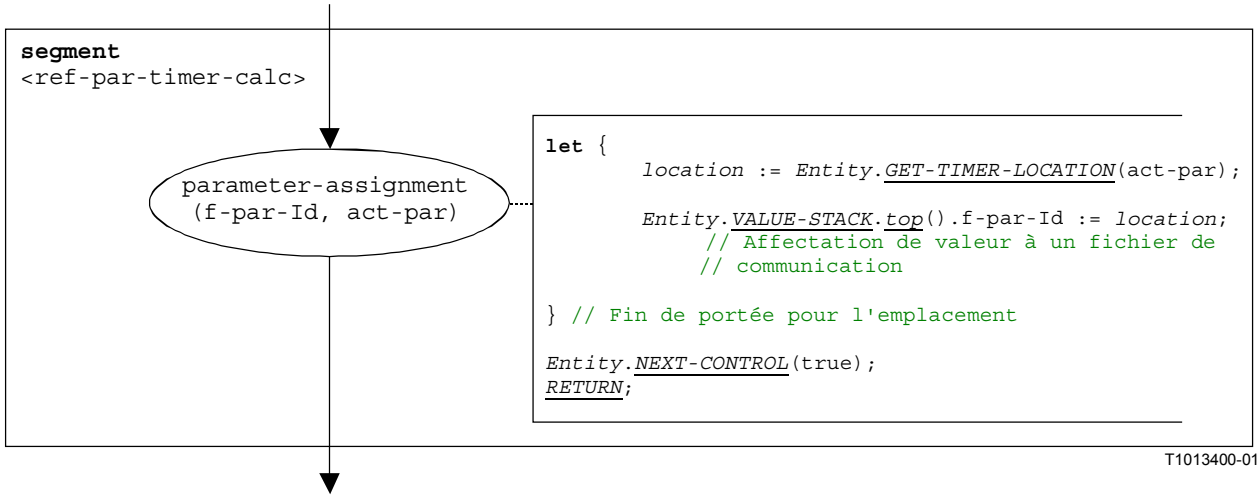


Figure B.72/Z.140 – Segment de graphe orienté <ref-par-timer-calc>

B.3.7.26 Segment de graphe orienté <parameter-handling>

Le segment de graphe orienté <parameter-handling> est utilisé dans au début d'un appel de fonction. Il initialise une nouvelle portée et crée variables et temporisateurs pour le traitement de paramètres. L'on part de l'hypothèse que le fichier de communication de la fonction appelée se trouve au sommet de la pile de valeurs.

L'exécution du segment de graphe orienté <parameter-handling> est décrite dans la Figure B.73.

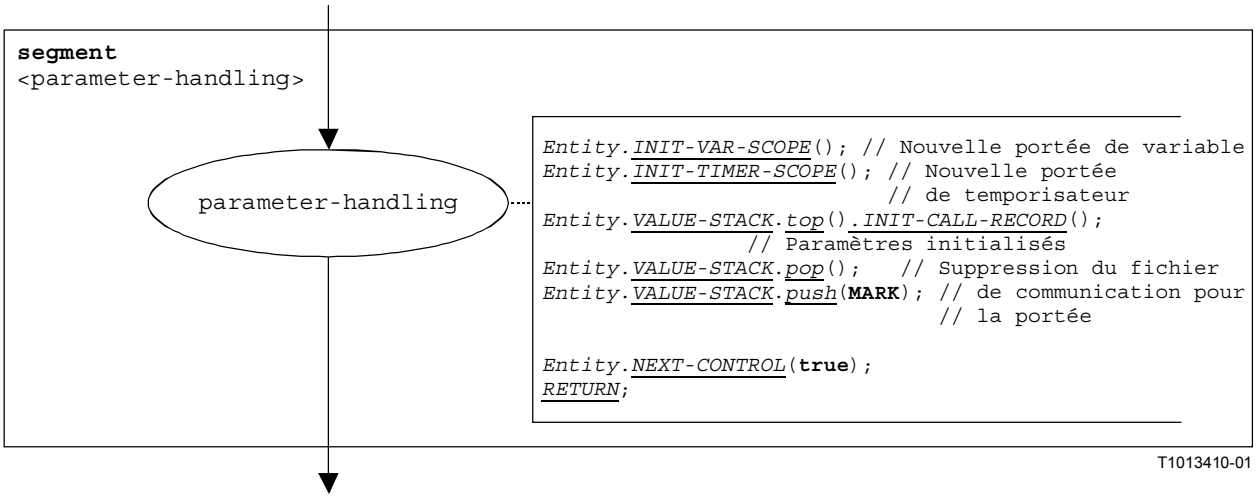


Figure B.73/Z.140 – Segment de graphe orienté <parameter-handling>

B.3.7.27 Opération d'obtention d'appel

La structure syntaxique de l'opération **getcall** est:

```
<portId>.getcall (<matchingSpec>) [from <component_expression>] ->
                                                    [<assignmentPart>]
```

L'expression facultative <component_expression> dans la clause **from** se rapporte à l'expéditeur de l'appel qui est traité par l'opération **getcall**. Elle peut être fournie sous la forme d'une valeur de variable ou de la valeur de retour d'une fonction, c'est-à-dire que l'on part de l'hypothèse qu'il s'agit d'une expression. L'expression facultative <assignmentPart> indique l'affectation d'informations reçues si l'appel reçu s'apparie avec la spécification d'appariement <matchingSpec> et à la clause (facultative) **from**.

Le segment de graphe orienté <getcall-op> dans la Figure B.74 définit l'exécution d'une opération **getcall**.

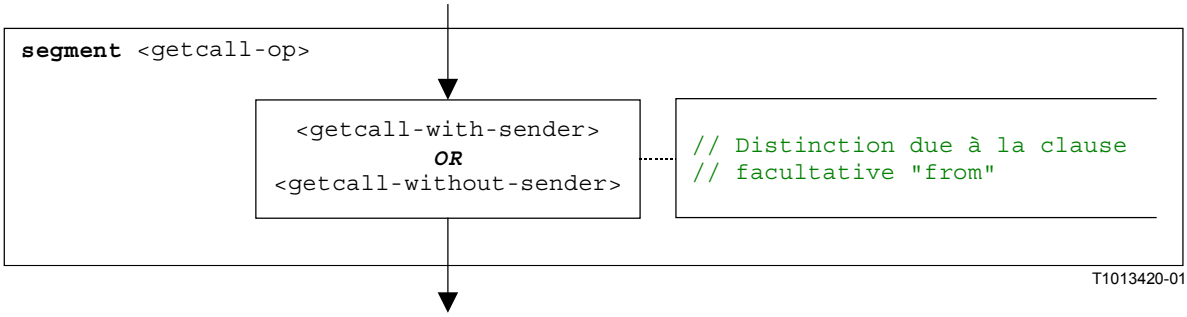


Figure B.74/Z.140 – Segment de graphe orienté <getcall-op>

B.3.7.27.1 Segment de graphe orienté <getcall-with-sender>

Le segment de graphe orienté `<getcall-with-sender>` dans la Figure B.75 définit l'exécution d'une opération **`getcall`** où l'expéditeur est spécifié sous la forme d'une expression.

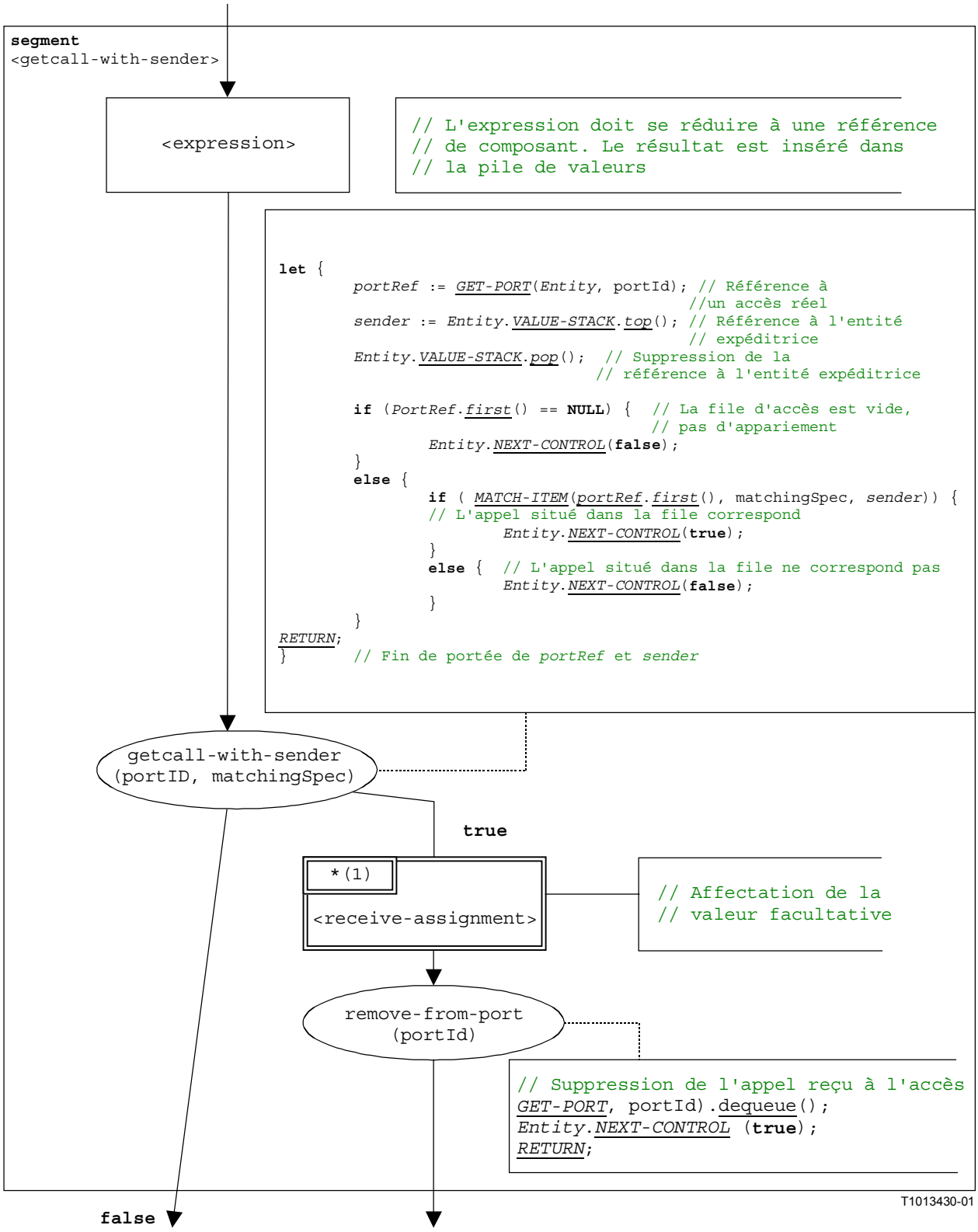


Figure B.75/Z.140 – Segment de graphe orienté <getcall-with-sender>

B.3.7.27.2 Segment de graphe orienté <getcall-without-sender>

Le segment de graphe orienté `<getcall-without-sender>` dans la Figure B.76 définit l'exécution d'une opération **`getcall`** sans clause **`from`**.

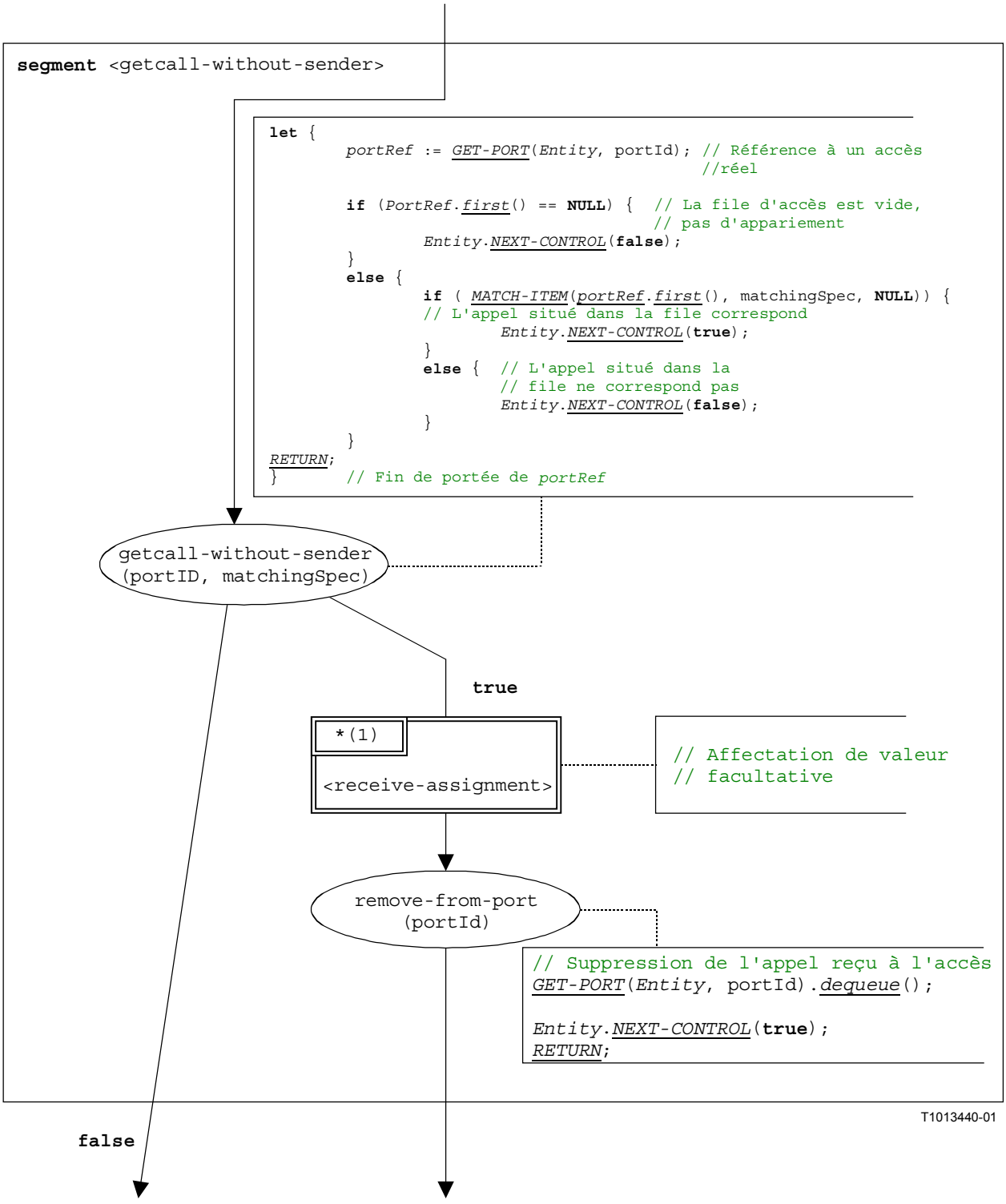


Figure B.76/Z.140 – Segment de graphe orienté <getcall-without-sender>

B.3.7.28 Opération d'obtention de réponse

La structure syntaxique de l'opération **getreply** est:

```
<portId>.getreply (<matchingSpec>) [from <component_expression>] ->
                                                        [<assignmentPart>]
```

L'expression facultative <component_expression> dans la clause **from** se rapporte à l'expéditeur de la réponse qui est traitée par l'opération **getreply**. Elle peut être fournie sous la forme d'une valeur de variable ou de la valeur de retour d'une fonction, c'est-à-dire que l'on part de l'hypothèse qu'il s'agit d'une expression. L'expression facultative <assignmentPart> indique l'affectation des informations reçues si la réponse s'apparie avec la spécification d'appariement <matchingSpec> et avec la clause (facultative) **from**.

Le segment de graphe orienté <getreply-op> dans la Figure B.77 définit l'exécution d'une opération **getreply**.

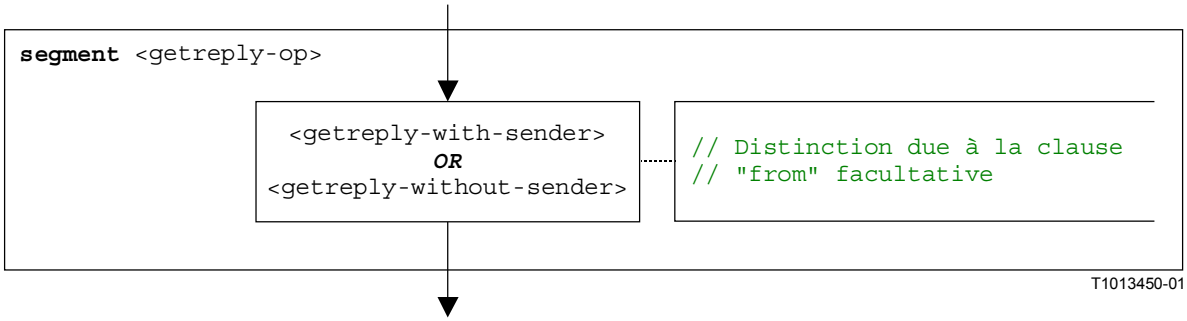


Figure B.77/Z.140 – Segment de graphe orienté <getreply-op>

B.3.7.28.1 Segment de graphe orienté <getreply-with-sender>

Le segment de graphe orienté <getreply-with-sender> dans la Figure B.78 définit l'exécution d'une opération **getreply** où l'expéditeur est spécifié sous la forme d'une expression.

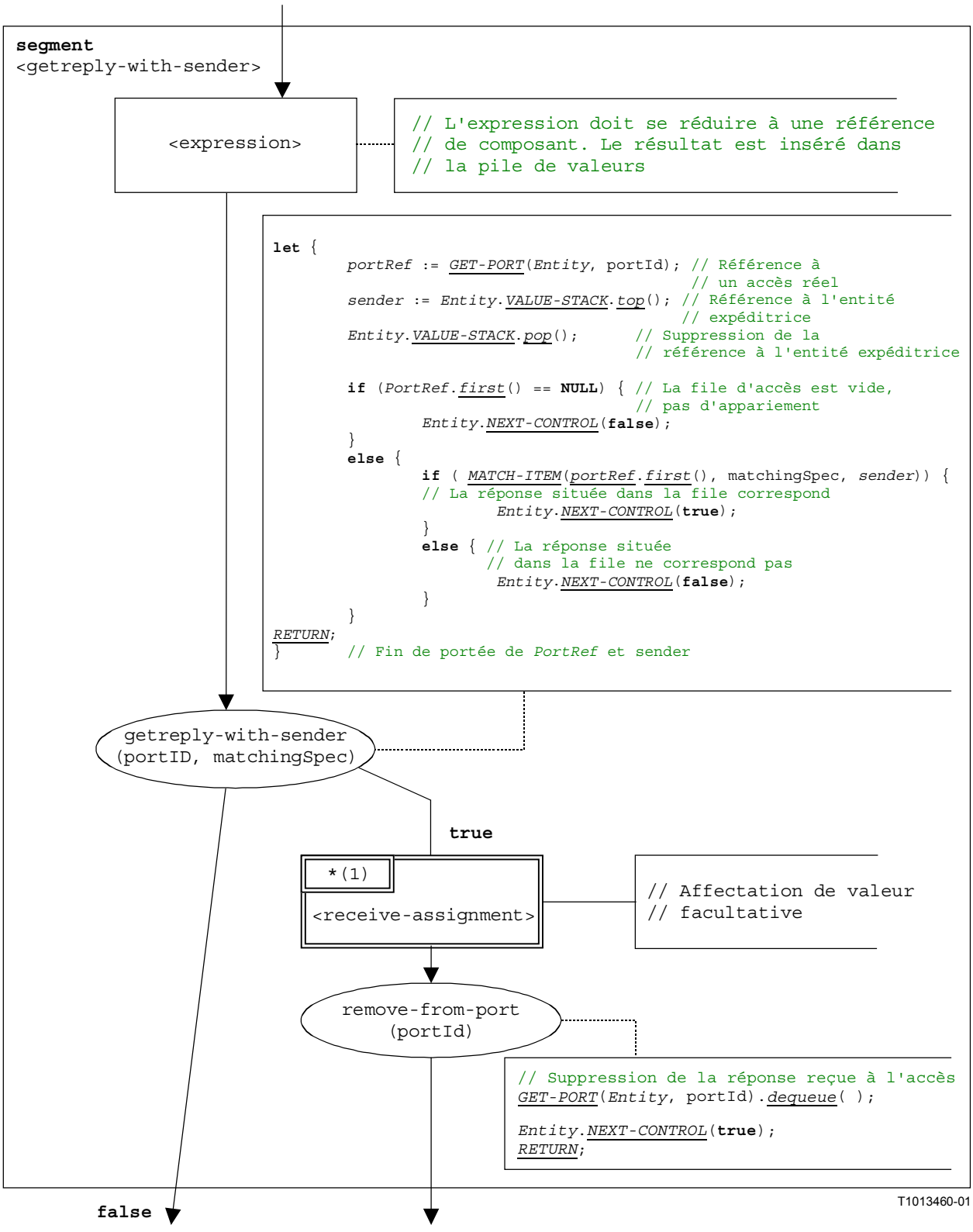


Figure B.78/Z.140 – Segment de graphe orienté <getreply-with-sender>

B.3.7.28.2 Segment de graphe orienté <getreply-without-sender>

Le segment de graphe orienté `<getreply-without-sender>` dans la Figure B.79 définit l'exécution d'un **`getreply`** opération sans clause **`from`**.

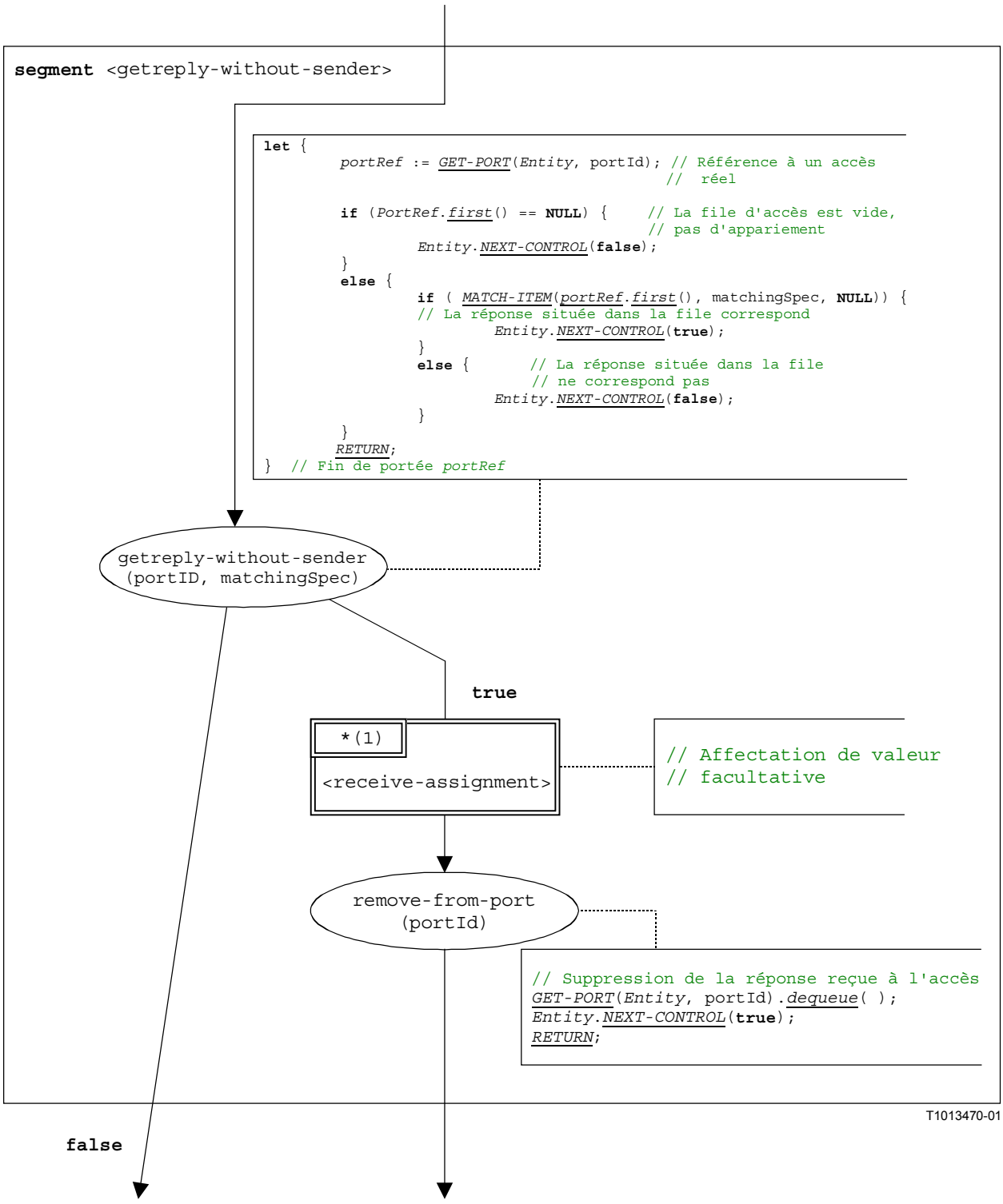


Figure B.79/Z.140 – Segment de graphe orienté <getreply-without-sender>

B.3.7.29 Instruction "aller à" (goto)

La structure syntaxique de l'opération **goto** instruction est:

goto <labelId>

Le segment de graphe orienté <goto-stmt> dans la Figure B.80 définit l'exécution de l'opération **goto**.

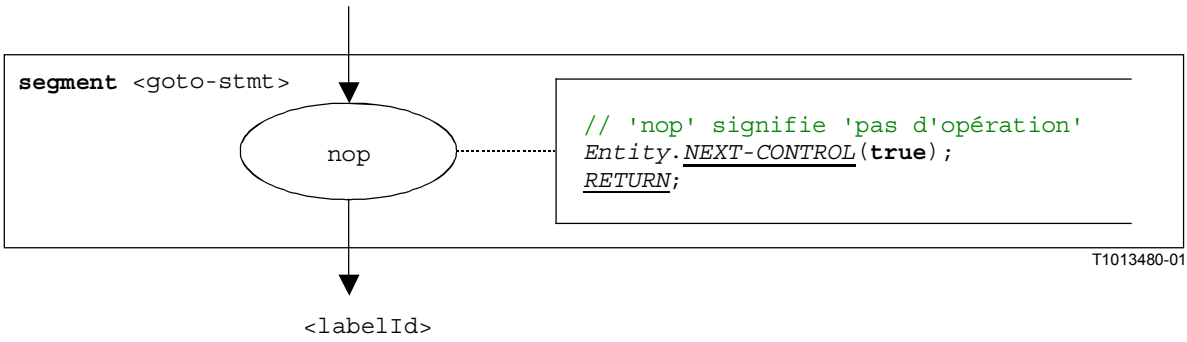


Figure B.80/Z.140 – Segment de graphe orienté <goto-stmt>

NOTE – Le paramètre <labelId> de l'instruction **goto** indique le transfert de la commande vers l'emplacement où l'étiquette <*labelId*> est définie (voir également § B.3.7.31).

