Union internationale des télécommunications

UIT-T

SECTEUR DE LA NORMALISATION
DES TÉLÉCOMMUNICATIONS
DE L'UIT

Z.143

(03/2006)

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

Techniques de description formelle – Notation de test et de commande de test

---

Notation de test et de commande de test version 3 (TTCN-3): sémantique opérationnelle

Recommandation UIT-T Z.143

UIT-T

Union internationale des télécommunications

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
Langage étendu de définition d'objets	Z.130–Z.139
Notation de test et de commande de test	**Z.140–Z.149**
Notation de prescriptions d'utilisateur	Z.150–Z.159
LANGAGES DE PROGRAMMATION	
CHILL: le langage de haut niveau 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.349
Interfaces homme-machine orientées données	Z.350–Z.359
Interfaces homme-machine pour la gestion des réseaux de télécommunication	Z.360–Z.379
QUALITÉ	
Qualité des logiciels de télécommunication	Z.400–Z.409
Aspects qualité des Recommandations relatives aux protocoles	Z.450–Z.459
MÉTHODES	
Méthodes de validation et d'essai	Z.500–Z.519
INTERGICIELS	
Environnement de traitement réparti	Z.600–Z.609

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

Recommandation UIT-T Z.143

Notation de test et de commande de test version 3 (TTCN-3): sémantique opérationnelle

Résumé

La présente Recommandation définit la sémantique opérationnelle de la notation de test et de commande de test version 3 (TTCN-3, *testing and test control notation version 3*). La sémantique opérationnelle est nécessaire pour pouvoir interpréter sans ambiguïté les spécifications en notation TTCN-3. La présente Recommandation est fondée sur le langage noyau de la notation TTCN-3, qui est défini dans la Rec. UIT-T Z.140.

Source

La Recommandation UIT-T Z.143 a été approuvée le 16 mars 2006 par la Commission d'études 17 (2005-2008) de l'UIT-T selon la procédure définie dans la Recommandation UIT-T A.8.

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.

Le respect de cette Recommandation se fait à titre volontaire. Cependant, il se peut que la Recommandation contienne certaines dispositions obligatoires (pour assurer, par exemple, l'interopérabilité et l'applicabilité) et considère que la Recommandation est respectée lorsque toutes ces dispositions sont observées. Le futur d'obligation et les autres moyens d'expression de l'obligation comme le verbe "devoir" ainsi que leurs formes négatives servent à énoncer des prescriptions. L'utilisation de ces formes ne signifie pas qu'il est obligatoire de respecter la Recommandation.

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 des renseignements les plus récents, il est vivement recommandé aux développeurs de consulter la base de données des brevets du TSB sous http://www.itu.int/ITU-T/ipr/.

© UIT 2007

Tous droits réservés. Aucune partie de cette publication ne peut être reproduite, par quelque procédé que ce soit, sans l'accord écrit préalable de l'UIT.