B.3.7.30 Instruction "si-sinon" (if-else)

La structure syntaxique de l'instruction **if-else-statement** est:

```
if (<boolean_expression>) <statement-block1>
    [else <statement-block2>]
```

La partie "else" de l'instruction **if-else** est facultative.

Le segment de graphe orienté <if-else-stmt> dans la Figure B.81 définit l'exécution de l'instruction **if-else**.

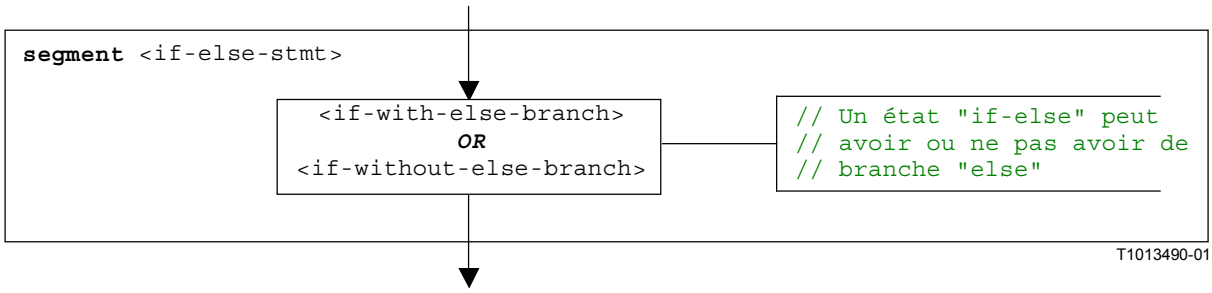


Figure B.81/Z.140 – Segment de graphe orienté <if-else-stmt>

B.3.7.30.1 Segment de graphe orienté <if-with-else-branch>

La Figure B.82 décrit l'exécution d'une instruction **if-else** qui contient une branche "**else**". L'expression <statement-block> contenue dans la branche **true** du nœud décisionnel dans la Figure B.82, correspond à l'expression <statement-block$_1$> dans la structure syntaxique ci-dessus. L'autre expression <statement-block> correspond à l'expression <statement-block$_2$> ci-dessus.

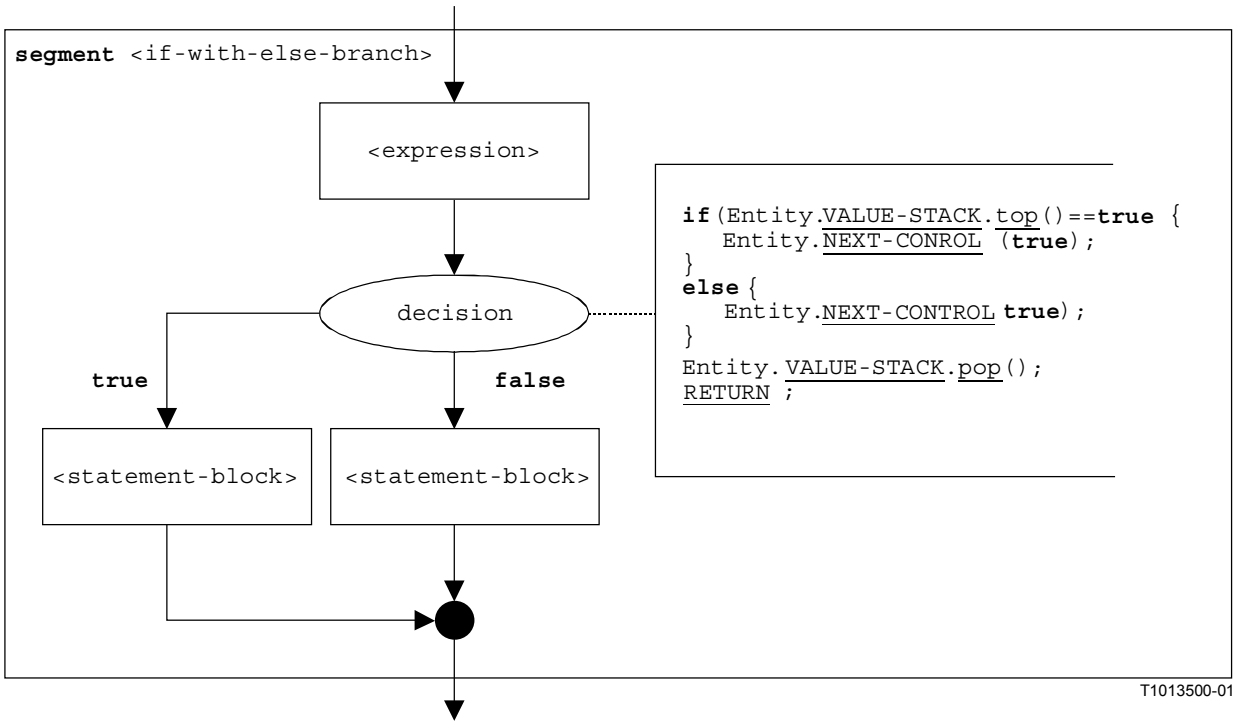


Figure B.82/Z.140 – Segment de graphe orienté <if-with-else-branch>

B.3.7.30.2 Segment de graphe orienté <if-without-else-branch>

La Figure B.83 décrit l'exécution d'une instruction **if-else** qui ne contient aucune branche "**else**". L'expression <statement-block> contenue dans la branche **true** du nœud décisionnel dans la Figure B.82 correspond à l'expression <statement-block$_1$> dans la structure syntaxique ci-dessus.

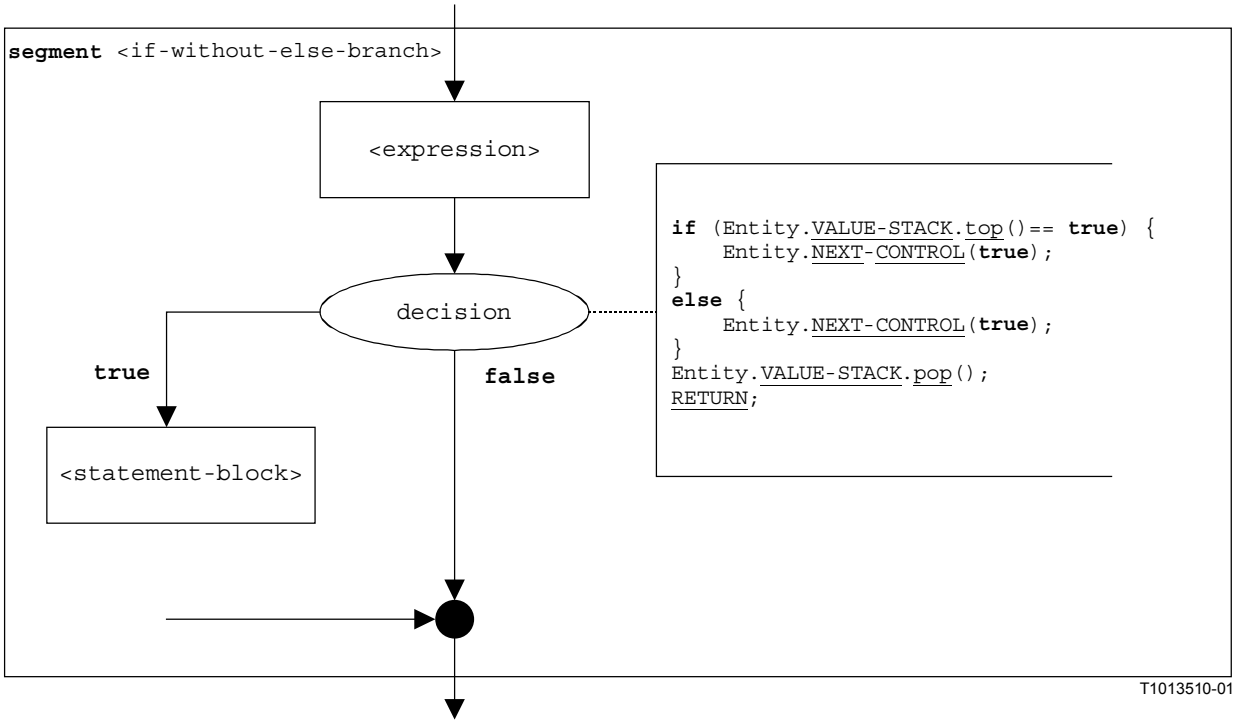


Figure B.83/Z.140 – Segment de graphe orienté <if-without-else-branch>

B.3.7.31 Instruction d'étiquetage

La structure syntaxique de l'instruction **label** est:

```
label <labelId>
```

Le segment de graphe orienté <label-stmt> dans la Figure B.84 définit l'exécution de l'instruction **label**.

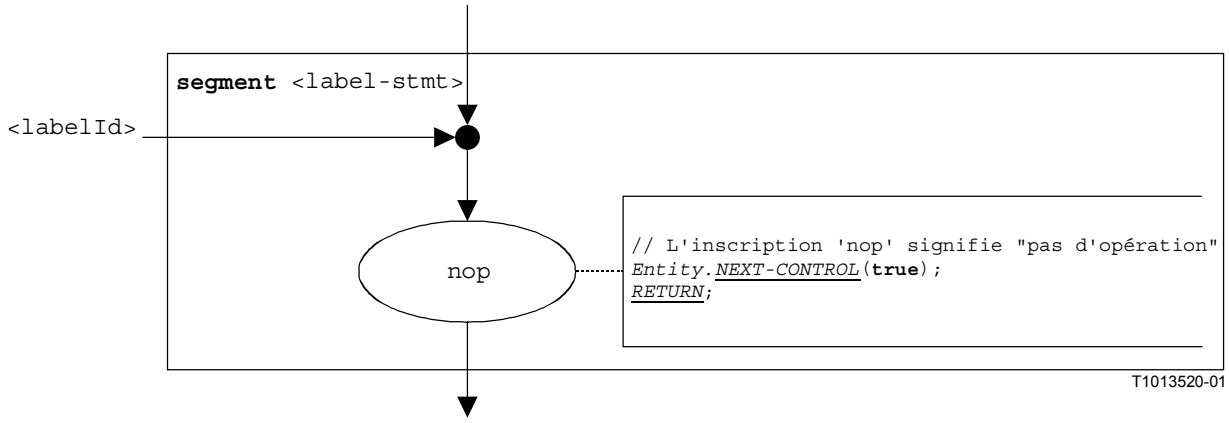


Figure B.84/Z.140 – Segment de graphe orienté <label-stmt>

NOTE – Le paramètre <labelId> de l'instruction **label** indique la possibilité qu'une étiquette puisse être la cible d'un saut au moyen d'une instruction **goto** (voir également § B.3.7.29).

B.3.7.32 Instruction de journalisation

La structure syntaxique de l'opération **log** est:

```
log (<informal-description>)
```

Le segment de graphe orienté <log-stmt> dans la Figure B.85 définit l'exécution de l'opération **log**.

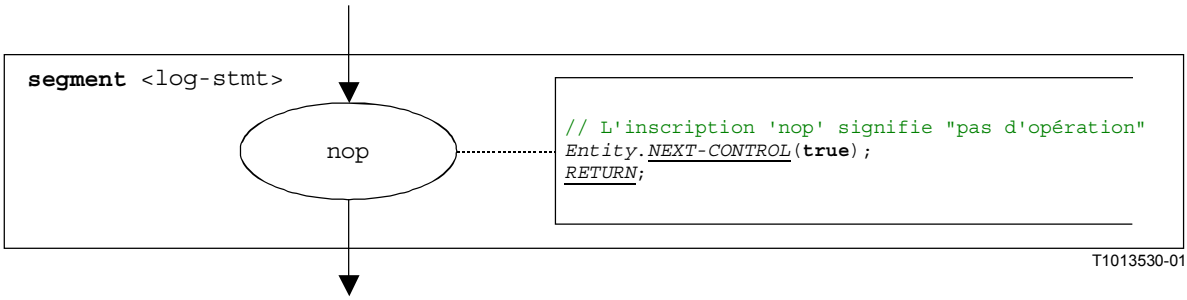


Figure B.85/Z.140 – Segment de graphe orienté <log-stmt>

NOTE – Le paramètre <informal description> de l'instruction **log** n'a pas de signification pour la sémantique opérationnelle et n'est donc pas représenté dans le segment de graphe orienté.

B.3.7.33 Opération de mappage

La structure syntaxique de l'opération **map** est:

```
map(<component_expression>.<portId1>,system.<portId2>)
```

Les identificateurs <portId1> et <portId2> sont considérés comme étant des identificateurs d'accès du composant de test correspondant et de l'interface avec le système de test. Les composants auxquels l'accès <portId1> appartient sont référencés au moyen de la référence de composant <component_expression>. La référence peut être mémorisée dans des variables ou être renvoyée

par une fonction. Par souci de simplicité, elle est considérée comme étant une expression qui se réduit à une référence de composant. Donc, la pile de valeurs est utilisée pour mémoriser la référence de composant.

NOTE – L'opération **map** ne tient pas compte du fait que l'instruction **system**.<portId> apparaît en tant que premier ou en tant que second paramètre. Par souci de simplicité l'on part de l'hypothèse que c'est toujours le second paramètre.

L'exécution de l'opération de mappage est définie par le segment de graphe orienté <map-op> indiqué dans la Figure B.86.

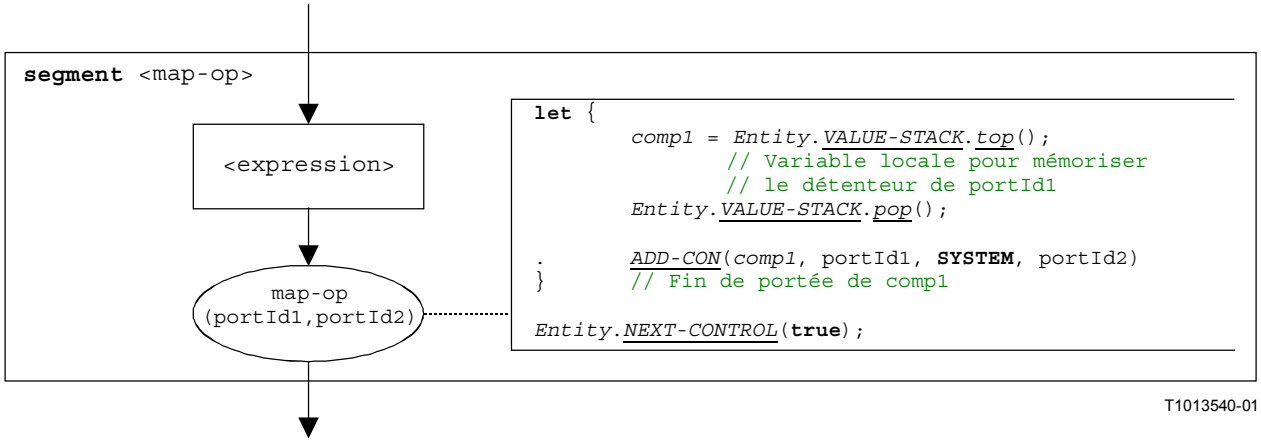


Figure B.86/Z.140 – Segment de graphe orienté <map-op>

B.3.7.34 Opération sur composant de test principal (MTC)

La structure syntaxique de l'opération **mtc** est:

mtc

Le segment de graphe orienté <mtc-op> dans la Figure B.87 définit l'exécution de l'opération **mtc**.

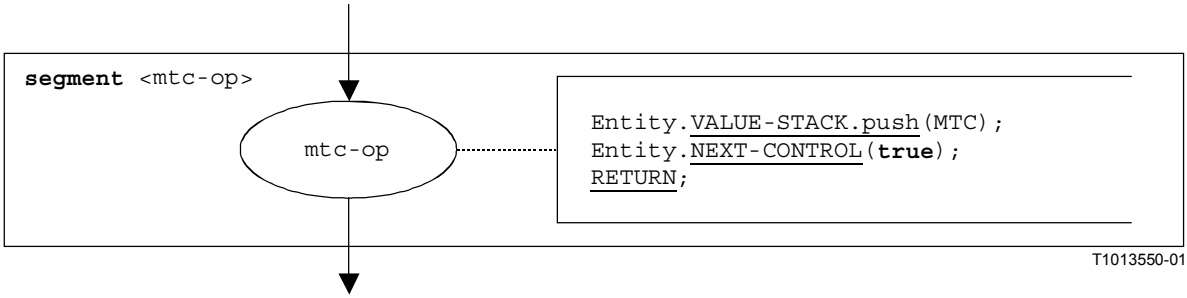


Figure B.87/Z.140 – Segment de graphe orienté <mtc-op>

B.3.7.35 Opération de déclenchement d'exception

La structure syntaxique de l'opération **raise** est:

<portId>.**raise** (<exceptSpec>) [**to** <component_expression>]

L'expression facultative <component_expression> dans la clause "**to**" se rapporte à l'entité réceptrice. Elle peut être fournie sous la forme d'une valeur de variable ou de la valeur de retour d'une fonction.

Le segment de graphe orienté <raise-op> dans la Figure B.88 définit l'exécution d'une opération **raise**.

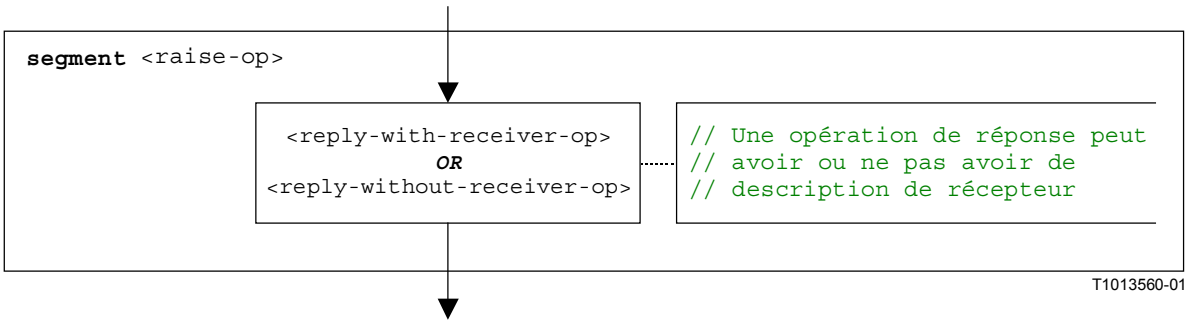


Figure B.88/Z.140 – Segment de graphe orienté <raise-op>

B.3.7.35.1 Segment de graphe orienté <raise-with-receiver-op>

Le segment de graphe orienté `<raise-with-receiver-op>` dans la Figure B.89 définit l'exécution d'une opération **`raise`** où le récepteur est spécifié sous la forme d'une expression.

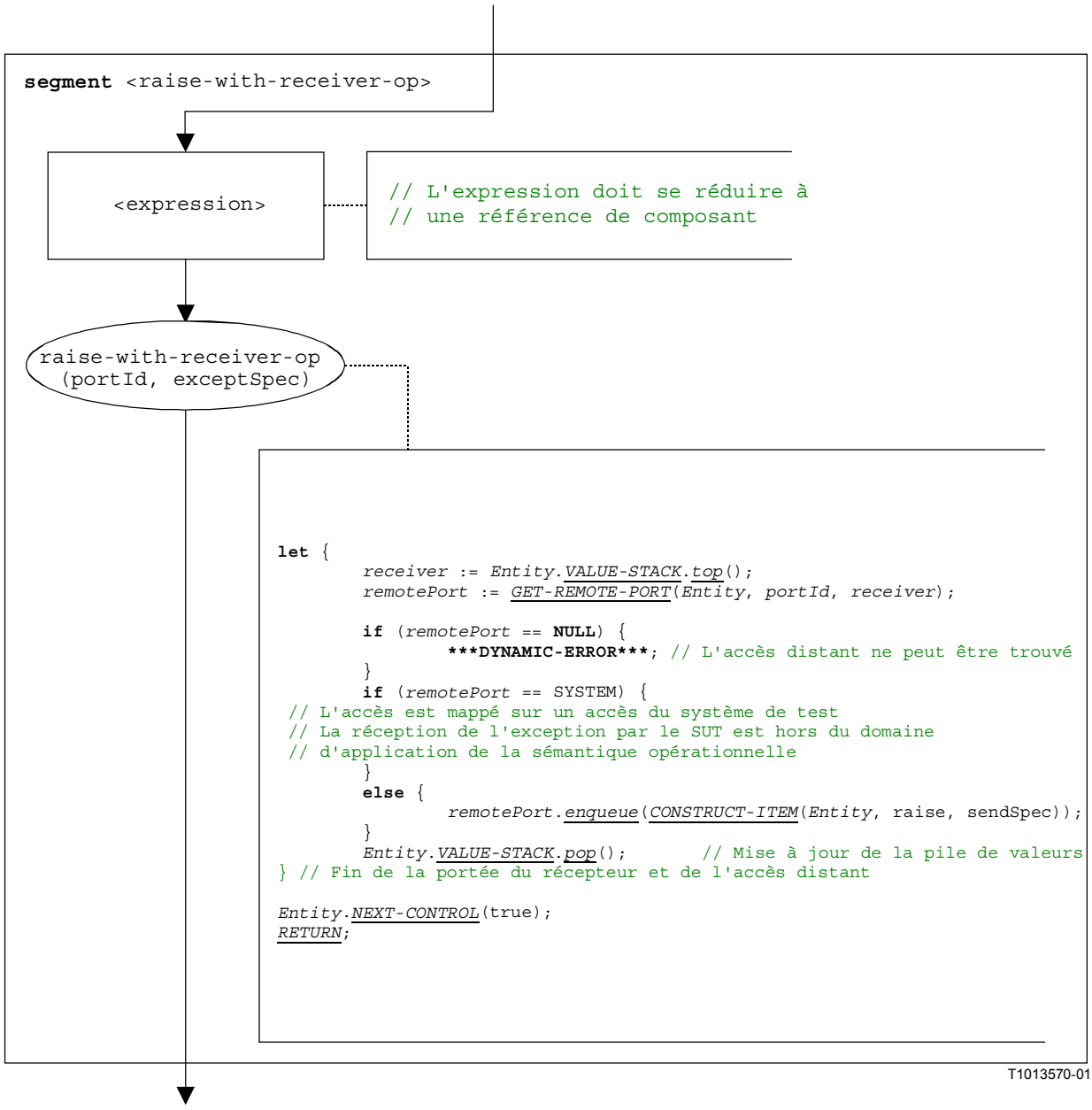


Figure B.89/Z.140 – Segment de graphe orienté <raise-with-receiver-op>

B.3.7.35.2 Segment de graphe orienté <raise-without-receiver-op>

Le segment de graphe orienté <raise-without-receiver-op> dans la Figure B.90 définit l'exécution d'une opération **raise** sans clause **to**.

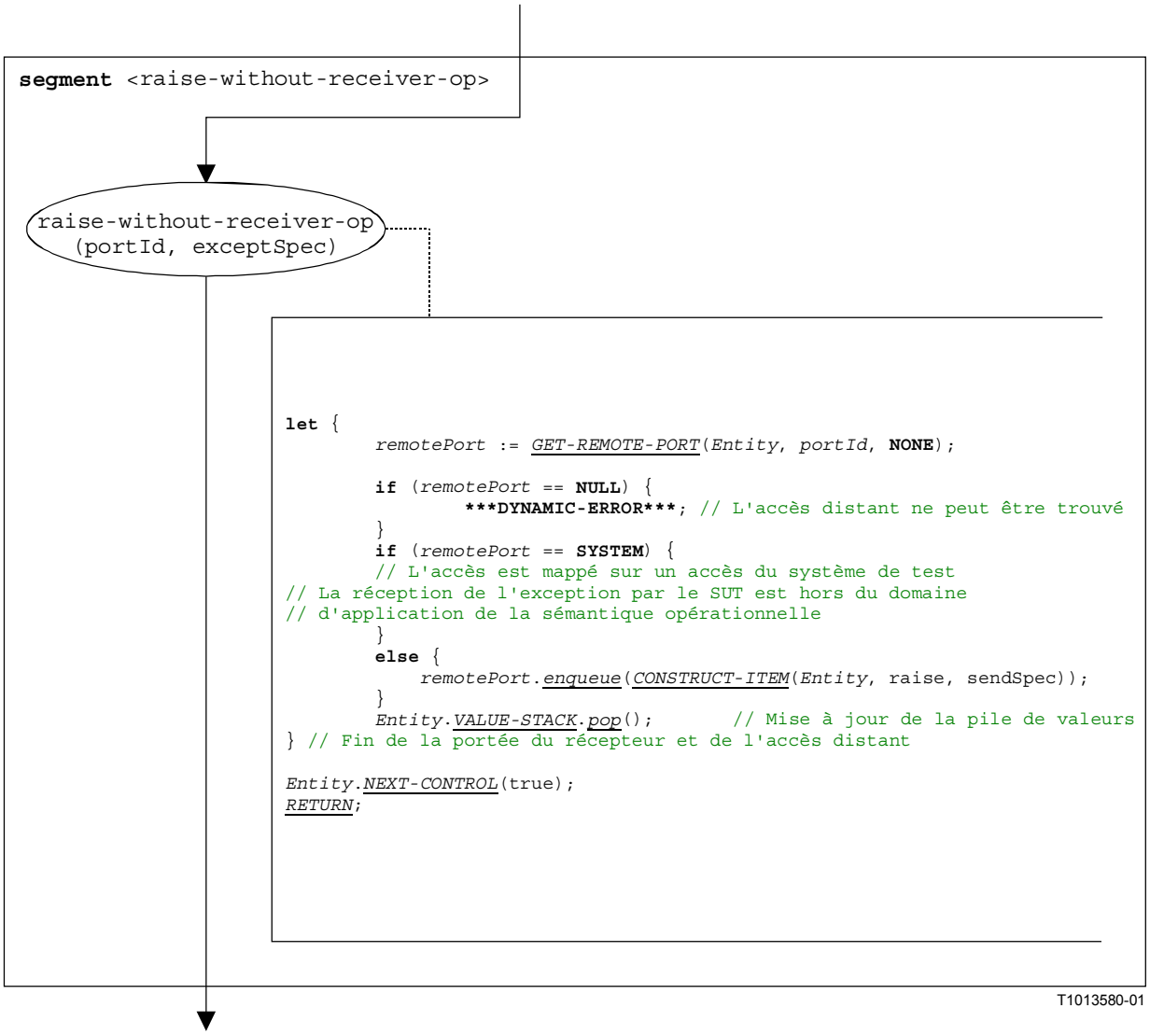


Figure B.90/Z.140 – Segment de graphe orienté <raise-without-receiver-op>

B.3.7.36 Opération de lecture de temporisateur

La structure syntaxique de l'opération "**read** timer" est:

<timerId>**.read**

Le segment de graphe orienté <read-timer-op> dans la Figure B.91 définit l'exécution de l'opération "**read** timer".