TABLE DES MATIÈRES

		Page
1	Domaine d'application	1
2	Références	1
3	Définitions et abréviations	1
	3.1 Définitions	1
	3.2 Abréviations	1
4	Introduction	1
5	Structure de la présente Recommandation	2
6	Restrictions	2
7	Remplacement des formes abrégées	2
	7.1 Ordre des étapes de remplacement	3
	7.2 Remplacement des constantes globales et des paramètres de module	3
	7.3 Imbrication des simples opérations de réception dans des instructions alt	3
	7.4 Imbrication des appels de variante autonomes dans des instructions alt	4
	7.5 Remplacement des instructions interleave	4
	7.6 Remplacement des opérations trigger	17
8	Sémantique de la notation TTCN-3 sur la base de graphes de flux	18
	8.1 Graphes de flux	18
	8.2 Représentation du comportement TTCN-3 sous forme de graphes de flux	23
	8.3 Définition des états concernant les modules TTCN-3	28
	8.4 Messages, appels de procédure, réponses et exceptions	37
	8.5 Enregistrements d'appel pour les fonctions, les variantes et les tests élémentaires	39
	8.6 Procédure d'évaluation d'un module TTCN-3	40
9	Segments de graphe de flux pour les constructions TTCN-3	41
	9.1 Instruction action	42
	9.2 Instruction activate	42
	9.3 Instruction alt	43
	9.4 Appel de variante	49
	9.5 Instruction assignment	49
	9.6 Opération call	49
	9.7 Opération catch	55
	9.8 Opération check	55
	9.9 Opération clear applicable aux ports	59
	9.10 Opération connect	59
	9.11 Définition d'une constante	60
	9.12 Opération create	60
	9.13 Instruction deactivate	61
	9.14 Opération disconnect	63
	9.15 Instruction do-while	63
	9.16 Opération done applicable aux composants	64
	9.17 Instruction execute	65
	9.18 Expression	68
	9.18b Segment de graphe de flux <dynamic-error>	70
	9.19 Segment de graphe de flux <finalize-component-init>	70
	9.20 Segment de graphe de flux <init-component-scope>	71
	9.21 Segment de graphe de flux <parameter-handling>	71
	9.22 Segment de graphe de flux <statement-block>	72
	9.23 Instruction for	73
	9.24 Appel de fonction	74
	9.25 Opération getcall	78
	9.26 Opération getreply	78
	9.27 Opération getverdict	79
	9.28 Instruction goto	79

Page

9.29 Instruction if-else 80
9.30 Instruction label 80
9.31 Instruction log 81
9.32 Opération map 81
9.33 Opération mtc 82
9.34 Déclaration de port 82
9.35 Opération raise 83
9.36 Opération read applicable aux temporisations 85
9.37 Opération receive 86
9.38 Instruction repeat 89
9.39 Opération reply 89
9.40 Instruction return 91
9.41 Opération running applicable aux composants 93
9.42 Opération running applicable aux temporisations 96
9.43 Opération self 97
9.44 Opération send 98
9.45 Opération setverdict 100
9.46 Opération start applicable aux composants 101
9.47 Opération start applicable aux ports 103
9.48 Opération start applicable aux temporisations 103
9.49 Opération stop applicable aux composants 105
9.50 Instruction stop applicable à l'exécution 109
9.51 Opération stop applicable aux ports 110
9.52 Opération stop applicable aux temporisations 111
9.53 Opération system 111
9.54 Déclaration d'une temporisation 112
9.55 Opération timeout applicable aux temporisations 113
9.56 Opération unmap 114
9.57 Déclaration d'une variable 115
9.58 Instruction while 116

10 Listes des composantes de la sémantique opérationnelle 117
10.1 Fonctions et états 117
10.2 Mots clés spéciaux 119
10.3 Graphes de flux des descriptions de comportement TTCN-3 120
10.4 Segments de graphe de flux 120

Recommandation UIT-T Z.143

Notation de test et de commande de test version 3 (TTCN-3): sémantique opérationnelle

1 Domaine d'application

La présente Recommandation définit la sémantique opérationnelle de la notation TTCN-3. Elle est fondée sur le langage noyau de la notation TTCN-3, qui est défini dans la Rec. UIT-T Z.140 [1].

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 de la présente Recommandation. Au moment de la publication, les éditions indiquées étaient en vigueur. Les Recommandations et autres références étant sujettes à révision, les utilisateurs de la présente Recommandation sont invités à rechercher la possibilité d'appliquer les éditions les plus récentes des Recommandations et autres références énumérées ci-dessous. Une liste des Recommandations UIT-T en vigueur est publiée périodiquement. La référence à un document figurant dans la présente Recommandation ne donne pas à ce document en tant que tel le statut de Recommandation.

[1] Recommandation UIT-T Z.140 (2006), *Notation de test et de commande de test version 3 (TTCN-3): langage noyau.*

3 Définitions et abréviations

3.1 Définitions

Dans la présente Recommandation, les termes et définitions figurant dans la Rec. UIT-T Z.140 [1] s'appliquent.

3.2 Abréviations

La présente Recommandation utilise les abréviations suivantes:

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

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

IDL langage de description d'interface (*interface description language*)

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

SUT système sous test (*system under test*)

TTCN notation de test et de commande de test (*testing and test control notation*)

4 Introduction

Le présent paragraphe définit la signification du comportement TTCN-3 de manière intuitive et non ambiguë. La sémantique opérationnelle n'est pas censée être formelle, d'où la capacité très limitée de réaliser des preuves mathématiques sur la base de cette sémantique.

Cette sémantique opérationnelle donne une vue de l'exécution d'un module TTCN sur la base d'états. On définit différents types d'état et on décrit la signification des différentes constructions TTCN-3

1) en utilisant des informations d'état pour définir les conditions préalables à l'exécution d'une construction;
2) en définissant la manière dont l'exécution d'une construction fera passer à un autre état.

La sémantique opérationnelle est restreinte à la signification du comportement TTCN-3, autrement dit les fonctions, les variantes, les tests élémentaires, la commande de module et les constructions de langage pour la définition d'un comportement de test, par exemple les opérations **`send`** et **`receive`**, les instructions **`if-else`** ou **`while`**. Pour expliquer la signification de certaines constructions TTCN-3, on les remplace par d'autres constructions de langage. Par exemple, l'instruction **`interleave`**, qui correspond à la forme abrégée d'une série d'instructions **`alt`** imbriquées, est remplacée par cette série lorsqu'il s'agit d'expliquer sa signification.

Dans la plupart des cas, la définition de la sémantique d'un langage est fondée sur une arborescence de la syntaxe abstraite du code qui doit être décrit. Cette sémantique ne repose pas sur une arborescence de la syntaxe abstraite mais sur une représentation graphique des descriptions de comportement TTCN-3 sous la forme de graphes de flux. Un graphe de flux décrit le flux de commande d'une fonction, d'une variante, d'un test élémentaire ou de la commande de module. Le mappage des descriptions de comportement TTCN-3 vers des graphes de flux est immédiat.

NOTE – Le mappage des instructions TTCN-3 vers des graphes de flux est une étape non formelle et n'est pas défini au moyen des règles BNF énoncées dans la Rec. UIT-T Z.140 [1]. En effet, les règles BNF ne sont pas optimales pour un mappage intuitif car plusieurs règles de sémantique statique sont codées sous forme de règles BNF afin de pouvoir procéder à des contrôles de la sémantique statique pendant la vérification de la syntaxe.

5 Structure de la présente Recommandation

La présente Recommandation comporte quatre parties:

1) la première partie (voir § 6) décrit les restrictions de la sémantique opérationnelle, autrement dit les questions relatives à la sémantique qui ne sont pas traitées dans la présente Recommandation;
2) la deuxième partie (voir § 7) définit la signification des notations abrégées et des macronotations TTCN-3 en les remplaçant par d'autres constructions de langage TTCN-3. La réalisation de ces remplacements dans un module TTCN-3 peut être considérée comme une étape de prétraitement avant que le module puisse être interprété conformément à la description de la sémantique opérationnelle qui suit;
3) la troisième partie (voir § 8) décrit la sémantique opérationnelle de la notation TTCN-3 au moyen de l'interprétation de graphes de flux et de la modification d'états;
4) la quatrième partie (voir § 9) spécifie le mappage des différentes instructions TTCN-3 vers des segments de graphe de flux, qui constituent les éléments de base des graphes de flux représentant les fonctions, les variantes, les tests élémentaires et la commande de module.

6 Restrictions

La sémantique opérationnelle porte uniquement sur les aspects de la notation TTCN-3 liés au comportement, autrement dit elle décrit la signification des instructions et des opérations. Aucune sémantique n'est définie:

a) pour les aspects de la notation TTCN-3 liés aux données (par exemple le codage, le décodage et l'utilisation des données importées à partir de spécifications non TTCN-3).
b) pour le mécanisme de regroupement. En ce qui concerne le regroupement, qui se rapporte à la partie définitions d'un module TTCN-3, il n'y a pas d'aspects liés au comportement.
c) pour l'instruction **`import`**. L'importation de définitions doit se faire dans la partie définitions d'un module TTCN-3. La sémantique opérationnelle traite les définitions importées comme si elles étaient définies dans le module d'importation.
d) pour le paramétrage des ports.