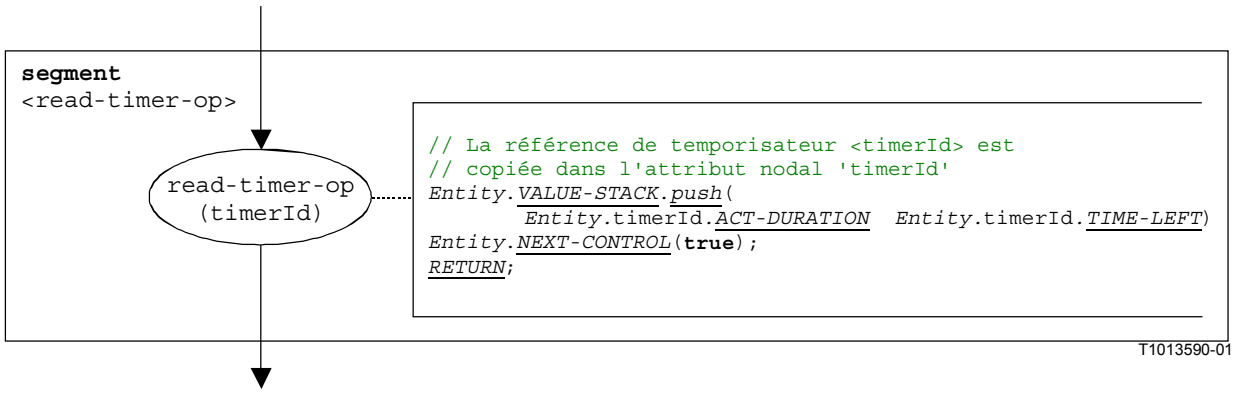


Figure B.91/Z.140 – Segment de graphe orienté <read-timer-op>

B.3.7.37 Opération de réception

La structure syntaxique de l'opération **receive** est:

```
<portId>.receive (<matchingSpec>) [from <component_expression>] ->
                                                        [<assignmentPart>]
```

L'expression facultative <component_expression> dans la clause **from** se rapporte à l'entité expéditrice. Elle peut être fournie sous la forme d'une valeur de variable ou de la valeur de retour d'une fonction, c'est-à-dire que l'on part de l'hypothèse qu'il s'agit d'une expression. L'expression facultative <assignmentPart> indique l'affectation d'informations reçues si le message reçu s'apparie avec la spécification d'appariement <matchingSpec> et avec la clause (facultative) **from**.

Le segment de graphe orienté <receive-op> dans la Figure B.92 définit l'exécution d'une opération **receive**.

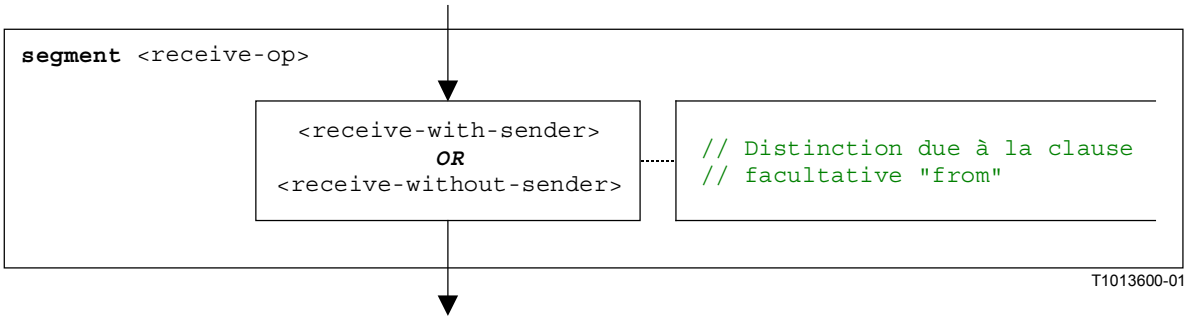


Figure B.92/Z.140 – Segment de graphe orienté <receive-op>

B.3.7.37.1 Segment de graphe orienté <receive-with-sender>

Le segment de graphe orienté <receive-with-sender> dans la Figure B.93 définit l'exécution d'une opération **receive** où l'expéditeur est spécifié sous la forme d'une expression.

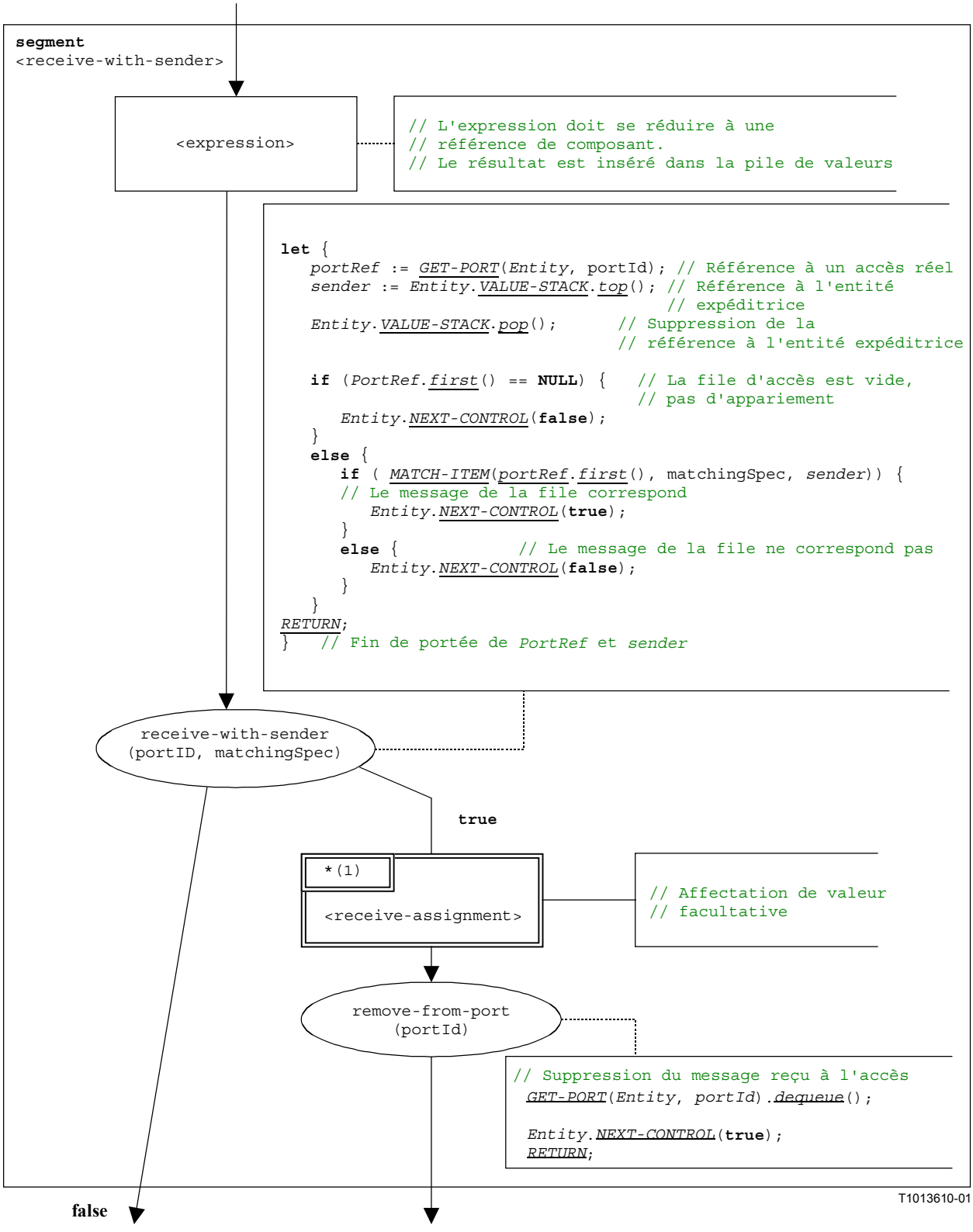


Figure B.93/Z.140 – Segment de graphe orienté <receive-with-sender>

B.3.7.37.2 Segment de graphe orienté <receive-without-sender>

Le segment de graphe orienté `<receive-without-sender>` dans la Figure B.95 définit l'exécution d'une opération **`receive`** sans clause **`from`**.

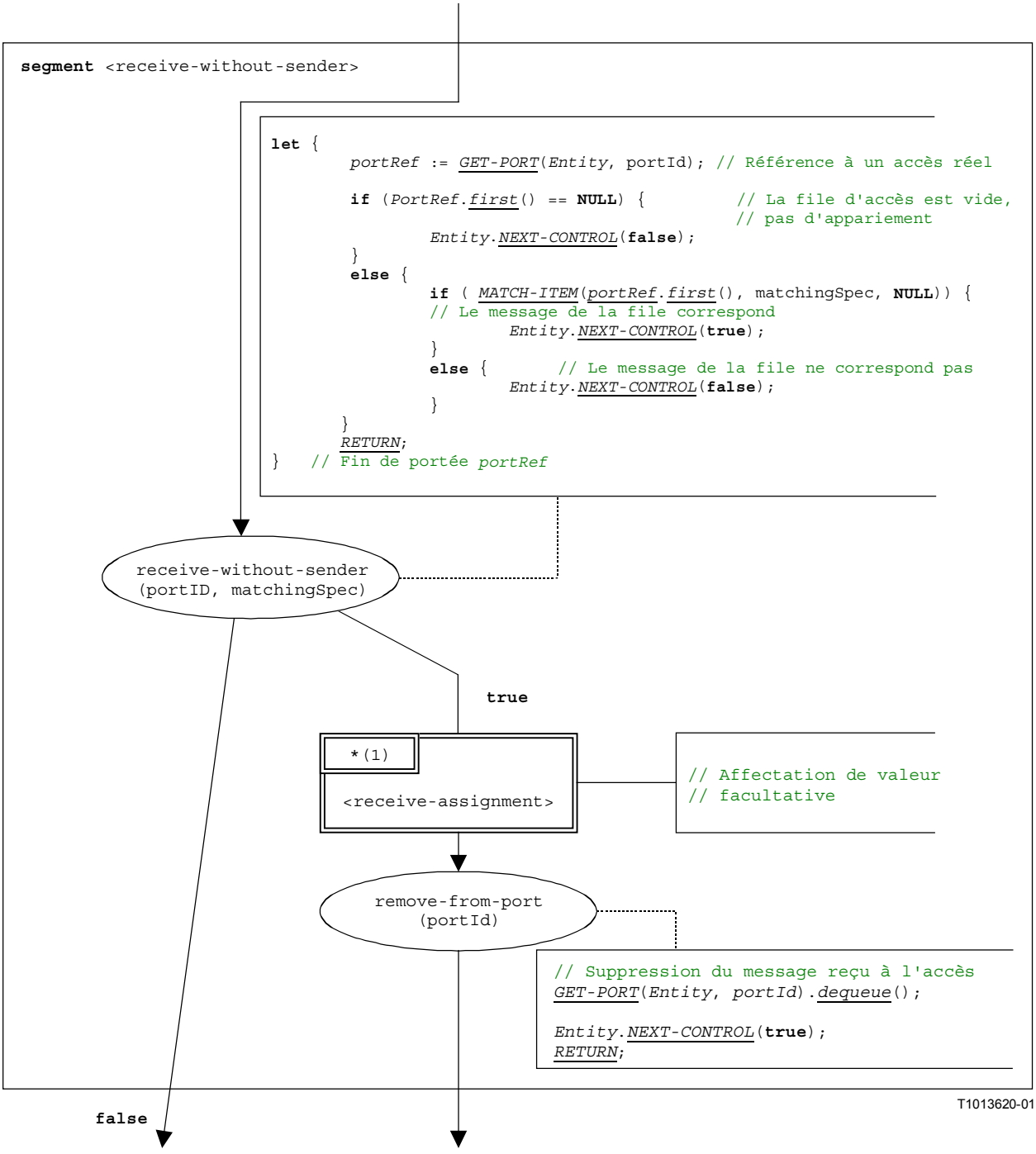


Figure B.94/Z.140 – Segment de graphe orienté <receive-without-sender>

B.3.7.37.3 Segment de graphe orienté <receive-assignment>

Le segment de graphe orienté `<receive-assignment>` dans la Figure B.95 définit l'extraction d'informations des messages reçus et leur affectation à des variables.

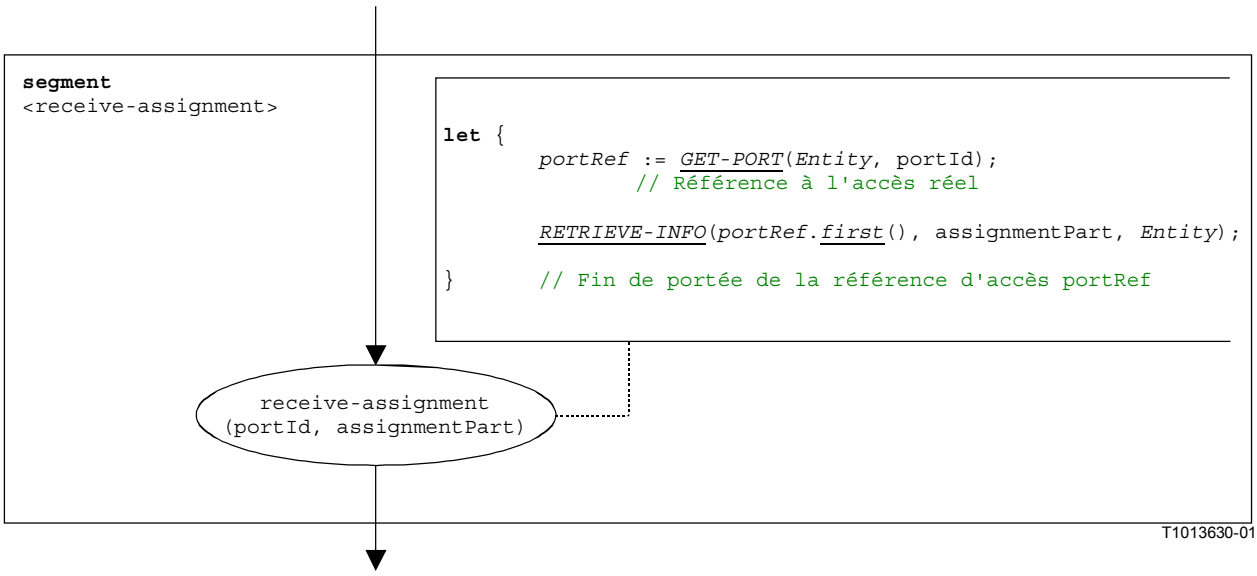


Figure B.95/Z.140 – Segment de graphe orienté <receive-assignment>

B.3.7.38 Opération de réponse

La structure syntaxique de l'opération **reply** est:

<portId>.**reply** (<replySpec>) [**to** <component_expression>]

L'expression facultative <component_expression> dans la clause "**to**" se rapporte à l'entité réceptrice. Elle peut être fournie sous la forme d'une valeur de variable ou de la valeur de retour d'une fonction.

Le segment de graphe orienté <reply-op> dans la Figure B.96 définit l'exécution d'une opération **reply**.

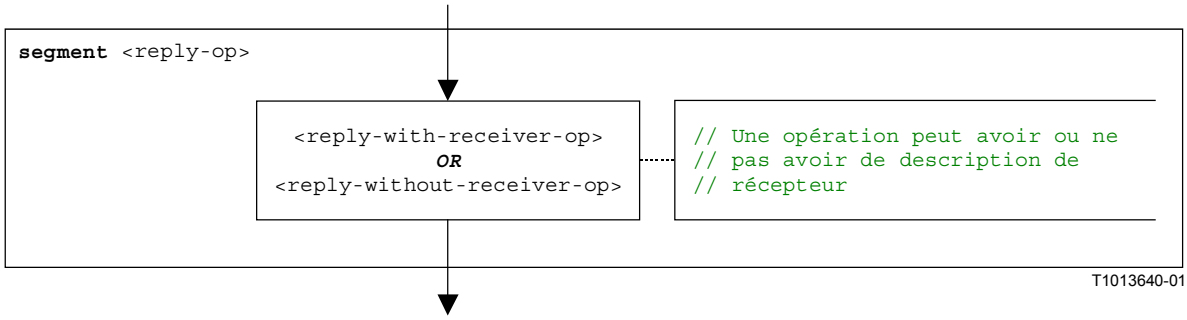


Figure B.96/Z.140 – Segment de graphe orienté <reply-op>

B.3.7.38.1 Segment de graphe orienté <reply-with-receiver-op>

Le segment de graphe orienté <reply-with-receiver-op> dans la Figure B.97 définit l'exécution d'une opération **reply** où le récepteur est spécifié sous la forme d'une expression.

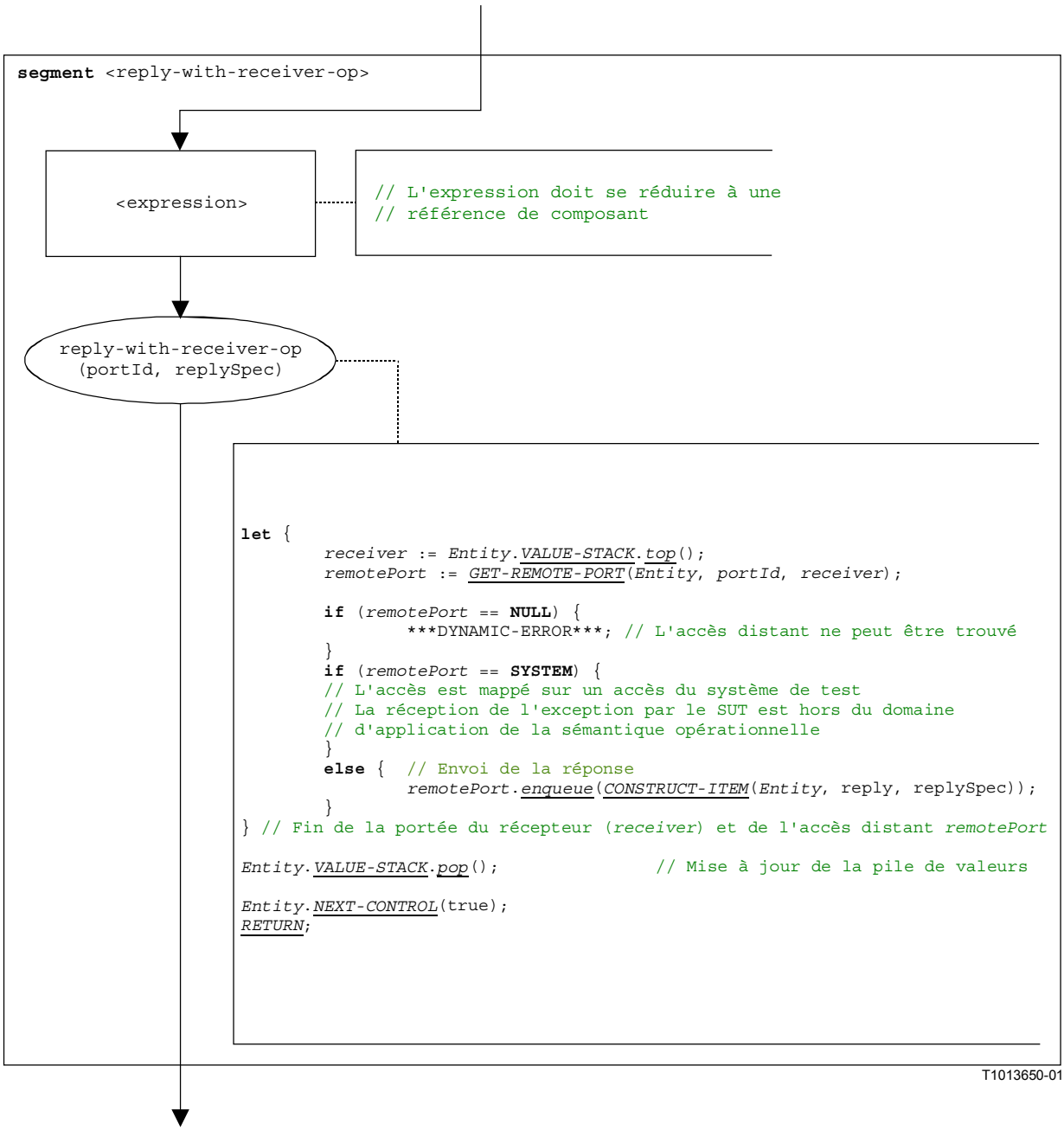


Figure B.97/Z.140 – Segment de graphe orienté <reply-with-receiver-op>

B.3.7.38.2 Segment de graphe orienté <reply-without-receiver-op>

Le segment de graphe orienté `<reply-without-receiver-op>` dans la Figure B.98 définit l'exécution d'une opération reply sans clause **`to`**.

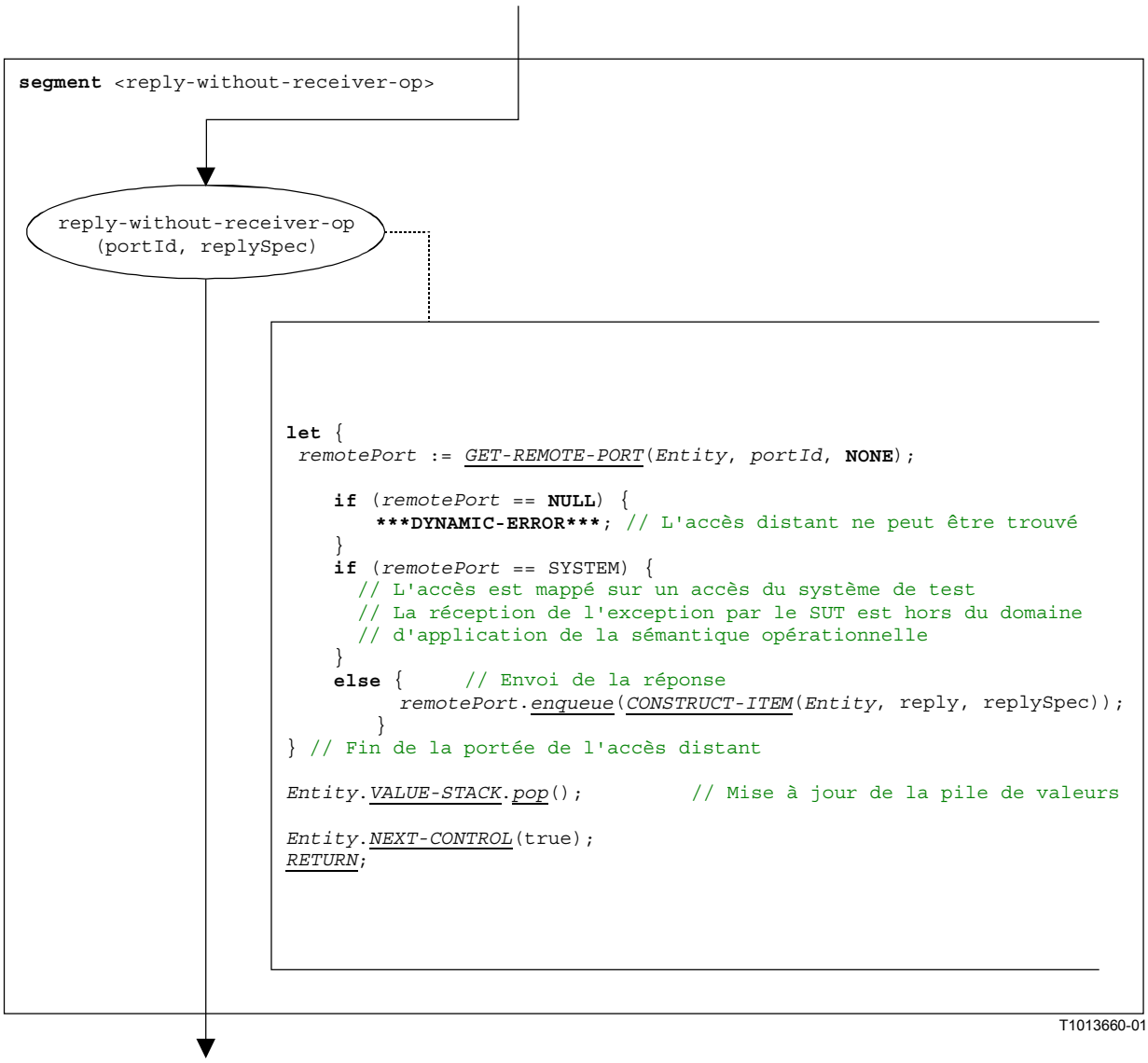


Figure B.98/Z.140 – Segment de graphe orienté <reply-without-receiver-op>

B.3.7.39 Instruction de retour

La structure syntaxique de l'opération return est:

return [<expression>]

L'expression facultative <expression> décrit une valeur possible de retour d'une fonction. L'exécution d'une instruction "**return**" signifie que la commande quitte l'unité de portée réelle, c'est-à-dire que variables et temporisateurs connus seulement dans cette unité de portée doivent être supprimés et que la pile de valeurs doit être mise à jour. Une instruction **return** possède l'effet d'une opération **stop**, si elle est la dernière instruction dans une description de comportement.

NOTE – En raison du remplacement de notations abrégées, les tests élémentaires et la commande de module se termineront toujours par une opération **stop**. Seuls d'autres composants de test peuvent se terminer par une opération **return**.

Le segment de graphe orienté <return-stmt> dans la Figure B.99 définit l'exécution d'une opération **return**.

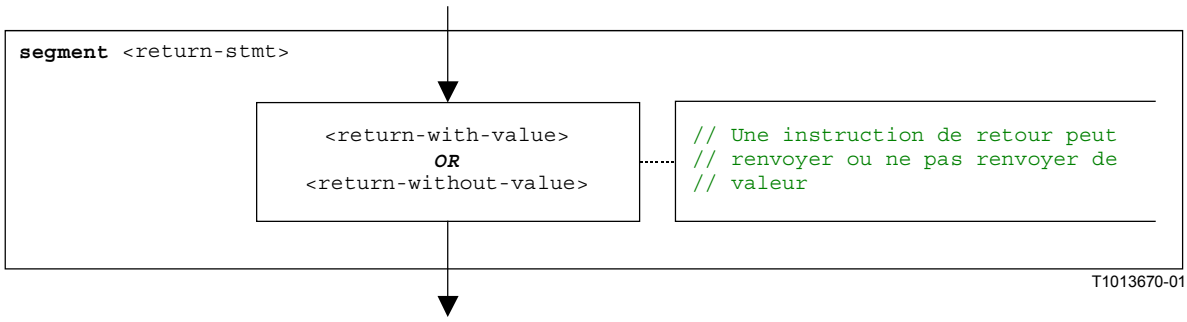


Figure B.99/Z.140 – Segment de graphe orienté <return-stmt>

B.3.7.39.1 Segment de graphe orienté <return-with-value>

Le segment de graphe orienté `<return-with-value>` dans la Figure B.100 définit l'exécution d'une opération **`return`** qui renvoie une valeur spécifiée sous la forme d'une expression.