7 Remplacement des formes abrégées

Les formes abrégées doivent être développées sous la forme des définitions complètes correspondantes au niveau textuel avant que la sémantique opérationnelle puisse être utilisée pour l'explication du comportement TTCN-3.

Les formes abrégées TTCN-3 sont les suivantes:

- listes de déclarations de paramètre de module, de constante et de variable du même type et listes de déclarations de temporisation;
- opérations de réception autonomes;
- appels de variante autonomes;
- opérations **`trigger`**;
- instructions **`return`** et **`stop`** manquantes à la fin des définitions de fonction et de test élémentaire;
- instructions **`stop`** applicables à l'exécution manquantes; et
- instructions **`interleave`**.

Outre le traitement des formes abrégées, la sémantique opérationnelle nécessite un traitement spécial des paramètres de module et des constantes globales, c'est-à-dire des constantes qui sont définies dans la partie définitions d'un module. Toutes les références à des paramètres de module et à des constantes globales doivent être remplacées par des valeurs concrètes. Autrement dit, on suppose que la valeur des paramètres de module et des constantes globales peut être déterminée avant que la sémantique opérationnelle s'applique.

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 mais il ne s'agit pas d'une ligne directrice pour l'implémentation d'un compilateur TTCN-3.

NOTE 2 – La sémantique opérationnelle traite les paramètres et les constantes locales des composants de test, des tests élémentaires, des fonctions et de la commande de module comme des variables. Le mauvais usage de constantes locales ou de paramètres **in**, **out** et **inout** doit être vérifié statiquement.

7.1 Ordre des étapes de remplacement

Les remplacements textuels des formes abrégées, des constantes globales et des paramètres de module doivent se faire dans l'ordre suivant:

1) remplacement des listes de déclarations de paramètre de module, de constante, de variable et de temporisation par des déclarations individuelles;
2) remplacement des constantes globales et des paramètres de module par des valeurs concrètes;
3) imbrication des opérations de réception autonomes dans des instructions **alt**;
4) imbrication des appels de variante autonomes dans des instructions **alt**;
5) développement des instructions **interleave**;
6) remplacement de toutes les opérations **trigger** par des opérations **receive** et des instructions **repeat** équivalentes;
7) ajout de **return** à la fin des fonctions sans instruction **return**, ajout d'opérations **self**.**stop** à la fin des définitions de test élémentaire sans instruction stop;
8) ajout de **stop** à la fin d'une partie commande de module sans instruction stop.

NOTE – Si on ne respecte pas cet ordre des étapes de remplacement, le résultat des remplacements ne représentera pas le comportement défini.

7.2 Remplacement des constantes globales et des paramètres de module

Les constantes déclarées dans la partie définitions d'un module sont globales pour la commande de module et pour tous les composants de test qui sont créés pendant 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 effectives 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, l'expression doit être évaluée. Le résultat de l'évaluation doit alors remplacer toutes les références à la constante ou au paramètre de module.

7.3 Imbrication des simples opérations de réception dans des instructions alt

Les opérations de réception TTCN-3 sont les suivantes: **receive**, **trigger**, **getcall**, **getreply**, **catch**, **check**, **timeout** et **done**.

NOTE – Les opérations **receive**, **trigger**, **getcall**, **getreply**, **catch** et **check** s'appliquent à des ports et elles permettent de déterminer la succession des branches résultant de la réception de messages, d'appels de procédure, de réponses et d'exceptions. Les opérations **timeout** et **done** ne sont pas des opérations de réception réelles, mais elles peuvent être utilisées de la même manière que les opérations de réception, c'est-à-dire comme des alternatives dans des instructions **alt**. Par conséquent, la sémantique opérationnelle traite les opérations **timeout** et **done** comme des opérations de réception.