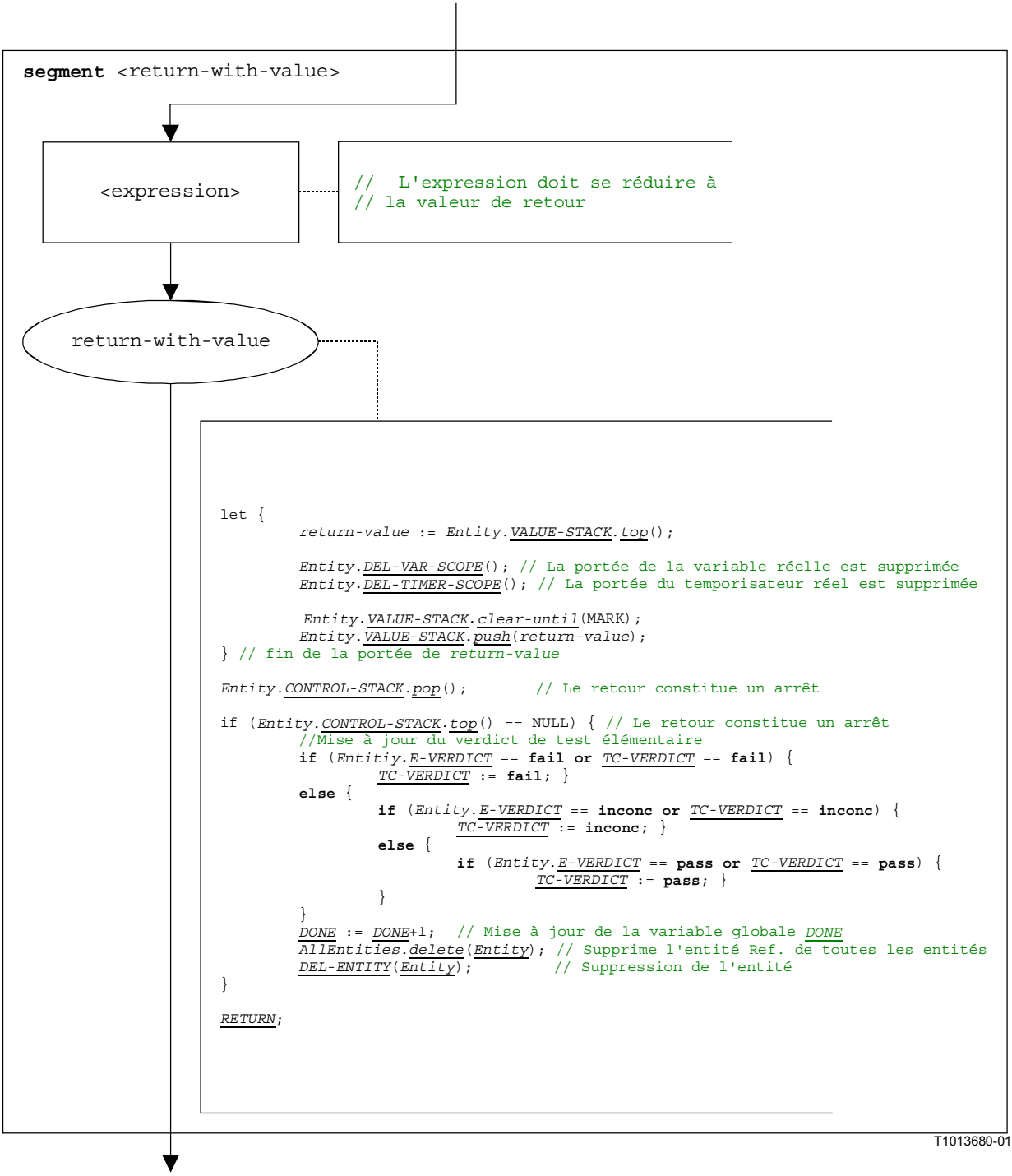


Figure B.100/Z.140 – Segment de graphe orienté <return-with-value>

B.3.7.39.2 Segment de graphe orienté <return-without-value>

Le segment de graphe orienté `<return-without-value>` dans la Figure B.101 définit l'exécution d'une instruction **`return`** qui ne renvoie aucune valeur.

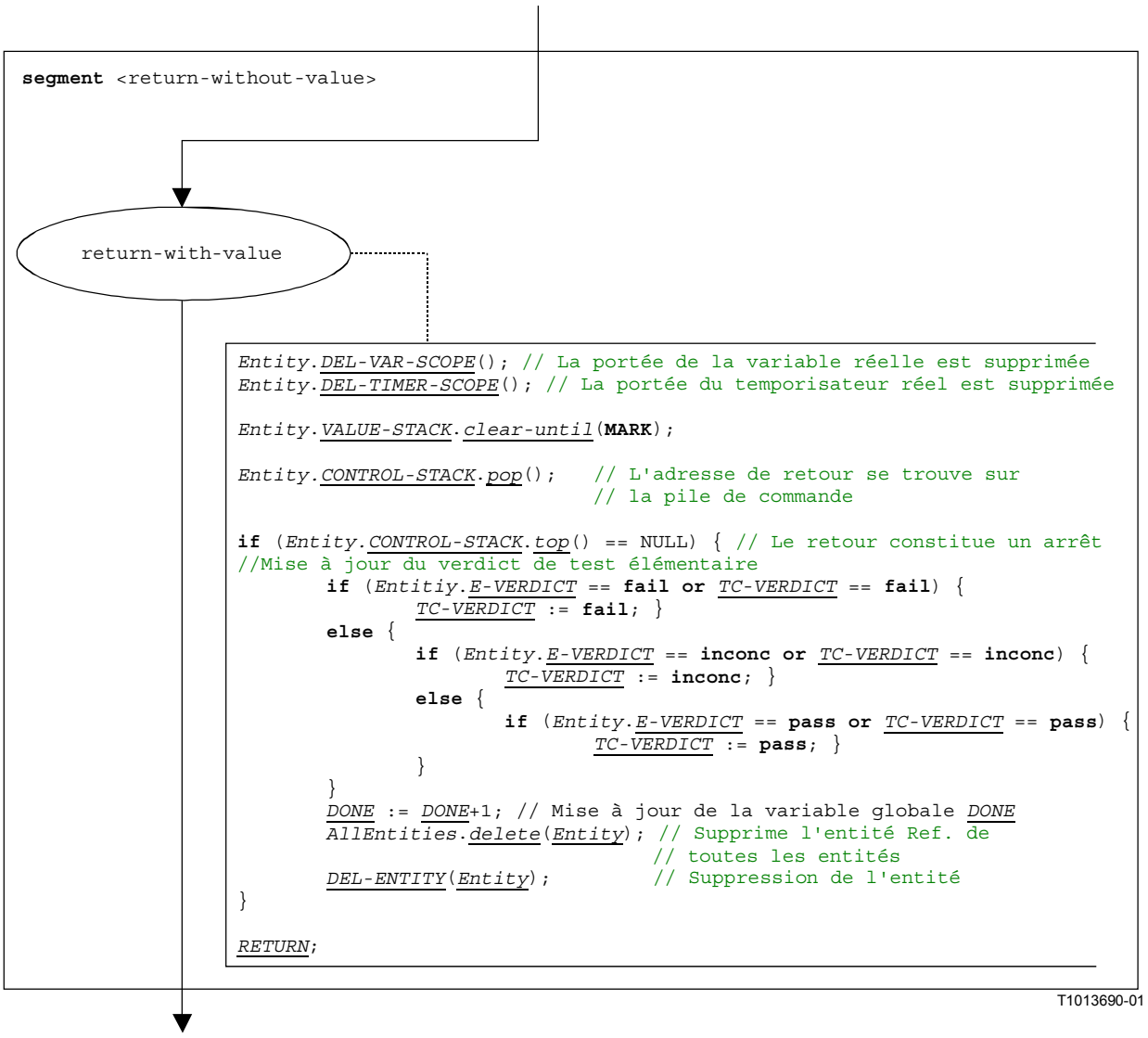


Figure B.101/Z.140 – Segment de graphe orienté <return-without-value>

B.3.7.40 Opération d'exécution de tous les composants

L'opération **running-all-components** se rapporte à l'usage du mot clé **all components** contenu dans **running** component (§ 42). L'opération **running-all-components** ne peut être appelée que par le composant **mtc**. Elle permet de vérifier si tous les composants de test parallèles d'un test élémentaire sont en cours d'exécution. La structure syntaxique de l'opération **running-all-components** est:

```
all component.running
```

L'exécution de l'opération **running-all-components** est définie par le segment de graphe orienté <running-all-comp-op> dans la Figure B.102.

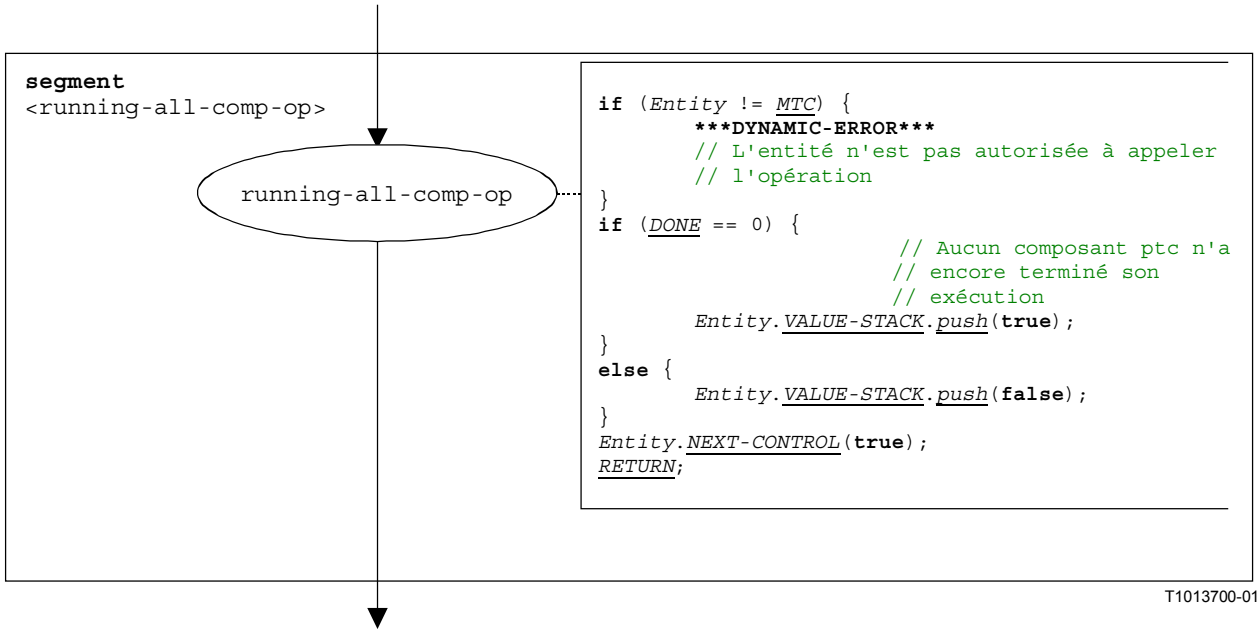


Figure B.102/Z.140 – Segment de graphe orienté <running-all-comp-op>

B.3.7.41 Opération d'exécution d'un composant quelconque

L'opération **running-any-component** se rapporte à l'usage du mot clé **any component** dans l'opération "**running** component" (§ 42). L'opération **running-any-component** ne peut être appelée que par le composant **mtc**. Elle permet de vérifier si au moins un composant de test parallèle d'un test élémentaire est encore en cours d'exécution. La structure syntaxique de l'opération **running-any-components** est:

any component.running

L'exécution de l'opération **running-any-component** est définie par le segment de graphe orienté <running-any-comp-op> dans la Figure B.103.

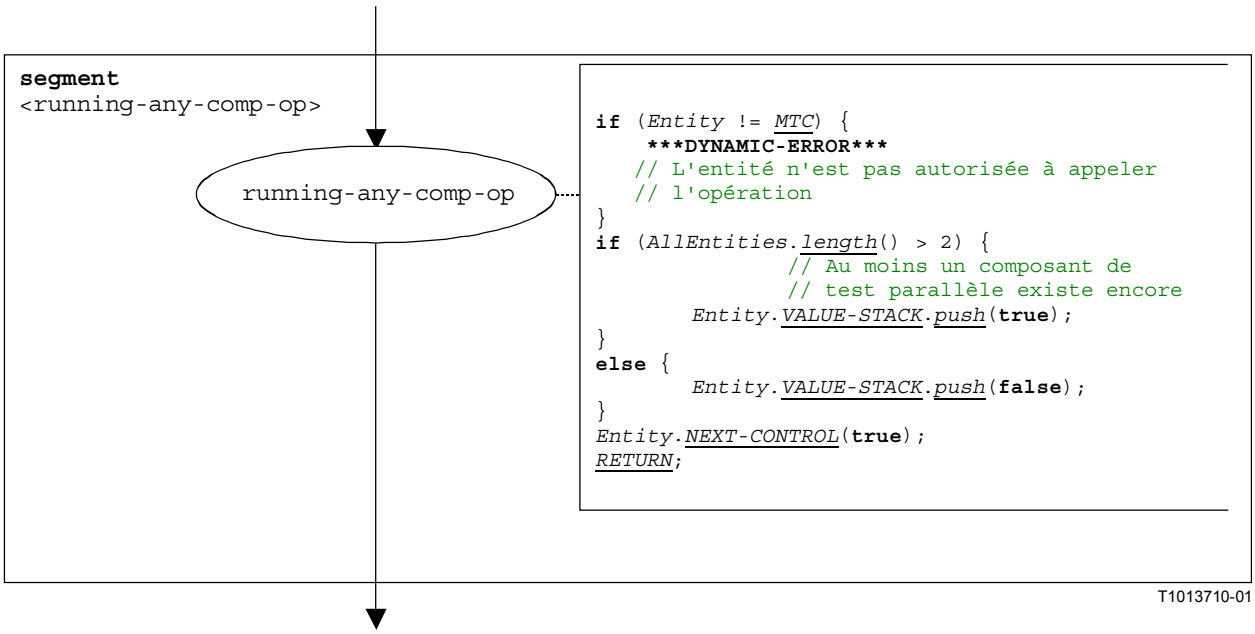


Figure B.103/Z.140 – Segment de graphe orienté <running-any-comp-op>

B.3.7.42 Opération d'exécution de composant

La structure syntaxique de l'opération **running** component est:

<component_expression>.**running**

L'opération "**running** component" vérifie si un composant est en cours ou s'est arrêté. L'utilisation d'une référence de composant identifie le composant à vérifier. La référence peut être mémorisée dans une variable ou être renvoyée par une fonction. Par souci de simplicité, cela est considéré comme étant une expression qui se réduit à une référence de composant.

Le segment de graphe orienté <running-component-op> dans la Figure B.104 définit l'exécution de l'opération "**running** component".

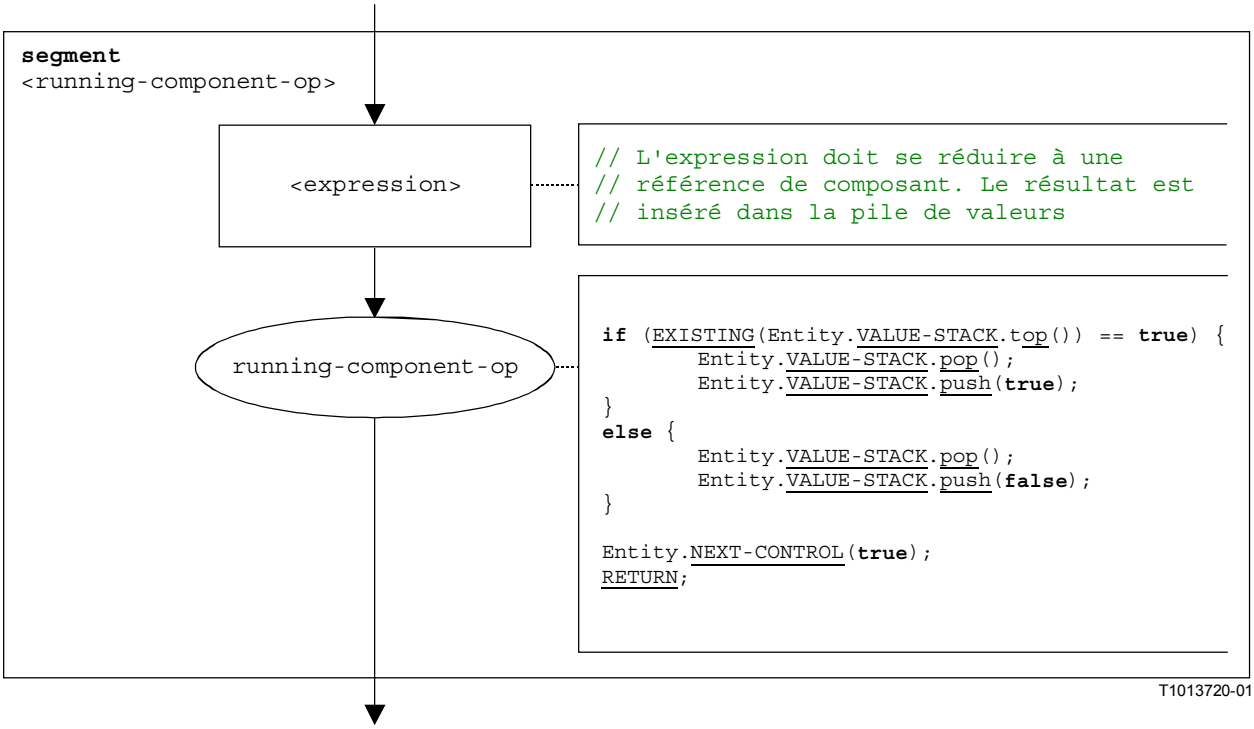


Figure B.104/Z.140 – Segment de graphe orienté <running-component-op>

B.3.7.43 Opération d'exécution de temporisation

La structure syntaxique de l'opération de temporisation active est:

<timerId>.**running**

Le segment de graphe orienté <running-timer-op> dans la Figure B.105 définit l'exécution de l'opération de temporisation active.

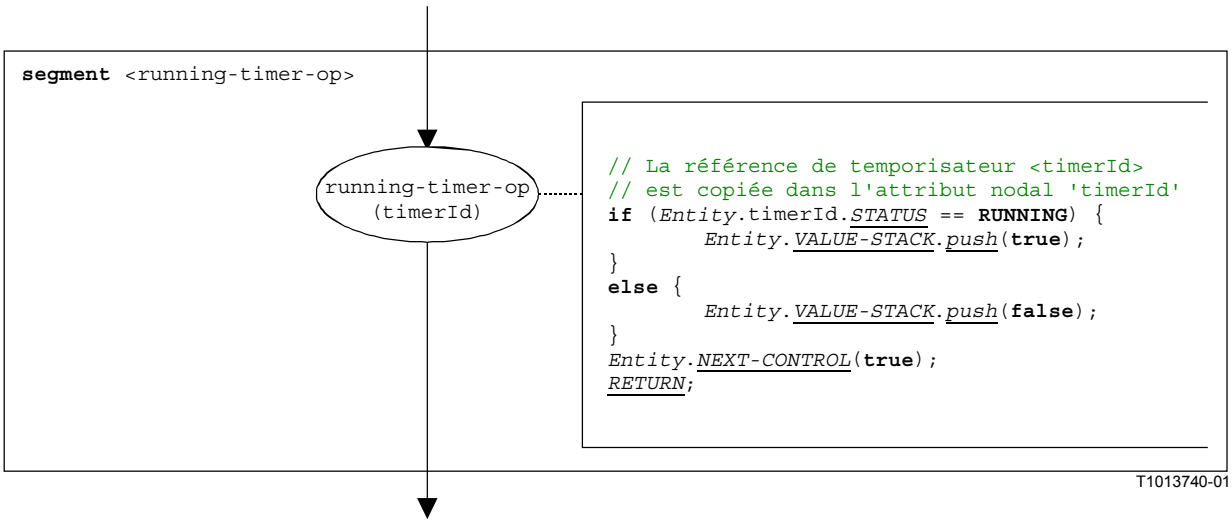


Figure B.105/Z.140 – Segment de graphe orienté <running-timer-op>

B.3.7.44 Opération d'envoi

La structure syntaxique de l'opération d'envoi est:

<portId>.**send** (<send-spec>) [**to** <component_expression>]

L'expression facultative <component_expression> dans la clause "**to**" se rapporte à l'entité réceptrice. Elle peut être fournie sous la forme d'une valeur de variable ou de la valeur de retour d'une fonction.

Le segment de graphe orienté <send-op> dans la Figure B.106 définit l'exécution d'une opération **send**.

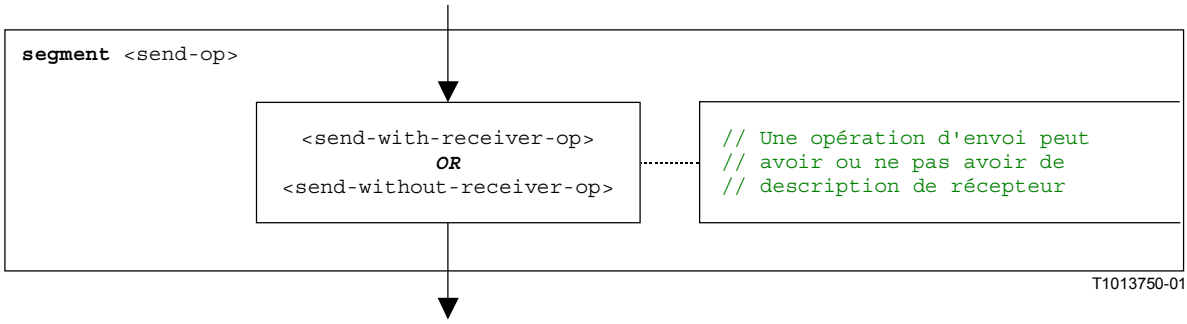


Figure B.106/Z.140 – Segment de graphe orienté <send-op>

B.3.7.44.1 Segment de graphe orienté <sent-with-receiver-op>

Le segment de graphe orienté <send-with-receiver-op> dans la Figure B.107 définit l'exécution d'une opération **send** où le récepteur est spécifié sous la forme d'une expression.

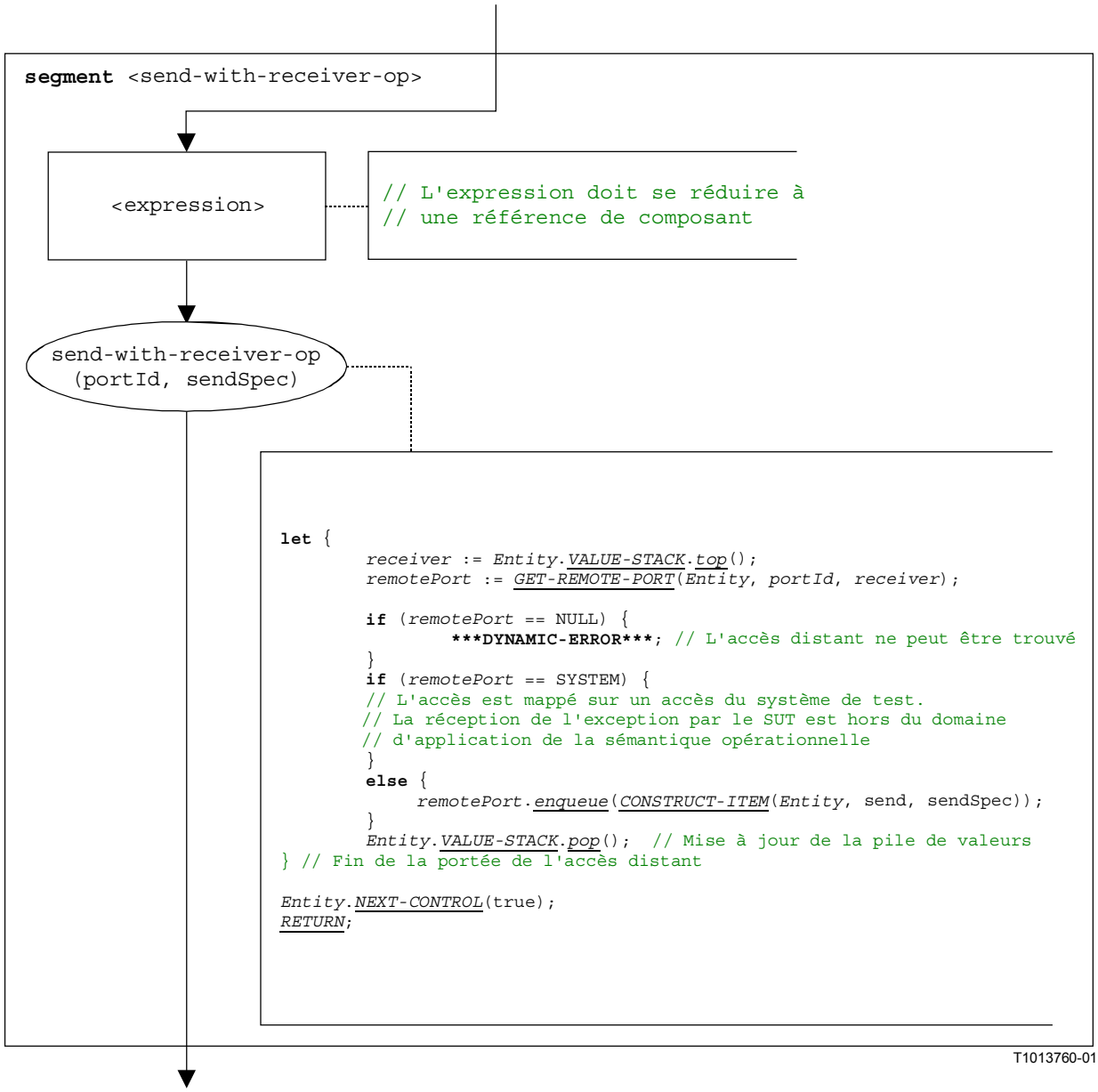


Figure B.107/Z.140 – Segment de graphe orienté <sent-with-receiver-op>

B.3.7.44.2 Segment de graphe orienté <sent-without-receiver-op>

Le segment de graphe orienté `<send-without-receiver-op>` dans la Figure B.108 définit l'exécution d'une opération **`send`** sans **`to`**.

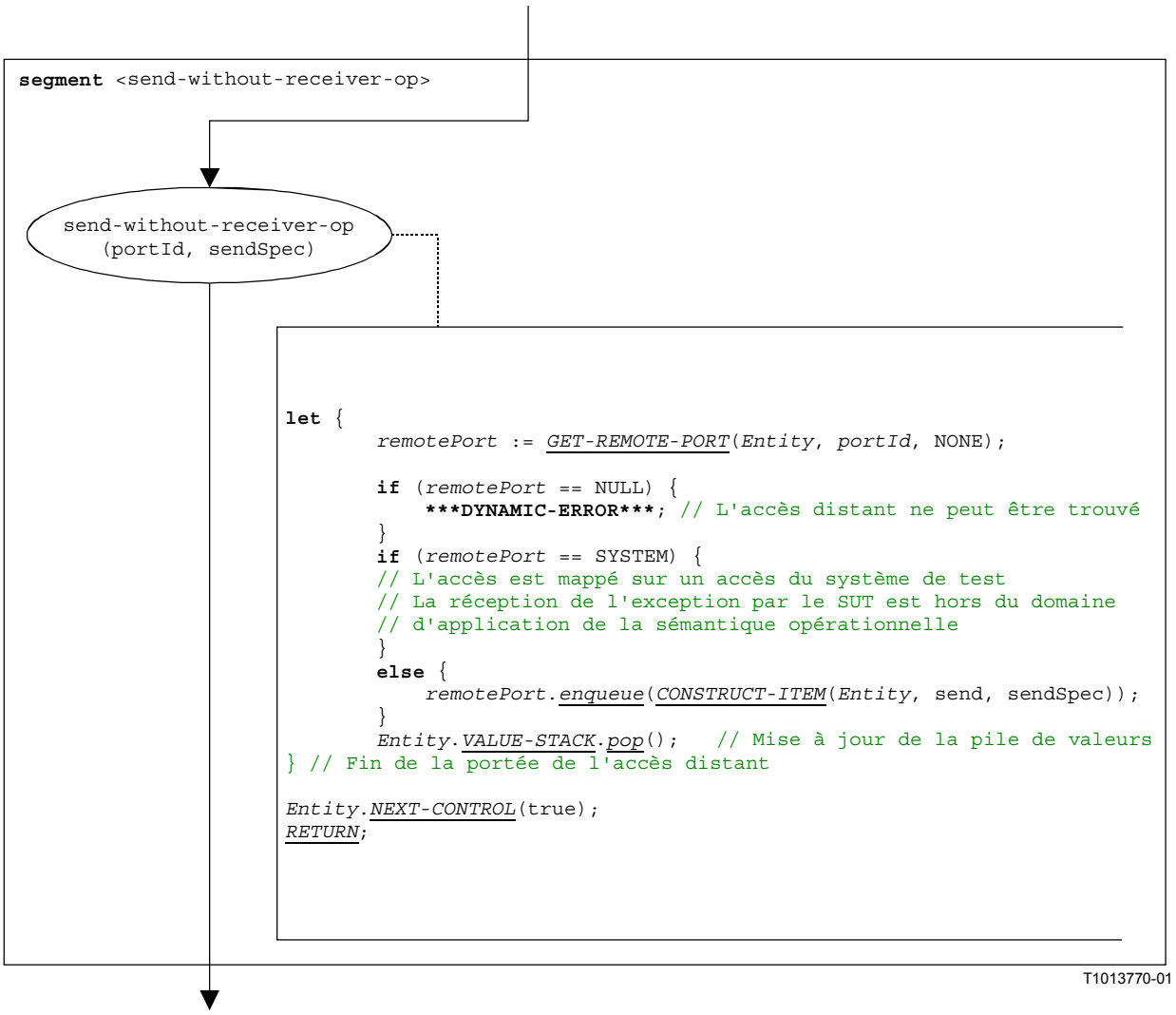


Figure B.108/Z.140 – Segment de graphe orienté <sent-without-receiver-op>

B.3.7.45 Opération automatique (self)

La structure syntaxique de l'opération **`self`** est:

`self`

Le segment de graphe orienté <`self-op`> dans la Figure B.109 définit l'exécution de l'opération **`self`**.

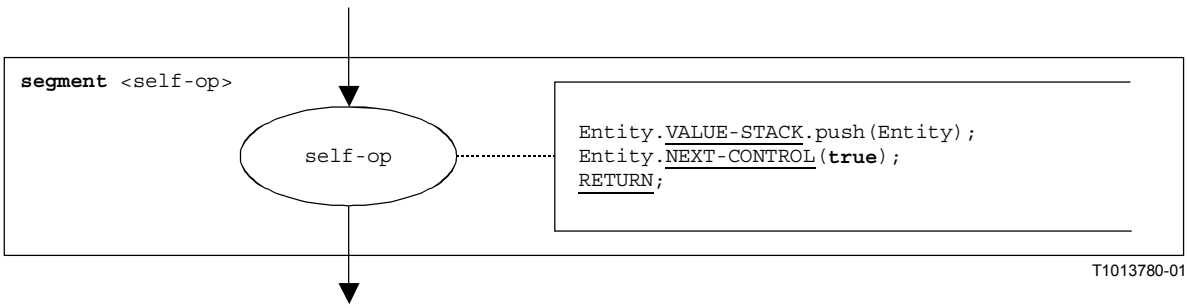


Figure B.109/Z.140 – Segment de graphe orienté <self-op>

B.3.7.46 Opération de début de composant

La structure syntaxique de l'opération **start** component est:

<component_expression>.**start**(<function-name>(<act-par-desc$_1$>, … , <act-par-desc$_n$>))

L'opération de démarrage de composant ouvre un composant nouvellement créé. L'utilisation d'une référence de composant identifie le composant qui doit démarrer. La référence peut être mémorisée dans une variable ou être renvoyée par une fonction. Par souci de simplicité, ceci est considéré comme étant une expression qui se réduit à une référence de composant.

L'expression <function-name> indique le nom de la fonction qui définit le comportement du nouveau composant et les expressions <act-par-descr$_1$>, …, <act-par-descr$_n$> fournissent la description des valeurs paramétriques réelles de <function-name>. En cas de paramètre de valeur la description d'un paramètre réel peut être fournie sous la forme d'une expression qui doit être évaluée avant que l'appel puisse être exécuté. Le traitement des paramètres formels et réels est analogue à leur traitement dans un appel de fonction (§ B.3.7.22).

Le segment de graphe orienté <start-component-op> dans la Figure B.110 définit l'exécution de l'opération **start** component. L'opération de démarrage de composant est exécutée en quatre étapes. Dans la première étape un fichier de communication est créé. Dans la deuxième étape les valeurs paramétriques réelles sont calculées. Dans la troisième étape la référence du composant qui doit démarrer est extraite, et, dans la quatrième étape de l'opération, la commande et le fichier de communication sont donnés au nouveau composant.

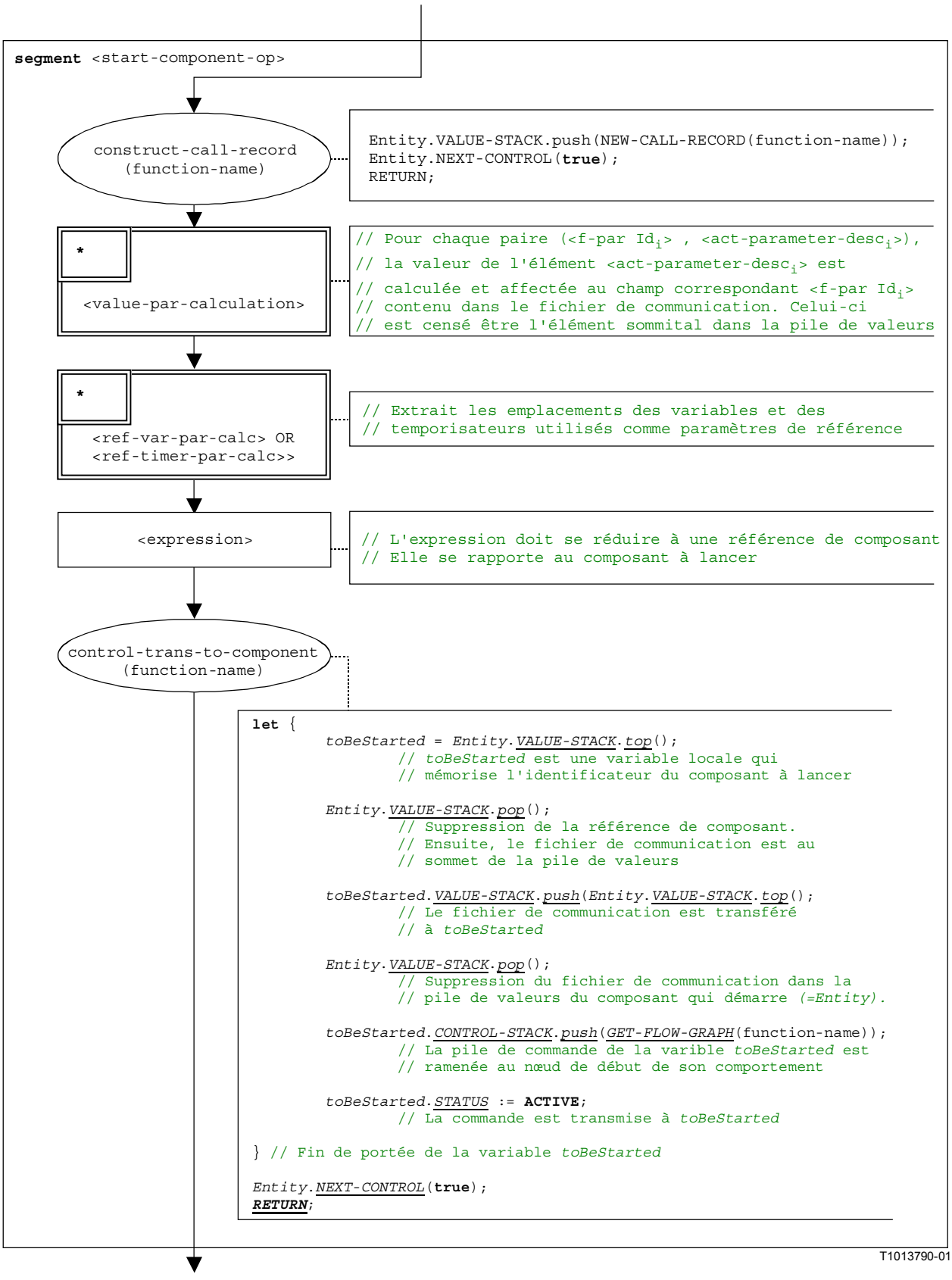


Figure B.110/Z.140 – Segment de graphe orienté <start-component-op>

B.3.7.47 Opération de début d'accès

La structure syntaxique de l'opération **start** port est:

```
<portId>.start
```

Le segment de graphe orienté <start-port-op> dans la Figure B.111 définit l'exécution de l'opération **start** port.

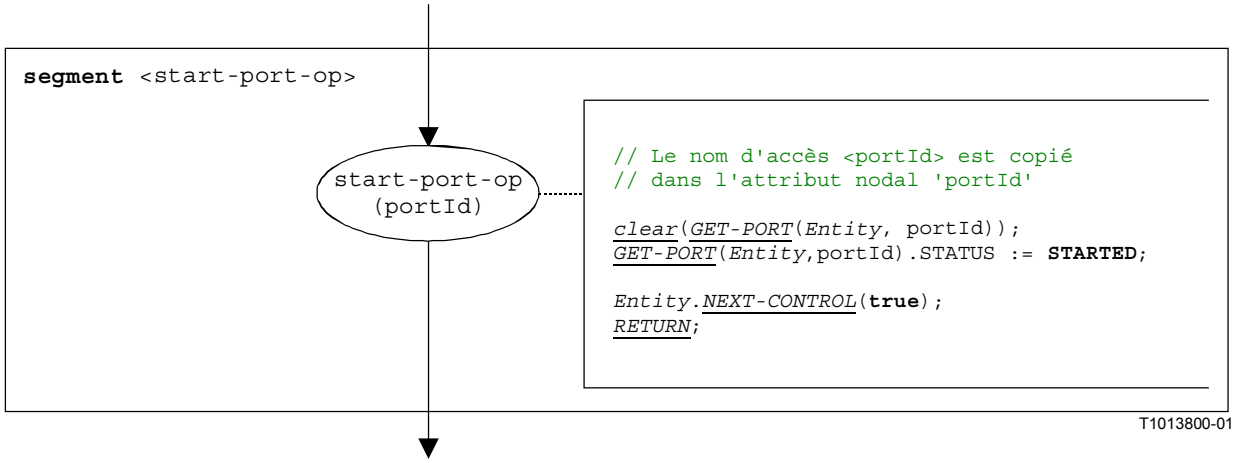


Figure B.111/Z.140 – Segment de graphe orienté <start-port-op>

B.3.7.48 Opération de début de temporisation

La structure syntaxique de l'opération **start timer** est:

```
<timerId>.start [(<float_expression>)]
```

Le paramètre <float_expression paramètre de l'opération **start timer** indique la durée facultative avec laquelle le temporisateur doit être armé. C'est une expression qui doit se réduire à une valeur de type **float**. Si fournie, l'expression doit être évaluée avant que l'opération de démarrage soit appliquée. Le résultat de l'évaluation est inséré dans la pile VALUE-STACK de l'entité.

Le segment de graphe orienté <start-timer-op> dans la Figure B.112 définit l'exécution de l'opération **start timer**.

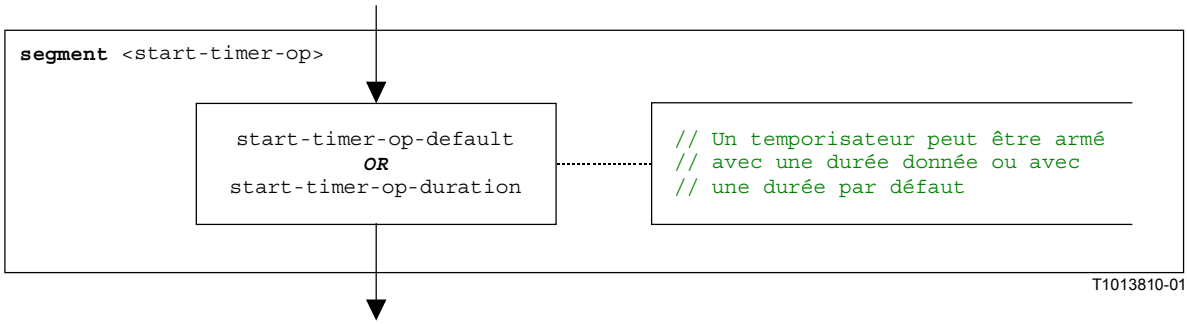


Figure B.112/Z.140 – Segment de graphe orienté <start-timer-op>

B.3.7.48.1 Segment de graphe orienté <start-timer-op-default>

Le segment de graphe orienté <start-timer-op-default> dans la Figure B.113 définit l'exécution de l'opération **start timer** avec la valeur par défaut.

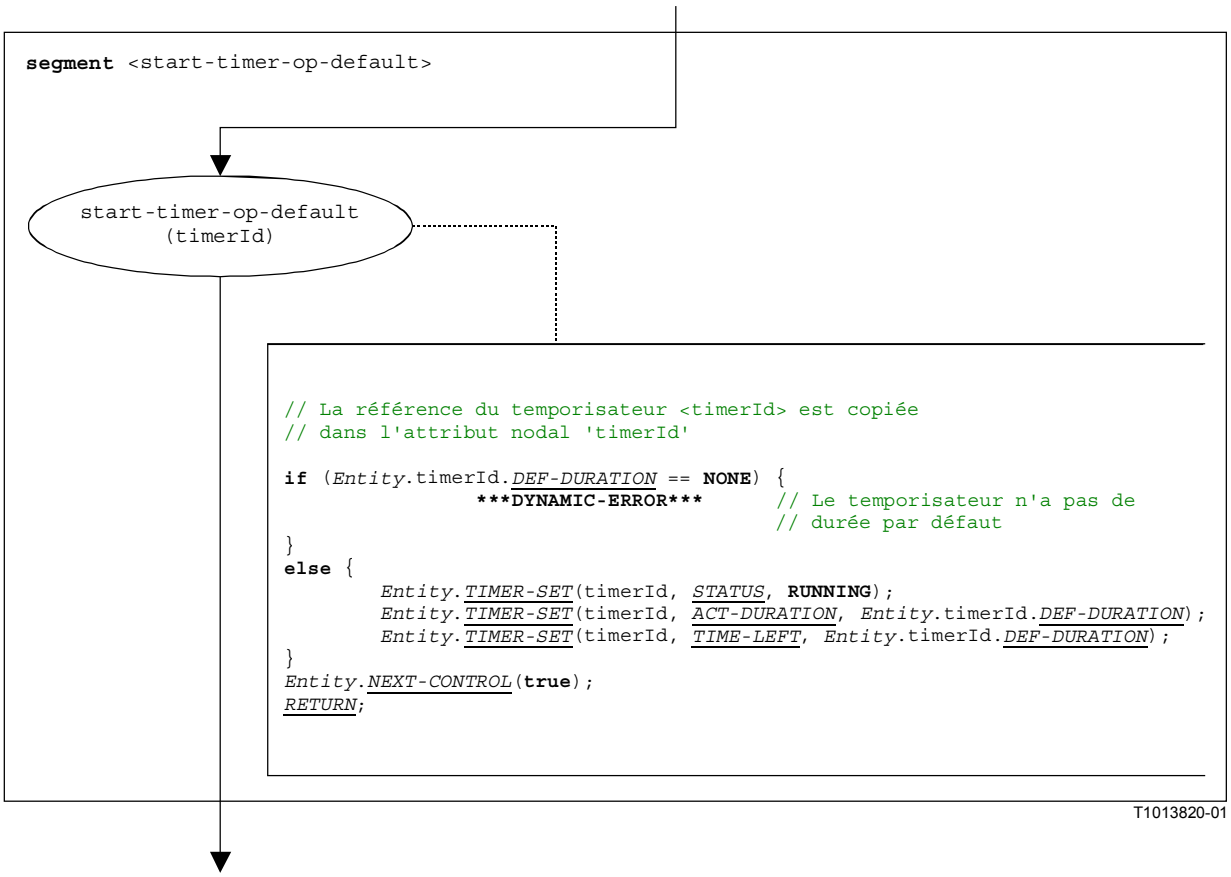


Figure B.113/Z.140 – Segment de graphe orienté <start-timer-op-default>

B.3.7.48.2 Segment de graphe orienté <start-timer-op-duration>

Le segment de graphe orienté <start-timer-op-duration> dans la Figure B.114 définit l'exécution de l'opération **start timer** avec une durée fournie.

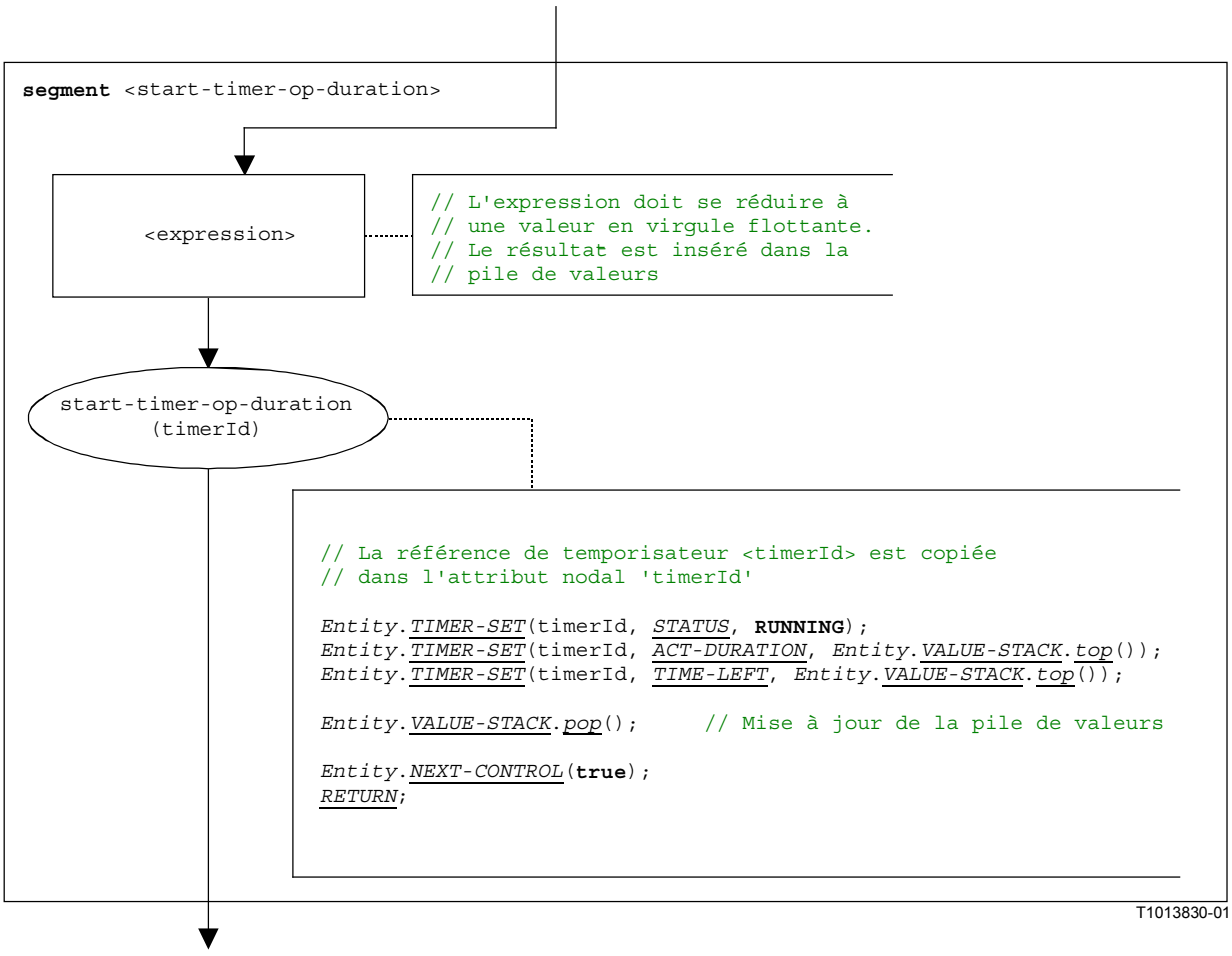


Figure B.114/Z.140 – Segment de graphe orienté <start-timer-op-duration>

B.3.7.49 Bloc d'instructions

La structure syntaxique d'un bloc d'instructions est:

{ <statement$_1$>; ... ; <statement$_n$> }

Un bloc d'instructions est une unité de portée. A l'entrée d'une unité de portée, de nouvelles portées pour variables, temporisateurs et la pile de valeurs doivent être initialisés. A la sortie d'une unité de portée, toutes les variables, temporisateurs et valeurs de pile de cette portée doivent être détruits.

Le segment de graphe orienté <statement-block> dans la Figure B.115 définit l'exécution d'un bloc d'instructions.

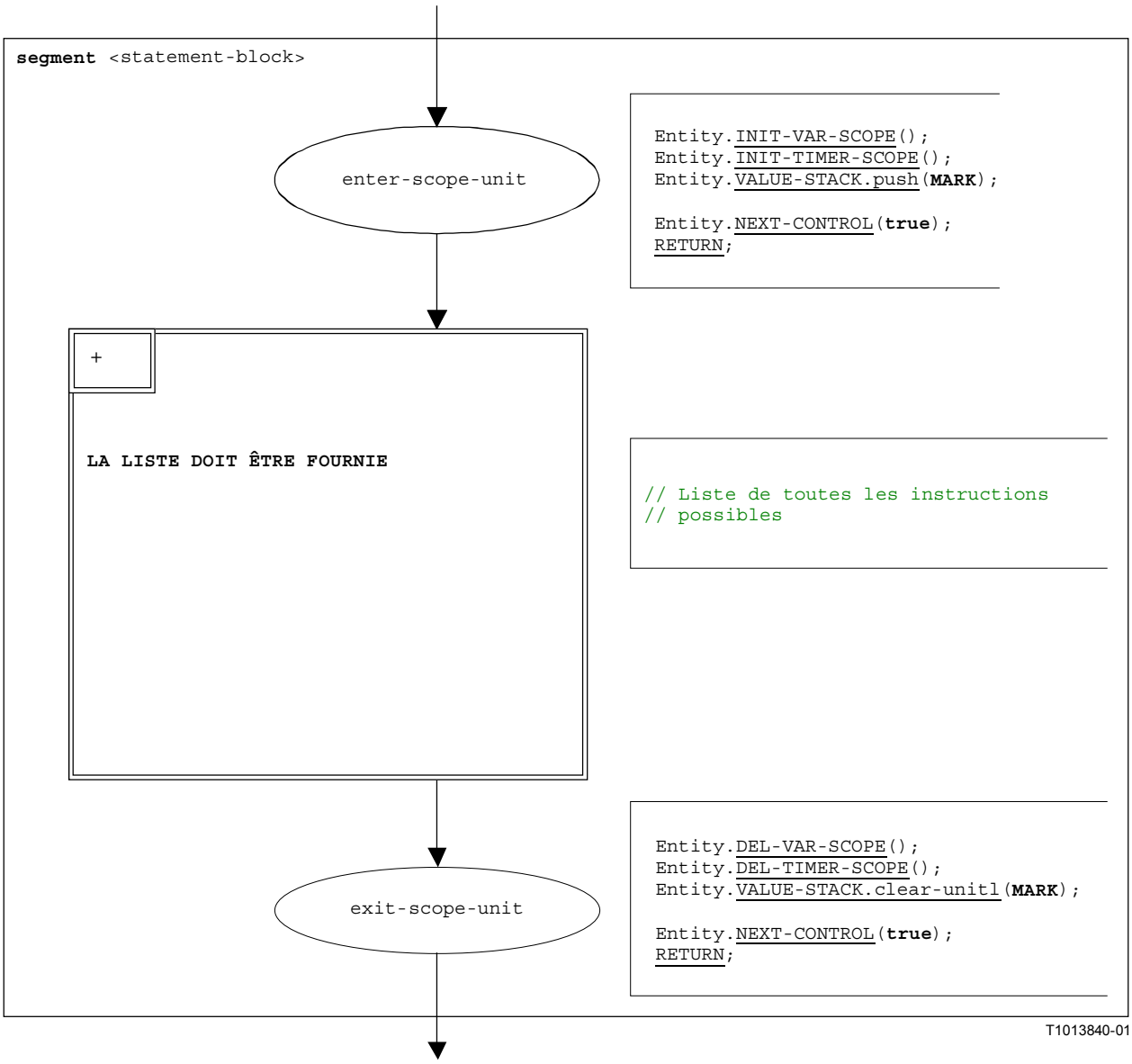


Figure B.115/Z.140 – Segment de graphe orienté <statement-block>

B.3.7.50 Opération d'arrêt

La structure syntaxique de l'opération **stop** d'une entité est:

stop

L'effet de l'opération **stop** dépend de l'entité qui l'exécute:

a) si l'opération **stop** est exécutée par la commande de module, la campagne de tests se termine, c'est-à-dire que tous les composants de tests et la commande de module disparaissent de l'état de module;

b) si l'opération **stop** est exécutée par le composant **mtc**, tous les composants de test parallèles et le composant **mtc** arrêtent l'exécution. Le verdict global de test élémentaire est mis à jour et inséré dans la pile de valeurs de la commande de module. Finalement, le contrôle est rendu à la commande de module et le composant **mtc** se termine;

c) si l'opération **stop** est exécutée par un composant de test, le verdict global de test élémentaire *TC-VERDICT* et la variable globale *DONE* sont mis à jour. Alors le composant disparaît complètement du module.

Le segment de graphe orienté <stop-entity-op> dans la Figure B.116 définit l'exécution de l'opération **stop** d'entité.