Une opération de réception peut être utilisée comme une instruction autonome dans une fonction, une variante ou un test élémentaire. L'opération **timeout** peut aussi être utilisée comme une instruction autonome dans la commande de module. En pareil cas, l'opération de réception est considérée comme la forme abrégée d'une instruction **alt** avec une seule alternative définie par l'opération de réception. Pour la sémantique opérationnelle, toute occurrence autonome d'opération de réception doit être remplacée par une instruction **alt** dans laquelle l'opération de réception est imbriquée.

EXEMPLE:

```
// L'occurrence autonome de
   :
   MyCL.trigger(MyType:?);
```

```
   :
// doit être remplacée par
   :
  alt {
     []   MyCL.trigger (MyType:?) { }
  }
   :

// ou
   :
  MyPTC.done;
   :

// doit être remplacé par
   :
  alt {
     []   MyPTC.done { }
  }
   :
```

7.4 Imbrication des appels de variante autonomes dans des instructions alt

La notation TTCN-3 permet d'appeler des variantes comme des fonctions dans des fonctions, des variantes, des tests élémentaires et la commande de module. La signification de l'appel autonome d'une variante est donnée par une instruction **alt** avec une seule branche qui appelle la variante. L'instruction **alt** est responsable de l'instantané qui est évalué dans la variante et de l'invocation du mécanisme par défaut si aucune des alternatives de la variante ne peut être choisie.

NOTE – Une variante utilisée dans la commande de module peut uniquement inclure des alternatives avec des opérations **timeout** et une branche **else**.

EXEMPLE:

```
// L'occurrence autonome de
   :
  myAltstep(MyPar1Val);
   :

// doit être remplacée par
   :
  alt {
     []   myAltstep(MyPar1Val) { }
  }
   :
```

7.5 Remplacement des instructions interleave

La signification d'une instruction **interleave** est définie par son remplacement par une série d'instructions **alt** imbriquées qui a la même signification. L'algorithme de construction du remplacement d'une instruction **interleave** est décrit dans le présent paragraphe. Le remplacement doit être opéré au niveau syntaxique.

Dans une instruction **interleave**, il est interdit:

1) d'utiliser les instructions de transfert de commande **for**, **while**, **do-while**, **goto**, **activate**, **deactivate**, **stop**, **repeat** et **return**;
2) d'appeler des variantes;
3) d'appeler des fonctions définies par l'utilisateur qui incluent des opérations de communication;
4) de protéger des branches de l'instruction **interleave** avec des expressions booléennes;
5) de spécifier des branches **else**.

Cela étant, toutes les instructions autonomes non mentionnées (par exemple assignment, **log**, **send** ou **reply**), les opérations **call** *bloquantes* et les instructions composites **interleave**, **if-else** et **alt** peuvent être utilisées dans une instruction **interleave**.

NOTE 1 – Les opérations **call** bloquantes et les instructions **if-else** peuvent être traitées comme les instructions autonomes si elles n'ont pas d'instructions **alt** imbriquées. Dans le cas d'instructions **alt** imbriquées, les alternatives contribuent à l'instruction **interleave** et nécessitent un traitement spécial. Dans un souci de simplicité, l'algorithme ci-dessous ne fait pas la distinction entre ces deux cas.

NOTE 2 – Les opérations **call** non bloquantes sont aussi autorisées dans les instructions **interleave**, elles sont considérées comme étant des instructions autonomes.

L'algorithme décrit dans le présent paragraphe fonctionne uniquement pour les instructions **interleave** sans instructions **interleave** imbriquées. Dans le cas d'une instruction **interleave** contenant des instructions **interleave** imbriquées, les instructions **interleave** imbriquées doivent être remplacées avant que l'algorithme puisse être appliqué.

NOTE 3 – Compte tenu des restrictions 1 à 5, il est toujours possible de trouver des remplacements finis en ce qui concerne les imbrications d'instructions **interleave**.

L'algorithme de remplacement opère sur une représentation graphique d'une instruction interleave et la transforme en une structure arborescente sémantiquement équivalente décrivant une série d'instructions **alt** imbriquées. Il faut donc une représentation graphique des instructions autonomes et des instructions composites **if-else**, **call** *bloquantes*, **alt** et **interleave**.