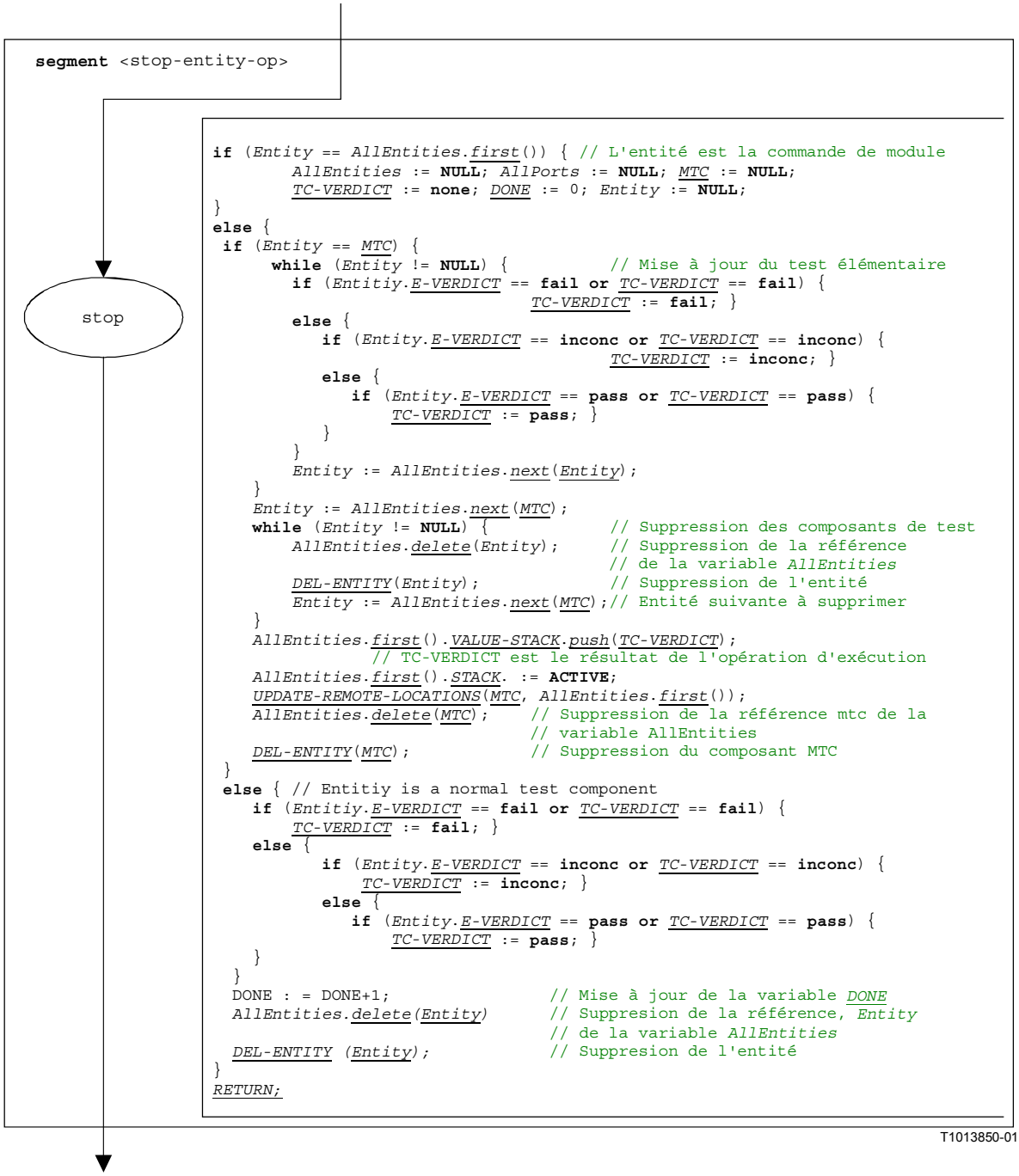


Figure B.116/Z.140 – Segment de graphe orienté <stop-entity-op>

B.3.7.51 Opération de fermeture d'accès

La structure syntaxique de l'opération **stop** port est:

<portId>**.stop**

Le segment de graphe orienté <stop-port-op> dans la Figure B.117 définit l'exécution de l'opération "**stop** port".

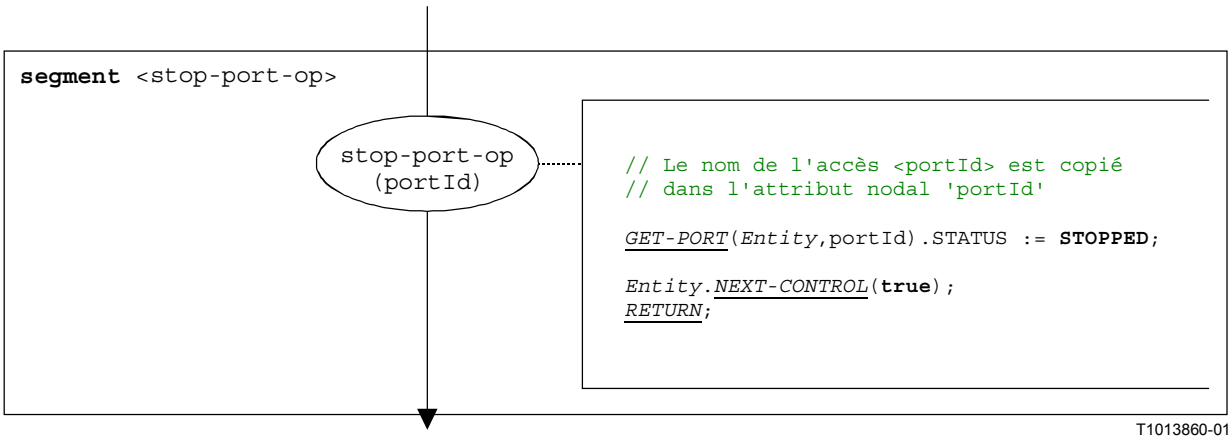


Figure B.117/Z.140 – Segment de graphe orienté <stop-port-op>

B.3.7.52 Opération de désarmement de temporisateur

La structure syntaxique de l'opération **`stop timer`** est:

`<timerId>`**`.stop`**

Le segment de graphe orienté `<stop-timer-op>` dans la Figure B.118 définit l'exécution de l'opération "**`stop timer`**".

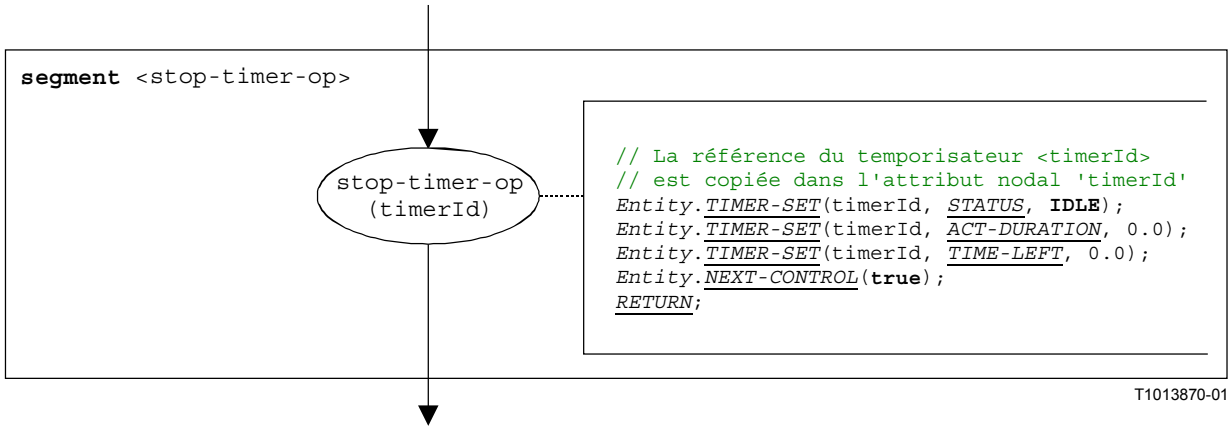


Figure B.118/Z.140 – Segment de graphe orienté <stop-timer-op>

B.3.7.53 Opération d'action de système à tester (SUT)

La structure syntaxique de l'opération **`sut.action`** est:

`sut.action` `(<informal description>)`

Le segment de graphe orienté `<sut.action-op>` dans la Figure B.119 définit l'exécution de l'opération **`sut.action`**.

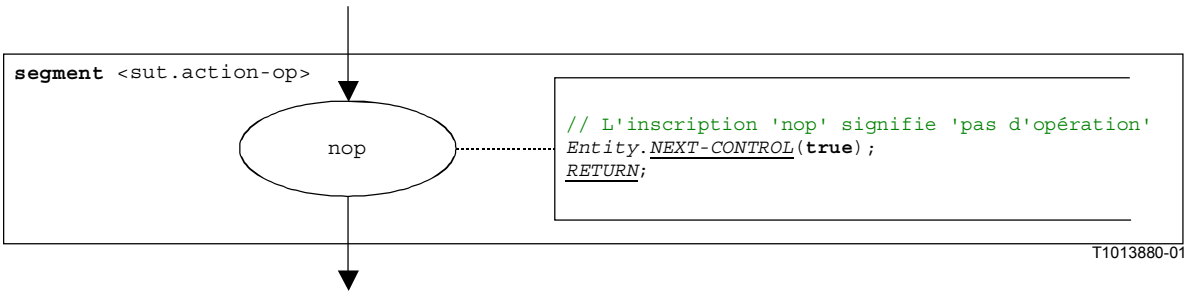


Figure B.119/Z.140 – Segment de graphe orienté <sut.action-op>

NOTE – Le paramètre <informal description> de l'opération **sut.action** n'a pas de signification pour la sémantique opérationnelle et n'est donc pas représenté dans le segment de graphe orienté.

B.3.7.54 Opération de système

La structure syntaxique de l'opération **system** est:

system

Le segment de graphe orienté <system-op> dans la Figure B.120 définit l'exécution de l'opération **system**.

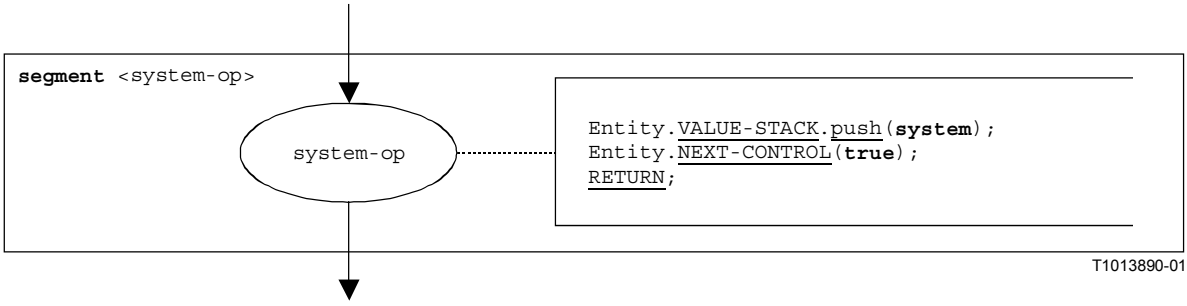


Figure B.120/Z.140 – Segment de graphe orienté <system-op>

B.3.7.55 Opération de fin de temporisation

La structure syntaxique de l'opération de fin de temporisation est:

<timerId>**.timeout**

Le segment de graphe orienté <timeout-timer-op> dans la Figure B.121 définit l'exécution de l'opération de fin de temporisation.

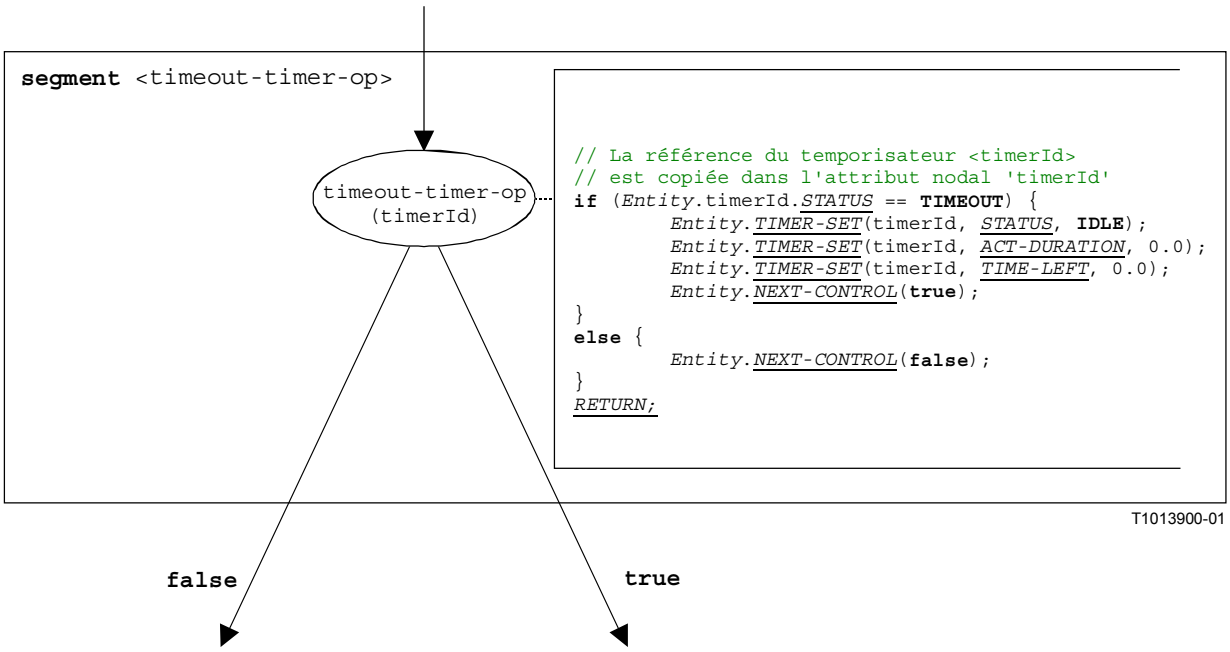


Figure B.121/Z.140 – Segment de graphe orienté <timeout-timer-op>

NOTE – Une opération **timeout** est imbriquée dans une instruction d'alternative. Selon que l'expiration donne la valeur **true** ou **false**, soit l'exécution continue avec l'instruction qui suit l'opération de fin de temporisation (branche **true**), soit la variante suivante qui est contenue dans l'instruction **alt** doit être vérifiée (branche **false**).

B.3.7.56 Opération de démappage

La structure syntaxique de l'opération **unmap** est:

unmap(<component_expression>.<portId1>,**system**.<portId2>)

Les identificateurs <portId1> et <portId2> sont considérés comme étant des identificateurs d'accès du composant de test correspondant et de l'interface avec le système de test. Le composant auquel l'accès <portId1> appartient est référencé au moyen de la référence de composant <component_expression>. La référence peut être mémorisée dans des variables ou est renvoyée par une fonction. Par souci de simplicité, elle est considérée comme étant une expression qui se réduit à une référence de composant. Donc, la pile de valeurs est utilisée pour mémoriser la référence de composant.

NOTE – L'opération **unmap** ne tient pas compte du fait que l'instruction **system**.<portId> apparaît en tant que premier ou en tant que second paramètre. Par souci de simplicité, l'on part de l'hypothèse que c'est toujours le second paramètre.

L'exécution de l'opération **unmap** est définie par le segment de graphe orienté <unmap-op> indiqué dans la Figure B.122.

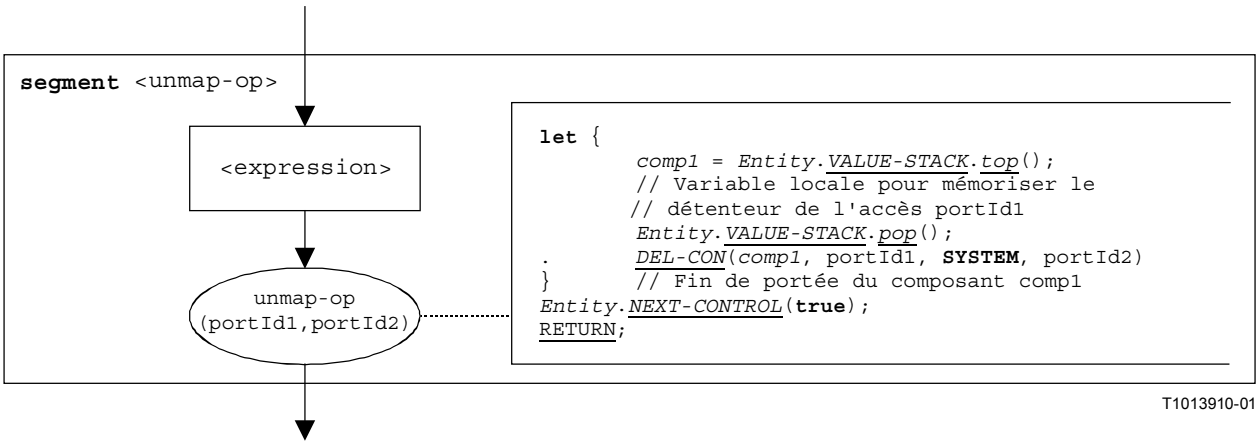


Figure B.122/Z.140 – Segment de graphe orienté <unmap-op>

B.3.7.57 Opération d'obtention de verdict (verdict.get)

La structure syntaxique de l'opération **verdict.get** est:

verdict.get

Le segment de graphe orienté <verdict.get-op> dans la Figure B.123 définit l'exécution de l'opération **verdict.get**.

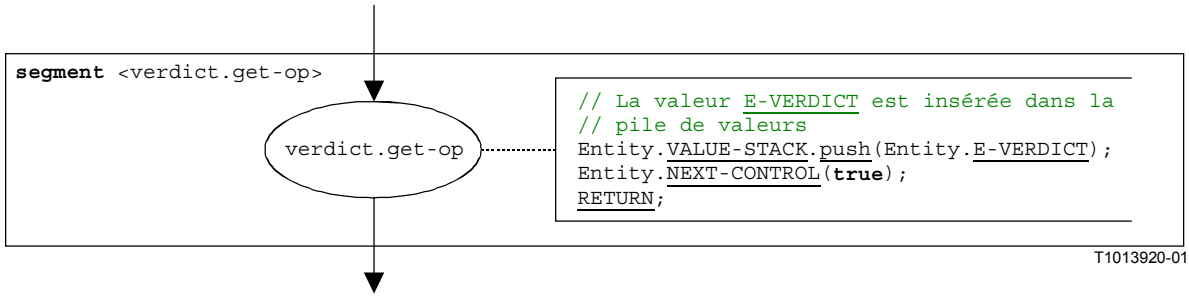


Figure B.123/Z.140 – Segment de graphe orienté <verdict.get-op>

B.3.7.58 Opération de formulation de verdict (verdict.set)

La structure syntaxique de l'opération **verdict.set** est:

verdict.set(<verdicttype_expression>)

NOTE – Le paramètre <verdicttype_expression> de l'opération **verdict.set** est une expression qui doit se réduire à une valeur de type **verdicttype**, c'est-à-dire **none**, **pass**, **inconc** ou **fail**. L'expression est évaluée avant que l'opération **verdict.set** soit appliquée.

Le segment de graphe orienté <verdict.set-op> dans la Figure B.124 définit l'exécution de l'opération **verdict.set**.

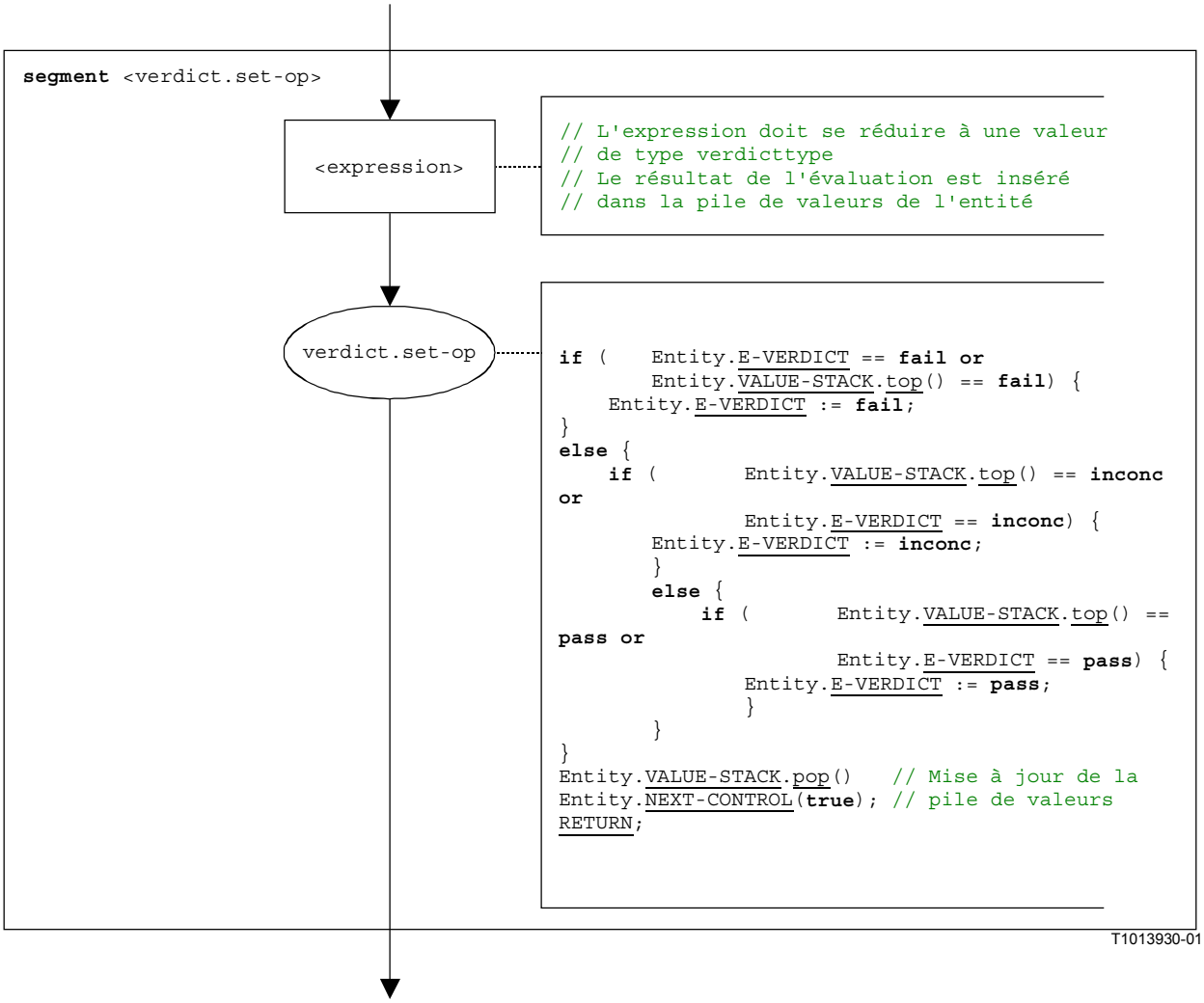


Figure B.124/Z.140 – Segment de graphe orienté <verdict.set-op>

B.3.7.59 Instruction "tant que" (while)

La structure syntaxique de l'opération **while** est:

while (<boolean-expression>) <statement-block>

L'exécution d'une instruction **while** est définie par le segment de graphe orienté <while-stmt> indiqué dans la Figure B.125.

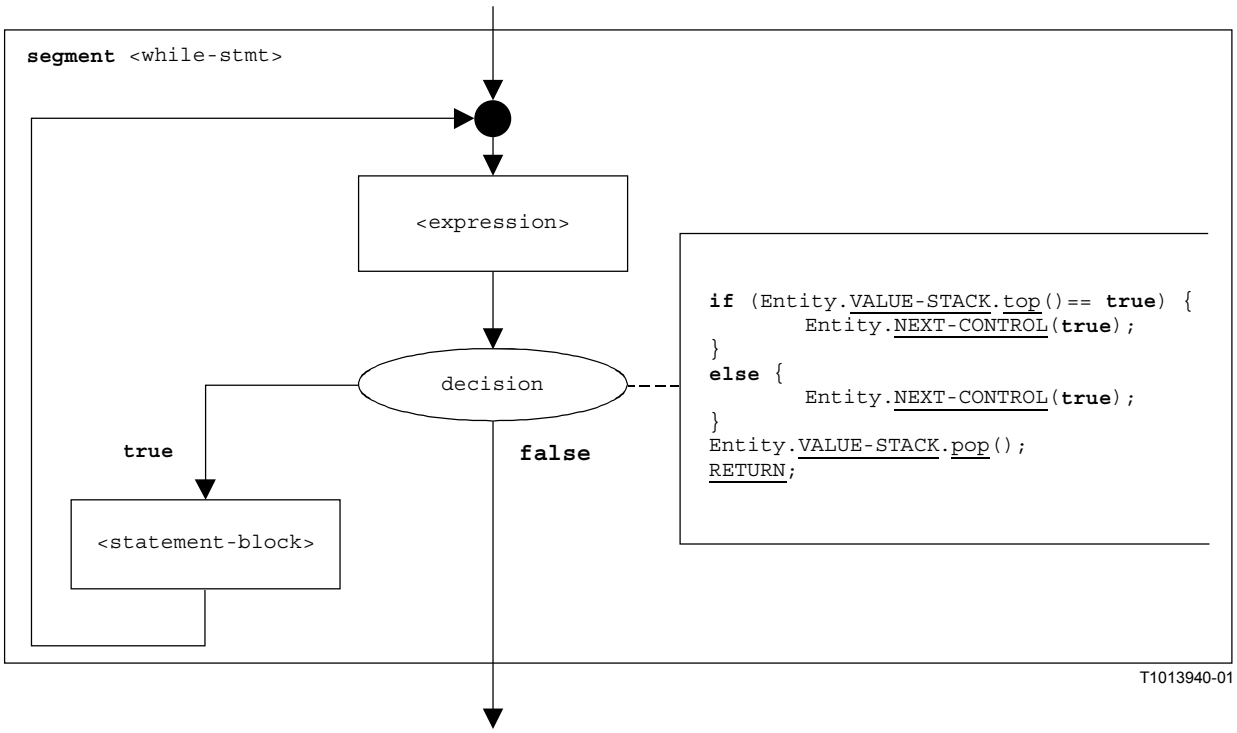


Figure B.125/Z.140 – Segment de graphe orienté <while-stmt>

B.3.8 Liste des composants sémantiques opérationnels

B.3.8.1 Fonctions et états

Nom	Description	Référence
NEXT	Extrait le nœud successeur d'un nœud donné dans un graphe orienté.	§ B.3.1.6
GET-FLOW-GRAPH	Extrait le nœud de début d'un graphe orienté	§ B.3.2.6
MTC	Référence à **mtc** dans l'état de module	§ B.3.3.1.1
TC-VERDICT	Verdict réel de test élémentaire dans l'état de module	§ B.3.3.1.1
DONE	Nombre de composants de test terminés (partie d'un état de module)	§ B.3.3.1.1
append	Opération de liste 'append': ajoute un élément en tant que dernier élément à une liste	§ B.3.3.1.1
delete	Opération de liste 'delete': supprime un élément d'une liste	§ B.3.3.1.1
first	Opération de liste 'first': renvoie le premier élément d'une liste	§ B.3.3.1.1
	Opération de file 'first': renvoie le premier élément d'une file d'attente	§ B.3.3.3.2
length	Opération de liste 'length': renvoie la longueur d'une liste	§ B.3.3.1.1
STATUS	Etat (**ACTIVE** ou **BLOCKED**) de commande de module ou de composant de test	§ B.3.3.2.1
	Etat (**IDLE**, **RUNNING** ou **TIMEOUT**) d'un temporisateur	§ B.3.3.2.4
	Etat (**STARTED** ou **STOPPED**) d'un accès	§ B.3.3.3.2

Nom	Description	Référence
E-VERDICT	Verdict de test local d'un composant de test	§ B.3.3.2.1
CONTROL-STACK	Pile de nœuds de graphe orienté indiquant l'état de commande réel d'une entité	§ B.3.3.2.1
VALUE-STACK	Pile de valeurs pour la mémorisation de résultats d'expressions, opérandes, opérations et fonctions	§ B.3.3.2.1
push	Opération de pile 'push': fait passer un élément dans une pile	§ B.3.3.2.1
pop	Opération de pile 'pop': désempile un élément d'une pile	§ B.3.3.2.1
top	Opération de pile 'top': renvoie l'élément sommital d'une pile	§ B.3.3.2.1
clear	Opération de pile 'clear': supprime une pile	§ B.3.3.2.1
	Opération de file 'clear': supprime tous les éléments d'une file d'attente	§ B.3.3.3.2
clear-until	Opération de pile 'clear-until': désempile des éléments jusqu'à ce qu'un élément spécifique soit l'élément sommital dans la pile	§ B.3.3.2.1
NEW-ENTITY	Crée un nouvel état d'entité	§ B.3.3.2.1
VAR-SET	Règle la valeur d'une variable	§ B.3.3.2.4
TIMER-SET	Règle les valeurs d'un temporisateur	§ B.3.3.2.4
DEF-DURATION	Durée par défaut d'un temporisateur	§ B.3.3.2.4
ACT-DURATION	Durée avec laquelle un temporisateur actif a été armé	§ B.3.3.2.4
TIME-LEFT	Temps qui reste à un temporisateur armé avant d'arriver à expiration	§ B.3.3.2.4
INIT-VAR	Crée une nouvelle corrélation de variable	§ B.3.3.2.4
INIT-TIMER	Crée une nouvelle corrélation de temporisateur	§ B.3.3.2.4
GET-VAR-LOC	Extrait l'emplacement d'une variable	§ B.3.3.2.4
GET-TIMER-LOC	Extrait l'emplacement d'un temporisateur	§ B.3.3.2.4
INIT-VAR-LOC	Crée une nouvelle corrélation de variable avec un emplacement existant	§ B.3.3.2.4
INIT-TIMER-LOC	Crée une nouvelle corrélation de temporisateur avec un emplacement existant	§ B.3.3.2.4
INIT-VAR-SCOPE	Initialise une nouvelle portée de variable	§ B.3.3.2.4
INIT-TIMER-SCOPE	Initialise une nouvelle portée de temporisateur	§ B.3.3.2.4
DEL-VAR-SCOPE	Supprime une portée de variable	§ B.3.3.2.4
DEL-TIMER-SCOPE	Supprime une portée de temporisateur	§ B.3.3.2.4
NEW-PORT	Crée un nouvel accès	§ B.3.3.3.2
GET-PORT	Extrait une référence d'accès	§ B.3.3.3.2
GET-REMOTE-PORT	Extrait la référence d'un d'accès distant	§ B.3.3.3.2
ADD-CON	Ajoute une connexion à un état d'accès	§ B.3.3.3.2
DEL-CON	Supprime une connexion à partir d'un état d'accès	§ B.3.3.3.2
enqueue	Opération de file 'enqueue': met un élément en tant que dernier élément dans une file d'attente	§ B.3.3.3.2

Nom	Description	Référence
dequeue	Opération de file 'dequeue': supprime le premier élément d'une file d'attente	§ B.3.3.3.2
DEL-ENTITY	Supprime une entité à partir d'un état de module	§ B.3.3.4
EXISTING	Vérifie si un composant de test existe ou non	§ B.3.3.4
UPDATE-REMOTE-REFERENCES	Met à la même valeur temporisateurs et variables ayant le même emplacement dans des entités différentes	§ B.3.3.4
CONSTRUCT-ITEM	Construit un élément à expédier	§ B.3.4.3
MATCH-ITEM	Vérifie si un message reçu, un appel, une réponse ou une exception correspond à une opération de réception	§ B.3.4.4
RETRIEVE-INFO	Extrait des informations d'un message reçu, d'un appel, d'une réponse ou d'une exception	§ B.3.4.4
NEW-CALL-RECORD	Crée un fichier de communication pour un appel de fonction	§ B.3.5.1
INIT-FLOW-GRAPHS	Initialise le traitement par graphe orienté	§ B.3.6.1
GET-UNIQUE-ID	Renvoie un nouvel identificateur unique lorsqu'elle est appelée	§ B.3.6.1
CONTINUE-COMPONENT	Le composant réel continue son exécution	§ B.3.6.1
RETURN	Renvoie la commande à la procédure d'évaluation de module définie au § B.3.6	§ B.3.6.1
DYNAMIC-ERROR	Décrit l'occurrence d'une erreur dynamique	§ B.3.6.1

B.3.8.2 Mots clés spéciaux

Mot clé	Description	Référence
MARK	Utilisé en tant que marque pour *VALUE-STACK*	§ B.3.3.2
ACTIVE	*ETAT* d'un état d'entité	§ B.3.3.2
BLOCKED	*ETAT* d'un état d'entité	§ B.3.3.2
NULL	Valeur symbolique pour les types pointeur et analogues afin d'indiquer qu'aucune adresse n'est visée	
IDLE	*ETAT* d'un état de temporisateur	§ B.3.3.2.4
RUNNING	*ETAT* d'un état de temporisateur	§ B.3.3.2.4
TIMEOUT	*ETAT* d'un état de temporisateur	§ B.3.3.2.4
STARTED	*ETAT* d'un accès	§ B.3.3.2.4
STOPPED	*ETAT* d'un accès	§ B.3.3.2.4
NONE	Utilisée pour décrire une valeur indéfinie	

B.3.8.3 Segments de graphe orienté

Identificateur	Structure TTCN-3 associée	Référence	
		Figure	§
<alt-stmt>	instruction d'alternative	Figure B.25	B.3.7.1
<alt-with-else>	instruction d'alternative	Figure B.26	B.3.7.1
<alt-without-else>	instruction d'alternative	Figure B.27	B.3.7.1
<assignment-stmt >	instruction d'affectation	Figure B.29	B.3.7.2
<b-call-with-receiver>	appel	Figure B.35	B.3.7.3.3
<b-call-without-receiver>	appel	Figure B.36	B.3.7.3.4
<b-call-with-rec-dur>	appel	Figure B.37	B.3.7.3.5
<b-call-without-rec-dur>	appel	Figure B.38	B.3.7.3.6
<blocking-call-op>	appel	Figure B.31	B.3.7.3
<call-op>	appel	Figure B.30	B.3.7.3
<catch-op>	acquisition	Figure B.39	B.3.7.4
<catch-with-sender>	utilisée dans l'opération d'acquisition	Figure B.40	B.3.7.4.1
<catch-without-sender>	utilisée dans l'opération d'acquisition	Figure B.41	B.3.7.4.2
<clear-port-op>	libération d'accès	Figure B.42	B.3.7.5
<constant-declaration>	déclaration d'une constante	Figure B.44	B.3.7.7
<connect-op>	connexion	Figure B.43	B.3.7.6
<create-op>	création	Figure B.45	B.3.7.8
<disconnect-op>	déconnexion	Figure B.53	B.3.7.12
<do-while-stmt>	instruction "faire tant que"	Figure B.54	B.3.7.13
<done-all-comp-op>	tous composants terminés	Figure B.55	B.3.7.14
<done-any-comp-op>	tout composant terminé	Figure B.56	B.3.7.15
<done-component-op>	composant terminé	Figure B.57	B.3.7.16
<execute-stmt>	exécution	Figure B.58	B.3.7.17
<execute-timeout>>	exécution	Figure B.59	B.3.7.17
<execute-without-timeout>>	exécution	Figure B.60	B.3.7.17
<expression>	expression	Figure B.61	B.3.7.18
<finalize-component-init>	structure utilisée dans le comportement de définitions de type "composant"	Figure B.66	B.3.7.19
<for-stmt>>	instruction "pour"	Figure B.68	B.3.7.21
<function-call>	appel de fonctions définies par l'utilisateur	Figure B.69	B.3.7.22
<func-op-call>	utilisée dans <expression>	Figure B.64	B.3.7.18.3
<getcall-op>	**getcall**	Figure B.74	B.3.7.27

Identificateur	Structure TTCN-3 associée	Référence	
		Figure	§
<getcall-with-sender>	utilisée dans l'opération **getcall**	Figure B.75	B.3.7.27.1
<getcall-without-sender>	utilisée dans l'opération **getcall**	Figure B.76	B.3.7.27.2
<getreply-op>	**getreply**	Figure B.76	B.3.7.28
<getreply-with-sender>	utilisée dans l'opération **getreply**	Figure B.78	B.3.7.28.1
<getreply-without-sender>	utilisée dans l'opération **getreply**	Figure B.79	B.3.7.28.2
<goto-stmt>	**goto**	Figure B.80	B.3.7.29
<if-else-stmt>	**if-else**	Figure B.80	B.3.7.30
<if-with-else-branch>	**if-else**	Figure B.82	B.3.7.30.1
<if-without-else-branch>	**if-else**	Figure B.83	B.3.7.30.2
<init-component-scope>	Utilisée dans le comportement de définitions de type de composant	Figure B.67	B.3.7.20
<label-stmt>	**label**	Figure B.84	B.3.7.31
<lit-value>	Utilisée dans <expression>	Figure B.62	B.3.7.18.1
<log-stmt>	**log**	Figure B.85	B.3.7.32
<map-op>	Opération **map**	Figure B.86	B.3.7.33
<mtc-op>	**mtc**	Figure B.87	B.3.7.34
<nb-call-with-receiver>	**call**	Figure B.33	B.3.7.3.1
<nb-call-without-receiver>	**call**	Figure B.34	B.3.7.3.2
<non-blocking-call-op>	**call**	Figure B.32	B.3.7.3
<operator-appl>	utilisée dans <expression>	Figure B.65	B.3.7.18.4
<parameter-handling>	création d'entités, d'appels de fonction	Figure B.73	B.3.7.26
<port-declaration>	déclaration d'un accès	Figure B.46	B.3.7.9
<raise-op>	**raise**	Figure B.88	B.3.7.35
<raise-with-receiver-op>	**raise**	Figure B.89	B.3.7.35.1
<raise-without-receiver-op>	**raise**	Figure B.90	B.3.7.35.2
<read-timer-op>	**read** (lecture de temporisateur)	Figure B.91	B.3.7.36
<receive-assignment>	utilisée dans l'opération **receive**	Figure B.95	B.3.7.37.3
<receive-op>	**receive**	Figure B.92	B.3.7.37
<receive-with-sender>	utilisée dans l'opération **receive**	Figure B.93	B.3.7.37.1
<receive-without-sender>	utilisée dans l'opération **receive**	Figure B.94	B.3.7.37.2
<receiving-branch>	**alt**	Figure B.28	B.3.7.1.1
<reply-op>	**reply**	Figure B.96	B.3.7.38

Identificateur	Structure TTCN-3 associée	Référence	
		Figure	§
<reply-with-receiver-op>	**reply**	Figure B.97	B.3.7.38.1
<reply-without-receiver-op>	**reply**	Figure B.98	B.3.7.38.2
<ref-par-var-calc>	création d'entités, appel de fonction	Figure B.71	B.3.7.24
<ref-par-timer-calc>	création d'entités, appel de fonction	Figure B.72	B.3.7.25
<return-stmt>	**return**	Figure B.99	B.3.7.39
<return-with-value>	**return**	Figure B.100	B.3.7.39.1
<return-without-value>	**return**	Figure B.101	B.3.7.39.2
<running-all comp-op>	**all component.running**	Figure B.102	B.3.7.40
<running-any comp-op>	**any component.running**	Figure B.103	B.3.7.41
<running-component-op>	**running** (composant)	Figure B.104	B.3.7.42
<running-timer-op>	**running** (temporisateur)	Figure B.105	B.3.7.43
<self-op>	**self**	Figure B.109	B.3.7.45
<send-op>	**send**	Figure B.106	B.3.7.44
<send-with-receiver-op>	**send**	Figure B.107	B.3.7.44.1
<send-without-receiver-op>	**send**	Figure B.108	B.3.7.44.2
<start-component-op>	**start** (composant)	Figure B.110	B.3.7.46
<start-port-op>	**start** (accès)	Figure B.111	B.3.7.47
<start-timer-op>	armement de temporisateur	Figure B.112	B.3.7.48
<start-timer-op-default>	armement de temporisateur	Figure B.113	B.3.7.48.1
<start-timer-op-duration>	armement de temporisateur	Figure B.114	B.3.7.48.2
<stop-entity-op>	arrêt d'exécution de commande de module, de composant mtc ou de composant de test	Figure B.116	B.3.7.50
<stop-port-op>	**stop** (accès)	Figure B.117	B.3.7.51
<statement-block>	bloc d'instructions	Figure B.115	B.3.7.49
<stop-timer-op>	**stop** (temporisateur)	Figure B.118	B.3.7.52
<sut.action-op>	**sut.action-op**	Figure B.119	B.3.7.53
<system-op>	**system**	Figure B.120	B.3.7.54
<timeout-timer-op>	**timeout** (temporisateur)	Figure B.121	B.3.7.55
<timer-declaration>	Déclaration d'un temporisateur	Figure B.47	B.3.7.10
<timer-decl-default>	Déclaration d'un temporisateur avec une durée par défaut	Figure B.48	B.3.7.10.1
<timer-decl-no-def>	déclaration d'un temporisateur sans durée par défaut	Figure B.49	B.3.7.10.2
<unmap-op>	opération **unmap**	Figure B.122	B.3.7.56
<value-par-calculation>	création d'entités, d'appels de fonction	Figure B.70	B.3.7.23

Identificateur	Structure TTCN-3 associée	Référence	
		Figure	§
<variable-declaration>	déclaration d'une variable	Figure B.50	B.3.7.11
<variable-declaration-init>	déclaration d'une variable avec valeurs initiales	Figure B.51	B.3.7.11.1
<variable-declaration-undef>	déclaration d'une variable sans valeur initiale	Figure B.52	B.3.7.11.2
<var-value>	utilisée dans <expression>	Figure B.63	B.3.7.18.2
<verdit.get-op>	**verdict.get**	Figure B.123	B.3.57
<verdit.set-op>	**verdict.set**	Figure B.124	B.3.7.58
<while-stmt>	**while** (instruction)	Figure B.125	B.3.7.59

Annexe C

Appariement de valeurs entrantes

C.1 Mécanismes d'appariements de modèles

La présente annexe spécifie les mécanismes d'appariement qui peuvent être utilisés dans les modèles de notation TTCN-3 (et seulement dans ces modèles).

C.1.1 Valeurs de concordance spécifiques

Les valeurs spécifiques sont les mécanismes d'appariement de base des modèles TTCN-3. Les valeurs spécifiques qui sont contenues dans les modèles sont des expressions qui ne contiennent pas de mécanismes d'appariement ou de structures génériques. Sauf indication contraire, un champ de modèle s'apparie avec la valeur de champ entrante correspondante si, et seulement si, cette valeur de champ entrante est exactement identique à la valeur à laquelle l'expression contenue dans le modèle se réduit. Par exemple:

```
// Soit la définition du type de message

type record MyMessageType
{
     integer  field1,
     charstring  field2,
     boolean  field3 optional,
     integer[4]  field4
}

// Un modèle de message utilisant des valeurs spécifiques pourrait être:
template MyMessageType MyTemplate:=
{
     field1 := 3+2,          // Valeur spécifique de type entier
     field2 := "My string",  // Valeur spécifique de type chaîne de caractères
     field3 := true,         // Valeur spécifique de type booléen
     field4 := {1,2,3}       // Valeur spécifique de séquence tabulaire d'entiers
}
```

C.1.2 Mécanismes d'appariement au lieu de valeurs

C.1.2.1 Liste de valeurs

Les listes de valeurs énumèrent des valeurs entrantes acceptables. Elles peuvent être utilisées pour des valeurs de tous types. Un champ de modèle qui utilise une liste de valeurs s'apparie avec le champ entrant correspondant si, et seulement si, la valeur de ce champ entrant correspond à l'une quelconque des valeurs contenues dans la liste. Chaque valeur de la liste doit être du type déclaré pour le champ de modèle dans lequel le mécanisme est utilisé. Par exemple:

```
template Mymessage MyTemplate:=
{
     field1 := (2,4,6),                    // Liste des valeurs d'entier
     field2 := ("String1", "String2"),     // Liste des valeurs de chaîne
                                           // de caractères
     :
     :
}
```

C.1.2.2 Liste de valeurs complémentées

Le mot clé **complement** indique une liste de valeurs qui ne sera pas acceptée en tant que valeurs entrantes (c'est-à-dire qu'il est le complément d'une liste de valeurs). Elle peut être utilisée pour toutes les valeurs de tous les types.

Chaque valeur dans la liste doit être du type déclaré pour le champ de modèle dans lequel le complément est utilisé. Un champ de modèle qui utilise un complément s'apparie avec le champ entrant correspondant si, et seulement si, le champ entrant ne s'apparie avec aucune des valeurs énumérées dans la liste de valeurs. La liste de valeurs peut être une valeur isolée, évidemment.

Exemple:

```
template Mymessage MyTemplate:=
{
     complement (1,3,5),          // Liste des valeurs inacceptables d'entier
     :
     field3 not(true)             // Apparie toute valeur d'entier false
     :
}
```

C.1.2.3 Omission de valeurs

Le mot clé **omit** indique qu'un champ facultatif de modèle doit être absent. Il peut être utilisé pour les valeurs de tous les types, à condition que le champ de modèle soit déclaré facultatif. Par exemple:

```
template Mymessage:MyTemplate:=
{    :
     :
     field3 := omit,              // Omettre ce champ
     :
}
```

C.1.2.4 Valeur quelconque

Le symbole d'appariement "**?**" (*AnyValue*) est utilisé pour indiquer que toute valeur valide entrante est acceptable. Il peut être utilisé pour les valeurs de tous les types. Un champ de modèle qui utilise le mécanisme de valeur quelconque s'apparie avec le champ entrant correspondant si, et seulement si, le champ entrant donne la valeur d'un élément particulier du type spécifié. Par exemple:

```
template Mymessage:MyTemplate:=
{
     field1 := ?,  // S'apparie avec toute valeur d'entier
     field2 := ?,  // S'apparie avec toute valeur de chaîne de caractères
                   // non vide
     field3 := ?,  // S'apparie avec toute valeur "true" ou "false"
     field4 := ?   // S'apparie avec toute séquence d'entiers
}
```

C.1.2.5 Valeur quelconque ou inexistante

Le symbole d'appariement "*****" (*AnyValueOrNone*) est utilisé pour indiquer que toute valeur valide entrante, y compris l'omission de cette valeur, est acceptable. Il peut être utilisé pour les valeurs de tous les types, à condition que le champ de modèle soit déclaré comme étant facultatif.

Un champ de modèle qui utilise ce symbole s'apparie avec le champ entrant correspondant si, et seulement si, soit le champ entrant donne la valeur d'un élément quelconque du type spécifié, soit le champ entrant est absent. Par exemple:

```
template Mymessage:MyTemplate:=
{    :
     field3 := *,  // S'apparie avec toute valeur "true" ou "false" ou champ omis
     :
}
```

C.1.2.6 Etendue de valeurs

Les étendues indiquent un domaine borné de valeurs acceptables. Elles ne doivent être utilisées que pour des valeurs de type entier (et leurs sous-types). Une valeur limite peut être:

a) soit l'infini ou -l'infini;

b) soit une expression qui se réduit à une valeur d'entier spécifique.

La borne inférieure doit être mise à gauche de l'étendue et la borne supérieure à droite. La borne inférieure doit être inférieure à la borne supérieure. Un champ de modèle qui utilise une étendue s'apparie avec le champ entrant correspondant si, et seulement si, la valeur de ce champ entrant est égale à une des valeurs contenues dans l'étendue. Par exemple:

```
template Mymessage:MyTemplate:=
{
     field1 := (1 .. 6),     // étendue de type entier
     :
     :
     :
}
// d'autres entrées pour le champ field1 pourraient être (-infinity à 8) or
// (12 à infinity)
```

C.1.3 Valeurs internes des mécanismes d'appariement

C.1.3.1 Elément quelconque

Le symbole d'appariement "**?**" (*AnyElement*) est utilisé pour indiquer qu'il remplace des éléments particuliers d'une chaîne (sauf les chaînes de caractères), un mot clé **record of**, **set of** ou une séquence tabulaire. Il ne doit être utilisé qu'à l'intérieur de valeurs de type chaîne, **record**, **set of** et séquences tabulaires. Par exemple:

```
template Mymessage MyTemplate:=
{    :
     field2 := "abcxyz",
     field3 := '10???'B,     // Où chaque "?" peut être 0 ou 1
     field4 := {1, ?, 3}     // Où ? peut être tout entier
}
```

NOTE – Le symbole "**?**" dans le champ `field4` peut être interprété comme variable *AnyValue* en tant que valeur d'entier ou comme variable *AnyElement* à l'intérieur d'un ensemble **`record of`**, **`set of`** ou séquence tabulaire. Comme les deux interprétations conduisent au même appariement, aucun problème n'est soulevé.

C.1.3.1.1 Utilisation de structures génériques à caractère unique

S'il est requis d'exprimer la structure générique "**?**" en chaînes de caractères, cela doit être fait au moyen de structures de caractères (voir § C.1.5). Par exemple, les chaînes "`abcdxyz`", "`abccxyz`" "`abcxxyz`" etc., correspondront toutes à la structure **`(pattern)`** "`abc?xyz`". Ce ne sera cependant pas le cas des chaînes "`abcxyz`", "`abcdefxyz`", etc.

C.1.3.2 Nombre quelconque d'éléments ou absence d'élément

Le symbole d'appariement "*****" (*AnyElementsOrNone*) est utilisé pour indiquer qu'il remplace zéro ou *n* éléments consécutifs d'une chaîne (sauf les chaînes de caractères) d'un fichier **`record of`**, d'un ensemble **`set of`** ou d'une séquence tabulaire. Il ne doit être utilisé qu'à l'intérieur de valeurs de types chaîne ou de séquences tabulaires. Le symbole "*****" correspond à la plus longue séquence d'éléments possible, selon la structure spécifiée par les symboles qui entourent le symbole "*****". Par exemple:

```
template Mymessage MyTemplate:=
{    :
     field2 := "abcxyz",
     field3 := '10*11'B,     // Où "*" peut être toute séquence de bits
                             // (le cas échéant vide)
     field4 := {*, 2, 3}     // Où le premier élément peut être toute valeur
                             // d'entier ou être omis
}

var charstring MyStrings[4];
MyPCO.receive(MyStrings:{"abyz", *, "abc" });
```

Si un symbole "*****" apparaît au niveau le plus élevé à l'intérieur d'une chaîne, d'un fichier **`record of`**, d'un ensemble **`set of`** ou d'une séquence tabulaire, il doit être interprété comme *AnyElementsOrNone*.

NOTE – Cette règle empêche l'interprétation (sinon possible) de "*****" en tant que variable *AnyValueOrNone* qui remplace un élément à l'intérieur d'une chaîne, d'un fichier **`record of`**, d'un ensemble **`set of`** ou d'une séquence tabulaire.

C.1.3.2.1 Utilisation de structures génériques à caractères multiples

S'il est requis d'exprimer la structure générique "*****" dans les chaînes de caractères, cela doit être fait au moyen de structures de caractères (voir § C.1.5). Par exemple: "`abcxyz`", "`abcdefxyz`" "`abcabcxyz`" etc., correspondront toutes à la structure **`pattern`** "`abc*xyz`".

C.1.4 Attributs d'appariement de valeurs

C.1.4.1 Restrictions de longueur

L'attribut de restriction de longueur est utilisé pour diminuer la longueur de valeurs de chaîne et le nombre d'éléments dans une structure **`set of`** ou **`record of`**. Il ne doit être utilisé qu'en tant qu'attribut des mécanismes suivants: Complément, *AnyValue*, *AnyValueOrNone*, *AnyElement* et *AnyElementsOrNone*. Il peut également être utilisé en association avec l'attribut **`ifpresent`**. La syntaxe de l'attribut **`length`** peut être trouvée aux § 6.2.3 et 6.3.3.

Les unités de longueur doivent être interprétées selon le Tableau 4 dans le cas de valeurs de chaîne. Pour les types **`set of`** et **`record of`** l'unité de longueur est le type répliqué. Les bornes doivent être indiquées par des expressions qui se réduisent à des valeurs d'entier spécifiques et non négatives. En

variante, le mot clé **infinity** peut être utilisé en tant que valeur de borne supérieure afin d'indiquer qu'il n'y a aucune borne supérieure de longueur.

Les spécifications de longueur de modèle ne doivent pas être en contradiction avec les restrictions de longueur (éventuelles) du type correspondant. Un champ de modèle qui utilise l'attribut de longueur en tant qu'attribut d'un symbole s'apparie avec le champ entrant correspondant si, et seulement si, ce champ entrant s'apparie à la fois avec le symbole et avec son attribut associé. L'attribut de longueur s'apparie si la longueur du champ entrant est supérieure ou égale à la borne inférieure spécifiée et inférieure ou égale à la borne supérieure. En cas de valeur de longueur isolée, l'attribut de longueur ne s'apparie que si la longueur du champ reçu est exactement égale à la valeur spécifiée.

Dans le cas d'un champ omis, l'attribut de longueur est toujours considéré comme concordant (c'est-à-dire qu'il est redondant avec la structure d'omission **omit**). Avec la variable *AnyValueOrNone* et avec l'indicateur **ifpresent**, il impose une restriction à l'éventuelle valeur entrante. Par exemple:

```
template Mymessage MyTemplate:=
{
     field1 := complement (4,5) length (1 .. 6), // Est identique à (1,2,3,6)
     field2 := "ab*ab" length(13)      // La longueur maximale de la chaîne
                                       // AnyElementsOrNone est de 9 caractères
       :
     }
```

C.1.4.2 L'indicateur de présence (IfPresent)

L'indicateur de présence signale qu'un appariement peut être effectué si un champ facultatif est présent (c'est-à-dire non omis). Cet attribut peut être utilisé avec tous les mécanismes d'appariement, à condition que le type soit déclaré facultatif.

Un champ de modèle qui utilise l'indicateur de présence s'apparie avec le champ entrant correspondant si, et seulement si, ce champ entrant s'apparie conformément au mécanisme d'appariement associé, ou si ce champ entrant est absent. Par exemple:

```
template Mymessage:MyTemplate:=
{    :
     field2 := "abcd" ifpresent,  // Correspond à "abcd" s'il n'y a pas d'omission
     :
     :
}
```

NOTE – La variable *AnyValueOrNone* possède exactement la même signification que: **? ifpresent**.