Une instruction autonome est décrite par un nœud avec l'instruction comme inscription. Une séquence d'instructions autonomes est décrite par un ensemble de nœuds raccordés par des arcs. Voir la Figure 1.

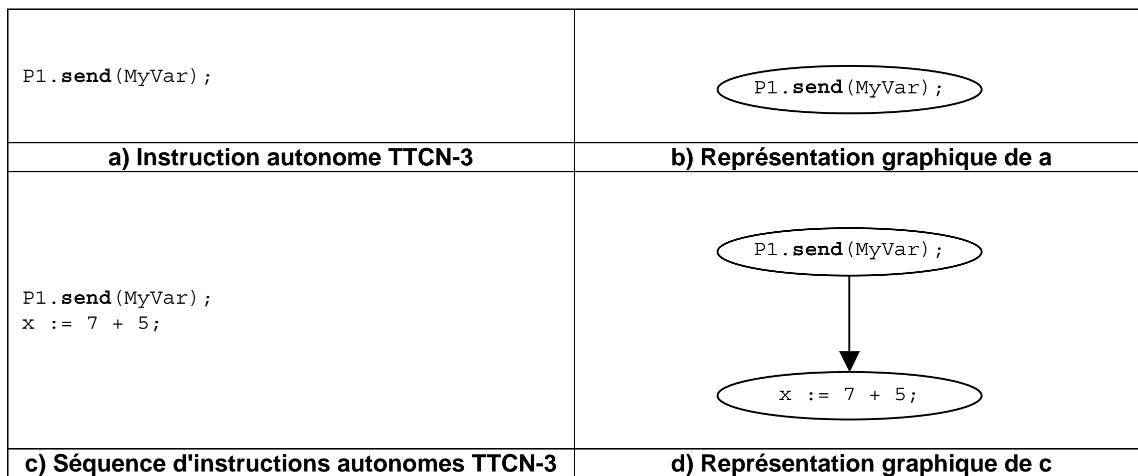


Figure 1/Z.143 – Représentation graphique d'instructions autonomes TTCN-3

La représentation graphique d'une instruction **if-else** est illustrée sur la Figure 2. Une instruction **if-else** est représentée par un nœud IF avec deux arcs allant à la première instruction des deux alternatives. Une instruction **if-else** sans branche ELSE est représentée de la même manière, si des instructions suivent l'instruction **if-else**. Dans ce cas, l'arc représentant la branche *else* va à la première instruction qui suit l'instruction **if-else**. Une instruction **if-else** sans branche ELSE et sans instruction qui suit est représentée par un nœud IF avec un seul arc.

NOTE 4 – Les inscriptions le long des arcs sur la Figure 1 sont uniquement données dans un souci de lisibilité. L'algorithme utilise uniquement la relation exprimée par l'arc et non l'inscription.