C.1.5 Séquences de caractères d'appariement

Les séquences de caractères peuvent être utilisées dans des modèles pour définir le format d'une chaîne de caractères à recevoir. Les séquences de caractères peuvent être utilisées pour apparier des valeurs de type **charstring** et **universal charstring**. En plus des caractères littéraux, les séquences de caractères permettent d'utiliser les métacaractères **?** et ***** pour indiquer respectivement un caractère quelconque et un nombre quelconque de caractères quelconques. Par exemple:

```
template charstring MyTemplate:= pattern "ab??xyz*";
```

Ce modèle s'apparierait avec toute chaîne de caractères composée des caractères 'ab', suivis de deux caractères quelconques, suivis des caractères 'xyz', suivis d'un nombre quelconque de caractères quelconques.

S'il est nécessaire d'interpréter littéralement un certain métacaractère, celui-ci doit être précédé du métacaractère '\'. Par exemple:

```
template charstring MyTemplate:= pattern "ab?\?xyz*";
```

Ce modèle s'apparierait avec toute chaîne de caractères composée des caractères 'ab', suivis d'un caractère quelconque, suivi des caractères '?xyz', suivis d'un nombre quelconque de caractères quelconques.

En plus des valeurs de chaîne directes, il est possible d'utiliser, à l'intérieur de l'instruction de structure, des références à des modèles existants, à des constantes existantes ou à des variables existantes. La référence doit se réduire à un des types de chaîne de caractères et à plusieurs de ces types. Par exemple:

```
const charstring MyString:= "ab?";
template charstring MyTemplate:= pattern MyString;
```

Ce modèle s'apparierait avec toute chaîne de caractères composée des caractères 'ab' suivis d'un caractère quelconque. En effet, toute chaîne de caractères faisant suite au mot clé **pattern,** soit explicitement ou par référence, sera interprétée conformément aux règles définies dans ce paragraphe.

L'instruction de structure permet également d'utiliser l'opérateur de concaténation et, en cas de chaîne de caractères universels, d'utiliser la production quadruple pour spécifier un seul caractère. Par exemple:

```
const charstring MyString:= "ab?";
template universal charstring MyTemplate:= pattern MyString
     & "de" & (1, 1, 13, 7);
```

Ce modèle s'apparierait avec toute chaîne de caractères composée des caractères 'ab' suivis d'un caractère quelconque, suivi des caractères 'de', suivis du caractère du jeu ISO/CEI 10646-1 avec groupe = 1, plan = 1, rangée = 65 et cellule = 7.

Annexe D

Fonctions de notation TTCN-3 prédéfinies

D.1 Fonctions de notation TTCN-3 prédéfinies

Cette annexe définit les fonctions TTCN-3 prédéfinies.

D.1.1 Conversion d'entier en caractère

```
int2char(integer value) return char
```

Cette fonction convertit une valeur **integer** dans l'étendue de 0 .. 127 (codage sur 8 bits) en valeur de caractère du jeu de la Rec. UIT-T T.50 et ISO/CEI 646 [5]. La valeur d'entier décrit le codage sur 8 bits du caractère.

La fonction renvoie –1 si la valeur de l'argument est un nombre négatif ou supérieur à 127.

D.1.2 Conversion de caractère en entier

```
char2int(char value) return integer
```

Cette fonction convertit une valeur **char** du jeu de la Rec. UIT-T T.50 et ISO/CEI 646 [5] en entier dans l'étendue de 0 .. 127. La valeur d'entier décrit le codage sur 8 bits du caractère.

D.1.3 Conversion d'entier en caractère universel

int2unichar(**integer** value) **return universal char**

Cette fonction convertit une valeur **integer** dans l'étendue de 0 .. 268435455 (codage sur 32 bits) en valeur de caractère du jeu ISO/CEI 10646-1 [6]. La valeur d'entier décrit le codage sur 32 bits du caractère.

La fonction renvoie –1 si la valeur de l'argument est un nombre négatif ou supérieur à 268435455.

D.1.4 Conversion de caractère universel en entier

unichar2int(**universal char** value) **return integer**

Cette fonction convertit une valeur **univeral char** du jeu ISO/CEI 10646-1 [6] en valeur d'entier dans l'étendue de 0 .. 268435455. La valeur d'entier décrit le codage sur 32 bits du caractère.

D.1.5 Conversion de chaîne binaire en entier

bit2int(**bitstring** value) **return integer**

Cette fonction convertit une valeur isolée de type **bitstring** en valeur isolée de type **integer**.

Aux fins de cette conversion, une chaîne binaire doit être interprétée comme une valeur d'entier positif de base 2. Le bit de droite est de poids faible et le bit de gauche est de poids fort. Les bits 0 et 1 représentent respectivement les valeurs décimales 0 et 1.

D.1.6 Conversion de chaîne hexadécimale en entier

hex2int(**hexstring** value) **return integer**

Cette fonction convertit une valeur isolée **hexstring** en valeur isolée **integer**.

Aux fins de cette conversion, une chaîne **hexstring** doit être interprétée comme une valeur positive en base 16 de type **integer**. Le chiffre hexadécimal de droite est de poids faible, le chiffre hexadécimal de gauche est de poids fort. Les chiffres hexadécimaux 0 .. F représentent respectivement les valeurs décimales 0 .. 15.

D.1.7 Conversion de chaîne d'octets en entier

oct2int(**octetstring** value) **return integer**

Cette fonction convertit une valeur isolée **octetstring** en valeur isolée **integer**.

Aux fins de cette conversion, une chaîne **hexstring** doit être interprétée comme une valeur positive en base 16 de type **integer**. Le chiffre hexadécimal de droite est de poids faible, le chiffre hexadécimal de gauche est de poids fort. Le nombre de chiffres hexadécimaux fourni doit être multiple de 2 car un octet se compose de deux chiffres hexadécimaux. Les chiffres hexadécimaux 0 .. F représentent respectivement les valeurs décimales 0 .. 15.

D.1.8 Conversion de chaîne de caractères en entier

str2int(**charstring** value) **return integer**

Cette fonction convertit une valeur de chaîne **charstring** représentant une valeur de type **integer** en valeur équivalente **integer**. Si la chaîne ne représente pas une valeur d'entier valide, la fonction renvoie la valeur zéro (0).

Exemples:

str2int("66") renverra la valeur d'entier 66

str2int("-66") renverra la valeur d'entier -66

str2int("abc") renverra la valeur d'entier 0

str2int("0") renverra la valeur d'entier 0

D.1.9 Conversion d'entier en chaîne binaire

int2bit(**integer** value, length) **return bitstring**

Cette fonction convertit une valeur isolée **integer** en valeur isolée **bitstring**. La chaîne résultante a une longueur indiquée en bits par l'attribut length.

Aux fins de cette conversion, une chaîne **bitstring** doit être interprétée comme une valeur positive en base 2 de type **integer**. Le bit de droite est de poids faible, le bit de gauche est de poids fort. Les bits 0 et 1 représentent respectivement les valeurs décimales 0 et 1. Si la conversion fournit une valeur de chaîne avec moins de bits que spécifié dans le paramètre length , alors la chaîne binaire doit être complétée à gauche avec des zéros. Une erreur de test élémentaire doit se produire si la valeur est négative ou si la chaîne binaire résultante contient plus de bits que spécifié dans le paramètre length.

D.1.10 Conversion d'entier en chaîne hexadécimale

int2hex(**integer** value, length) **return hexstring**

Cette fonction convertit une valeur isolée **integer** en valeur isolée **hexstring**. La chaîne résultante a une longueur indiquée en chiffres hexadécimaux par l'attribut length.

Aux fins de cette conversion, une chaîne **hexstring** doit être interprétée comme une valeur positive en base 16 de type **integer**. Le chiffre hexadécimal de droite est de poids faible, le chiffre hexadécimal de gauche est de poids fort. Les chiffres hexadécimaux 0 .. F représentent respectivement les valeurs décimales 0 .. 15. Si la conversion fournit une valeur de chaîne avec moins de chiffres hexadécimaux que spécifié dans le paramètre length , alors la chaîne **hexstring** doit être complétée à gauche avec des zéros. Une erreur de test élémentaire doit se produire si la valeur est négative ou si la chaîne résultante **hexstring** contient plus de chiffres hexadécimaux que spécifié dans le paramètre length.

D.1.11 Conversion d'entier en chaîne d'octets

int2oct(**integer** value, length) **return octetstring**

Cette fonction convertit une valeur isolée **integer** en valeur isolée **octetstring**. La chaîne résultante a une longueur indiquée en octets par l'attribut length.

Aux fins de cette conversion, une chaîne **octetstring** doit être interprétée comme une valeur positive en base 16 de type **integer**. Le chiffre hexadécimal de droite est de poids faible, le chiffre hexadécimal de gauche est de poids fort. Le nombre de chiffres hexadécimaux fourni doit être multiple de 2 car un octet se compose de deux chiffres hexadécimaux. Les chiffres hexadécimaux 0 .. F représentent respectivement les valeurs décimales 0 .. 15. Si la conversion fournit une valeur de chaîne avec moins de chiffres hexadécimaux que spécifié dans le paramètre length , alors la chaîne **hexstring** doit être complétée à gauche avec des zéros. Une erreur de test élémentaire doit se produire si la valeur est négative ou si la chaîne résultante **hexstring** contient plus de chiffres hexadécimaux que spécifié dans le paramètre length.

D.1.12 Conversion d'entier en chaîne de caractères

int2str(**integer** value) **return charstring**

Cette fonction convertit la valeur d'entier en sa chaîne équivalente (la base de la chaîne de retour est toujours décimale).

Exemples:

int2str(66) renverra la valeur **charstring** "66"
int2str(-66) renverra la valeur **charstring** "-66"
int2str(0) renverra la valeur **integer** "0"

D.1.13 Longueur d'un type de chaîne

```
lengthof(any_string_type value) return integer
```

Cette fonction renvoie la longueur d'une valeur qui est du type **bitstring**, **hexstring**, **octetstring**, ou toute chaîne de caractères. Les unités de longueur pour chaque type chaîne sont définies dans le Tableau 4.

Exemple:

```
lengthof('010'B) // renvoie 3
lengthof('F3'H)  // renvoie 2
lengthof('F2'O)  // renvoie 1
lengthof ("Length_of _Example") // renvoie 17
```

D.1.14 Nombre d'éléments dans un type structuré

```
sizeof(structured_type value) return integer
```

Cette fonction renvoie le nombre réel d'éléments d'une structure **record**, **record of**, **set**, **set of**, **template** ou d'une séquence tabulaire.

```
// Soit
type record MyPDU
     { boolean field1,
       integer field2
     }

// alors
sizeof(MyPDU)
// renvoie 2
```

D.1.15 La fonction de présence (IsPresent)

```
ispresent(any_type value) return boolean
```

Cette fonction renvoie la valeur **true** si, et seulement si, la valeur du champ cité en référence est présente dans l'instance réelle de l'objet de données cité en référence. L'argument de **ispresent** doit être une référence à un champ situé à l'intérieur d'un objet de données défini comme étant **optional**.

```
// Soit
type record MyRecord
     { boolean field1 optional,
       integer field2
     }
// et soit MyPDU un modèle de type MyRecord
// et received_PDU également de type MyRecord
// alors
MyPort.receive(MyPDU) -> value received_PDU
ispresent(received_PDU.field1)
// renvoie la valeur "true" si field1 est présent
// dans l'instance réelle de MyPDU
```

D.1.16 La fonction de sélection (IsChosen)

```
ischosen(any_type value) return boolean
```

Cette fonction renvoie la valeur **true** si, et seulement si, la référence de l'objet de données spécifie la variante du type **union** qui est réellement sélectionnée pour un objet de données.

Exemple:

```
// Soit

type union MyUnion
     { PDU_type1 p1,
       PDU_type2 p2,
       PDU_type  p3
     }

// et soit MyPDU un modèle de type MyUnion
// et received_PDU également de type MyUnion
// alors
MyPort.receive(MyPDU) -> value received_PDU
ischosen(received_PDU.p2)
// renvoie la valeur "true" si l'instance réelle de MyPDU contient une unité PDU
// du type PDU_type2
```

Annexe E

Utilisation d'autres types de données en notation TTCN-3

E.1 Utilisation de la notation ASN.1 en notation TTCN-3

Cette annexe définit l'utilisation facultative de la notation ASN.1 avec la notation TTCN-3.

La notation TTCN-3 fournit une interface nette pour l'utilisation, dans des modules TTCN-3, d'une version ASN.1 1997 (telle que définie dans la série de Recommandations UIT-T X.680 [7], [8], [9], [10]). Après importation dans un module TTCN-3 l'identificateur linguistique de la version 1997 de l'ASN.1 doit être "ASN.1:1997".

Lorsque la notation ASN.1 est utilisée avec la notation TTCN-3, les mots clés énumérés dans le Tableau E.1 ne doivent pas être utilisés comme identificateurs dans un module TTCN-3. Les mots clés ASN.1 doivent suivre les prescriptions de la Rec. X.680 [7].

Tableau E.1/Z.140 – Liste des mots clés ASN.1

ABSENT	**EMBEDDED**	**INTERSECTION**	**SEQUENCE**
ABSTRACT-SYNTAX	**END**	**Iso10646string**	**SET**
ALL	**ENUMERATED**	**MAX**	**SIZE**
APPLICATION	**EXCEPT**	**MIN**	**STRING**
AUTOMATIC	**EXPLICIT**	**MINUS-INFINITY**	**SYNTAX**
BEGIN	**EXPORTS**	**NULL**	**T61String**
BIT	**EXTERNAL**	**NumericString**	**TAGS**
BMPSTRING	**FALSE**	**OBJECT**	**TeletexString**
BOOLEAN	**FROM**	**ObjectDescriptor**	**TRUE**
BY	**GeneralizedTime**	**OCTET**	**TYPE-IDENTIFIER**
CHARACTER	**GeneralString**	**OF**	**UNION**
CHOICE	**IA5String**	**OPTIONAL**	**UNIQUE**
CLASS	**IDENTIFIER**	**PDV**	**UNIVERSAL**
COMPONENT	**IMPLICIT**	**PLUS-INFINITY**	**UniversalString**
COMPONENTS	**IMPORTS**	**PRESENT**	**UTCTime**
CONSTRAINED	**INCLUDES**	**PrintableString**	**VideotexString**
DEFAULT	**INSTANCE**	**PRIVATE**	**VisibleString**
DEFINITIONS	**INTEGER**	**REAL**	**WITH**

E.1.1 Equivalents de type ASN.1 et de type TTCN-3

Les types ASN.1 énumérés dans le Tableau E.2 sont considérés comme étant équivalents à leurs contreparties TTCN-3.

Tableau E.2/Z.140 – Liste des équivalents ASN.1 et TTCN-3

Le type ASN.1	se mappe sur l'équivalent TTCN-3
BOOLEAN	boolean
INTEGER	integer
REAL	float
OBJECT IDENTIFIER	objid
BIT STRING	bitstring
OCTET STRING	octetstring
SEQUENCE	record
SEQUENCE OF	record of
SET	set
SET OF	set of
ENUMERATED	enumerated
CHOICE	union

Les opérateurs, fonctions, mécanismes d'appariement, notations de valeur, etc., de la notation TTCN-3, qui peuvent être utilisés avec un type TTCN-3 indiqué dans le Tableau E.2, peuvent tous être utilisés également avec le type ASN.1 correspondant.

E.1.2 Types et valeurs de données ASN.1

Les types et valeurs ASN.1 peuvent être utilisés dans les modules TTCN-3. Les définitions ASN.1 sont faites au moyen d'un module ASN.1 distinct.

Exemple:

```
MyASN1module DEFINITIONS ::=
BEGIN
       Z::=   INTEGER              -- Simple définition de type


       BMessage::= SET             -- Définition de type ASN.1
       {
          name Name,
          title  VisibleString,
          date Date
       }

       johnValues Bmessage ::=     -- Définition de valeur ASN.1
       {
          name "John Doe",
          title  "Mr",
          date "April 12th"
       }
END
```

Le module ASN.1 doit être écrit selon la syntaxe de la série des Recommandations UIT-T X.680 [7], [8], [9] et [10]. Une fois déclarés, les types et valeurs ASN.1 peuvent être utilisés avec des modules TTCN-3 exactement de la même façon que les types et valeurs TTCN-3 issus d'autres modules TTCN-3 sont utilisés (c'est-à-dire que les définitions requises doivent être importées).

Exemple:

```
module MyTTCNModule
{
       import all from MyASN1module language "ASN.1:1997";

       const Bmessage MyTTCNConst:= johnValues;
}
```

NOTE – Les définitions ASN.1 autres que les types et valeurs (c'est-à-dire les classes d'objets informationnels ou les ensembles d'objets informationnels) ne sont pas directement accessibles à partir de la notation TTCN-3. De telles définitions doivent être réduites à un type ou à une valeur contenu dans le module ASN.1 avant de pouvoir être citées en référence de l'intérieur du module TTCN-3.

E.1.2.1 Portée des identificateurs ASN.1

Les identificateurs ASN.1 importés suivent les mêmes règles de portée que les types et valeurs TTCN-3 importés (voir § 5.4).

E.1.3 Paramétrisation en notation ASN.1

Il est permis de faire référence à des définitions de type et de valeur ASN.1 à partir du module TTCN-3. Toutes les définitions en notation ASN.1 et utilisées dans un module en notation TTCN-3 doivent cependant être paramétrées avec des paramètres réels (les types ou valeurs ouverts ne sont pas permis) et les paramètres réels fournis doivent pouvoir être résolus au moment de la compilation.

Le langage noyau de la notation TTCN-3 ne prend pas en charge la paramétrisation d'objets spécifiés uniquement en notation ASN.1. La paramétrisation spécifique de la notation ASN.1, qui implique des objets ne pouvant pas être définis directement dans le langage noyau TTCN-3, doit donc être résolue dans la partie ASN.1 avant d'être utilisée dans la notation TTCN-3. Les objets ASN.1 spécifiques sont les suivants:

a) ensembles de valeurs;

b) classes d'objets informationnels;

c) objets informationnels;

d) ensembles d'objets informationnels.

Par exemple, la notation suivante n'est pas autorisée parce qu'elle définit un type TTCN-3 qui prend comme paramètre réel un ensemble d'objets ASN.1.

```
MyASN1module DEFINITIONS ::=
BEGIN
     -- Définition de module ASN.1
     -- Définition de classe d'objets informationnels
     MESSAGE ::= CLASS { &msgTypeValueINTEGER UNIQUE,
                         &MsgFields}
     -- Définition d'objet informationnel
     setupMessage MESSAGE ::=            {    &msgTypeValue  1,
                  &MsgFields         OCTET STRING}
       setupAckMessage MESSAGE ::=      {    &msgTypeValue  2,
                  &MsgFields         BOOLEAN}
     -- Définition d'ensemble d'objets informationnels
     MyProtocol MESSAGE ::= { setupMessage | setupAckMessage}

     -- Type ASN.1 contraint par ensemble d'objets
     MyMessage{ MESSAGE : MsgSet} ::= SEQUENCE
     {
       code   MESSAGE.&msgTypeValue({ MsgSet}),
       Type   MESSAGE.&MsgFields({ MsgSet})
}
END

module MyTTCNModule
{
     // Définition de module TTCN-3
     import all from MyASN1module language "ASN.1:1997";

     // Type TTCN-3 illégal avec ensemble d'objets transmis comme paramètre
     type record Q(MESSAGE MyMsgSet) ::= { Z          field1,
                                 MyMessage(MyMsgSet)  field2}
}
```

Afin d'en faire une définition légale, le type ASN.1 supplémentaire MyMessage1 doit être défini comme indiqué ci-dessous. Cela résout la paramétrisation par ensemble d'objets informationnels et peut donc être directement utilisé dans le module TTCN-3.

```
MyASN1module DEFINITIONS ::=

BEGIN
     -- Définition de module ASN.1

     ...

     MyProtocol MESSAGE ::= { setupMessage | setupAckMessage}

     -- Type ASN.1 supplémentaire afin de supprimer la paramétrisation par
     -- ensemble d'objets
     MyMessage1 ::= MyMessage{ MyProtocol}
END

module MyTTCNModule
{
     // Définition de module TTCN-3
     import all from MyASN1module language "ASN.1:1997";
```

```
    // Type TTCN-3 légal sans ensemble d'objets transmis comme paramètre
    type record Q := { Z              field1,
             MyMessage1               field2}
}
```

E.1.4 Définition des types de message en notation ASN.1

En notation ASN.1, les messages sont définis au moyen de SEQUENCE (ou le cas échéant SET).

Exemple:

```
MyASN1module DEFINITIONS ::=
BEGIN
    -- Définition de module ASN.1

    MyMessageType ::= SEQUENCE
    {
      field1  Field1Type,
      field2  Field2Type    OPTIONAL, -- Ce champ peut être omis
      :
      fieldN  FieldNType
    }
END
```

Il est évident que le messages définis au moyen de la notation ASN.1 peuvent également être sous-structurés au moyen de SEQUENCE, SET, etc.

E.1.5 Définition des modèles de message ASN.1

Si des messages sont définis en ASN.1 au moyen, par exemple de SEQUENCE (ou le cas échéant SET), les messages réels, pour les événements de type **send** et **receive,** peuvent être spécifiés au moyen de la syntaxe de valeur ASN.1.

Exemple:

```
MyASN1module DEFINITIONS ::=
BEGIN
    -- Définition de module ASN.1

    -- La définition de message
MyMessageType::= SEQUENCE
    { field1 [1] IA5STRING,          // comme une chaîne de caractères TTCN-3
      field2 [2] INTEGER OPTIONAL,   // comme un entier TTCN-3
      field3 [4] Field3Type,         // comme un enregistrement TTCN-3
      field4 [5] Field4Type          // comme une séquence tabulaire TTCN-3
    }

    Field3Type::= SEQUENCE {field31 BIT STRING, field32 INTEGER, field33 OCTET
      STRING},
    Field4Type::= SEQUENCE OF BOOLEAN


    -- peut avoir la valeur suivante
    myValue MyMessageType::=
    {
      field1  "une chaîne",
      field2  123,
      field3  {field31 '11011'B, field32 456789, field33 'FF'O},
      field4  {true, false}
    }
END
```

E.1.5.1 Messages de réception ASN.1 utilisant la syntaxe de modèle TTCN-3

Les mécanismes d'appariement ne sont pas pris en charge dans la syntaxe ASN.1 de base. Si l'on souhaite par exemple utiliser des mécanismes d'appariement avec un message de réception ASN.1, la syntaxe TTCN-3 pour les modèles de réception doit alors être utilisée. Noter que cette syntaxe comporte des références de composant afin de pouvoir faire référence aux composants individuels contenus dans des éléments ASN.1 de type SEQUENCE, SET, etc.

Exemple:

```
import type myMessageType from MyASN1module language "ASN.1:1997";

// un modèle de message au moyen de mécanismes d'appariement à l'intérieur de la
// notation TTCN-3 pourrait être:
template myMessageType  MyValue:=
{
     field1 :=      "A"<?>"tr"<*>"g",
     field2 :=      *,
     field3.field31 :=  '110??'B,
     field3.field32 :=  ?,
     field3.field33 :=  'F?'O,
     field4.[0] :=      true,
     field4.[1] :=      false
}

// la syntaxe suivante est également valide
template myMessageType  MyValue:=
{
     field1 := "A"<?>"tr"<*>"g",          // chaîne avec structure générique
     field2 := *,                         // tout entier ou aucun
     field3 := {'110??'B, ?, 'F?'O},
     field4 := {?, false}
}
```

E.1.5.2 Séquencement des champs de modèle

Lorsque des modèles TTCN-3 sont utilisés pour des types ASN.1, la signification de l'ordre des champs dans ces modèles dépendra du type de construction ASN.1 utilisée pour définir le type de message. Par exemple, si l'on utilise la construction SEQUENCE ou SEQUENCE OF, les champs de message doivent être envoyés ou appariés dans l'ordre spécifié dans le modèle. Si la construction SET ou SET OF est utilisée, les champs de message peuvent être envoyés ou appariés dans un ordre quelconque.

E.1.6 Informations de codage

La notation TTCN-3 permet de faire référence à des règles de codage et à des variantes à l'intérieur des règles de codage à associer à divers éléments linguistiques de la notation TTCN-3. Il est également possible de définir des codages invalides. Ces informations de codage sont spécifiées au moyen de l'instruction **with** conformément à la syntaxe ci-après:

Exemple:

```
module MyModule
{    :
     import type myMessageType from MyASN1module language "ASN.1:1997" with
       {encode:= "PER:1997"}
             // Toutes les instances de MyMessageType devront être codées au
             // moyen des règles PER:1997

} with {encode "BER:1997"}  // Le codage par défaut pour tout le monde
                            // (suite de tests) est BER:1997
```

E.1.6.1 Attributs de codage ASN.1

Les chaînes suivantes sont les attributs de codage prédéfinis (normalisés) pour la notation ASN.1:

a) "BER:1997" signifie: codé selon la Rec. UIT-T X.690 (BER) [1997];

b) "CER:1997" signifie: codé selon la Rec. UIT-T X.690 (CER) [1997];

c) "DER:1997" signifie: codé selon la Rec. UIT-T X.690 (DER) [1997].

d) "PER-BASIC-UNALIGNED:1997" signifie: codé selon la Rec. UIT-T X.691 (règles PER non alignées) [1997];

e) "PER-BASIC-ALIGNED:1997" signifie: codé selon la Rec. UIT-T X.691 (règles PER alignées) [1997];

f) "PER-CANONICAL-UNALIGNED:1997" signifie: codé selon la Rec. UIT-T X.691 (règles PER canoniques non alignées) [1997];

g) "PER-CANONICAL-ALIGNED:1997" signifie: codé selon la Rec. UIT-T X.691 (règles PER canoniques alignées) [1997].

SÉRIES DES RECOMMANDATIONS UIT-T

Série A	Organisation du travail de l'UIT-T
Série B	Moyens d'expression: définitions, symboles, classification
Série C	Statistiques générales des télécommunications
Série D	Principes généraux de tarification
Série E	Exploitation générale du réseau, service téléphonique, exploitation des services et facteurs humains
Série F	Services de télécommunication non téléphoniques
Série G	Systèmes et supports de transmission, systèmes et réseaux numériques
Série H	Systèmes audiovisuels et multimédias
Série I	Réseau numérique à intégration de services
Série J	Réseaux câblés et transmission des signaux radiophoniques, télévisuels et autres signaux multimédias
Série K	Protection contre les perturbations
Série L	Construction, installation et protection des câbles et autres éléments des installations extérieures
Série M	RGT et maintenance des réseaux: systèmes de transmission, circuits téléphoniques, télégraphie, télécopie et circuits loués internationaux
Série N	Maintenance: circuits internationaux de transmission radiophonique et télévisuelle
Série O	Spécifications des appareils de mesure
Série P	Qualité de transmission téléphonique, installations téléphoniques et réseaux locaux
Série Q	Commutation et signalisation
Série R	Transmission télégraphique
Série S	Equipements terminaux de télégraphie
Série T	Terminaux des services télématiques
Série U	Commutation télégraphique
Série V	Communications de données sur le réseau téléphonique
Série X	Réseaux de données et communication entre systèmes ouverts
Série Y	Infrastructure mondiale de l'information et protocole Internet
Série Z	**Langages et aspects généraux logiciels des systèmes de télécommunication**

Imprimé en Suisse
Genève, 2002