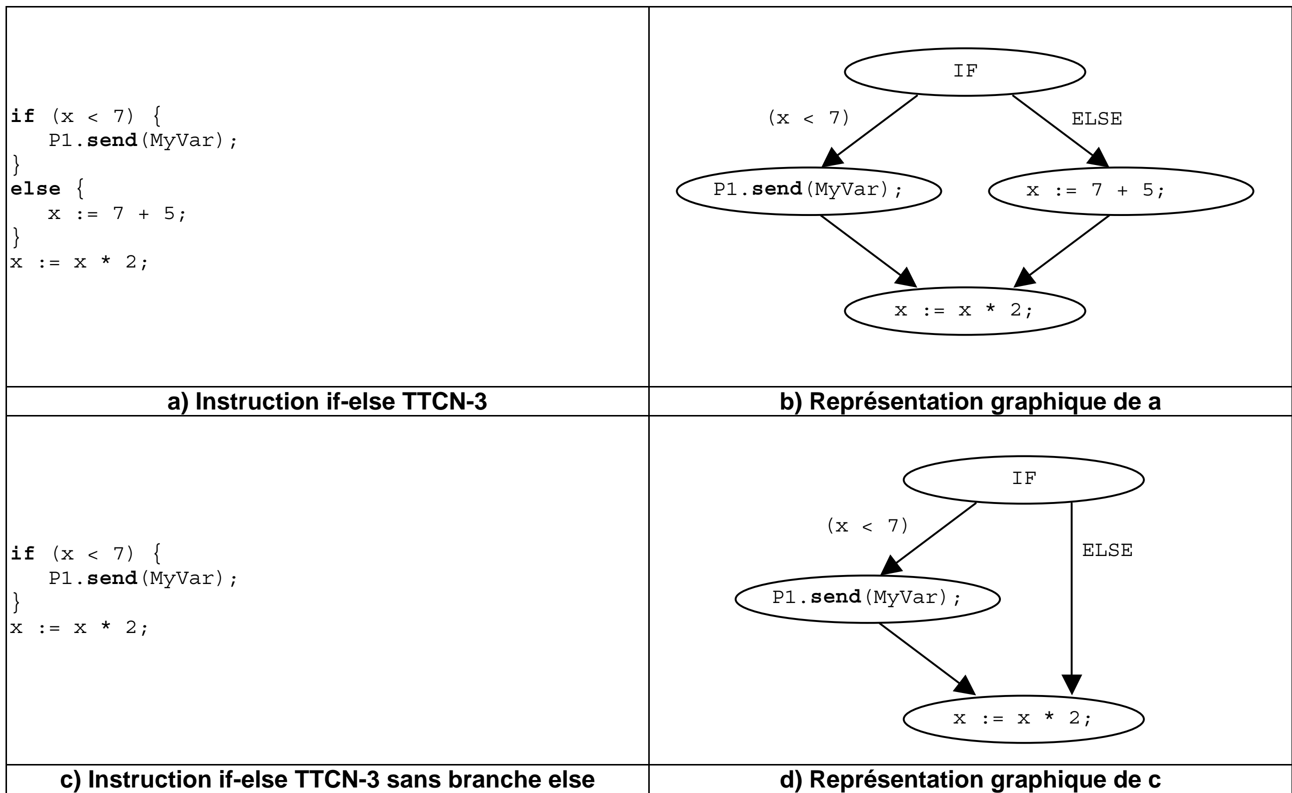
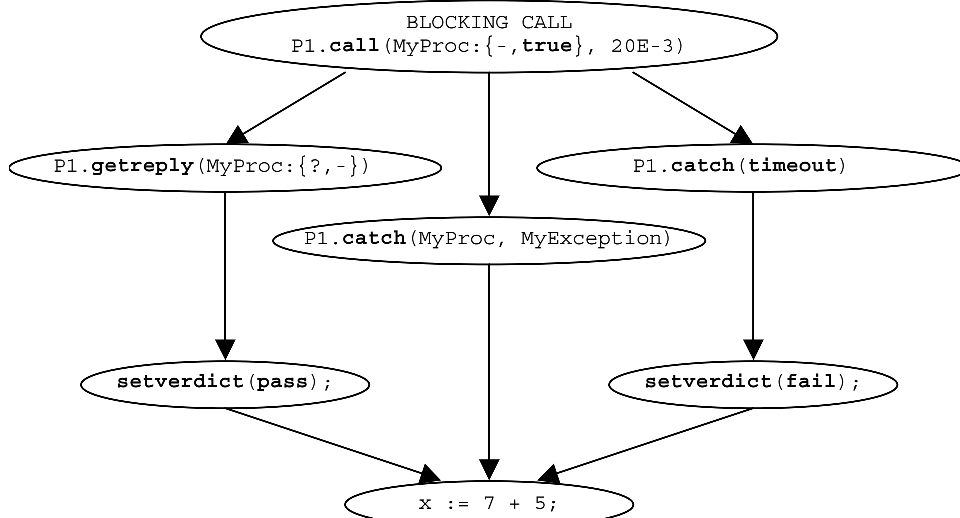


Figure 2/Z.143 – Représentation graphique d'instructions if-else TTCN-3

La représentation graphique d'une instruction **`call`** bloquante est illustrée sur la Figure 3. Une instruction **`call`** bloquante est représentée par un nœud BLOCKING-CALL avec des arcs allant vers les instructions **`getreply`** et **`catch`** des différentes alternatives.

```
P1.call (MyProc:{-, true}, 20E-3) {
   [] P1.getreply(MyProc:{?,-} {
        setverdict(pass);
      }
   [] P1.catch(MyProc, MyException) {}
   [] P1.catch(timeout) {
        setverdict(fail);
      }
}
x := 7 + 5;
```

a) Instruction call bloquante TTCN-3



b) Représentation graphique de a

Figure 3/Z.143 – Représentation graphique d'une instruction call bloquante TTCN-3

La représentation graphique d'une instruction **`alt`** est illustrée sur la Figure 4. Une instruction **`alt`** est représentée par un nœud *alt* avec plusieurs arcs allant vers les différentes alternatives.

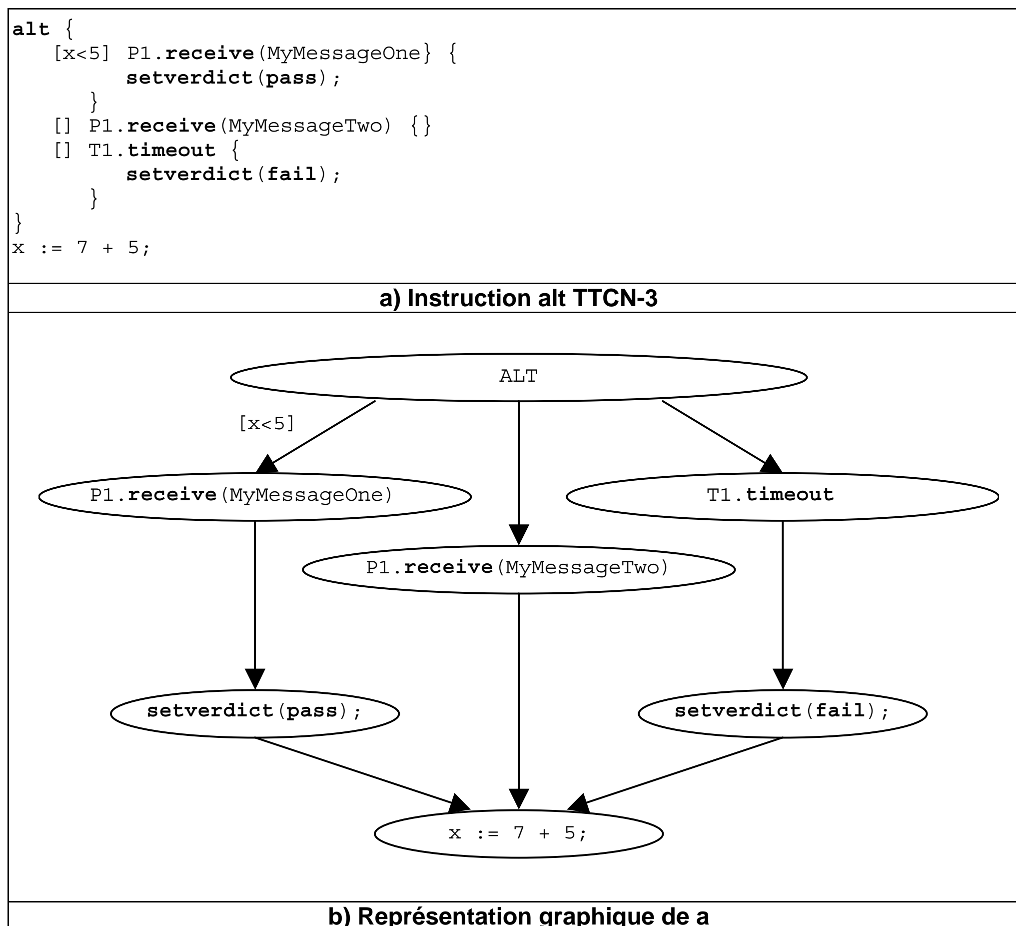
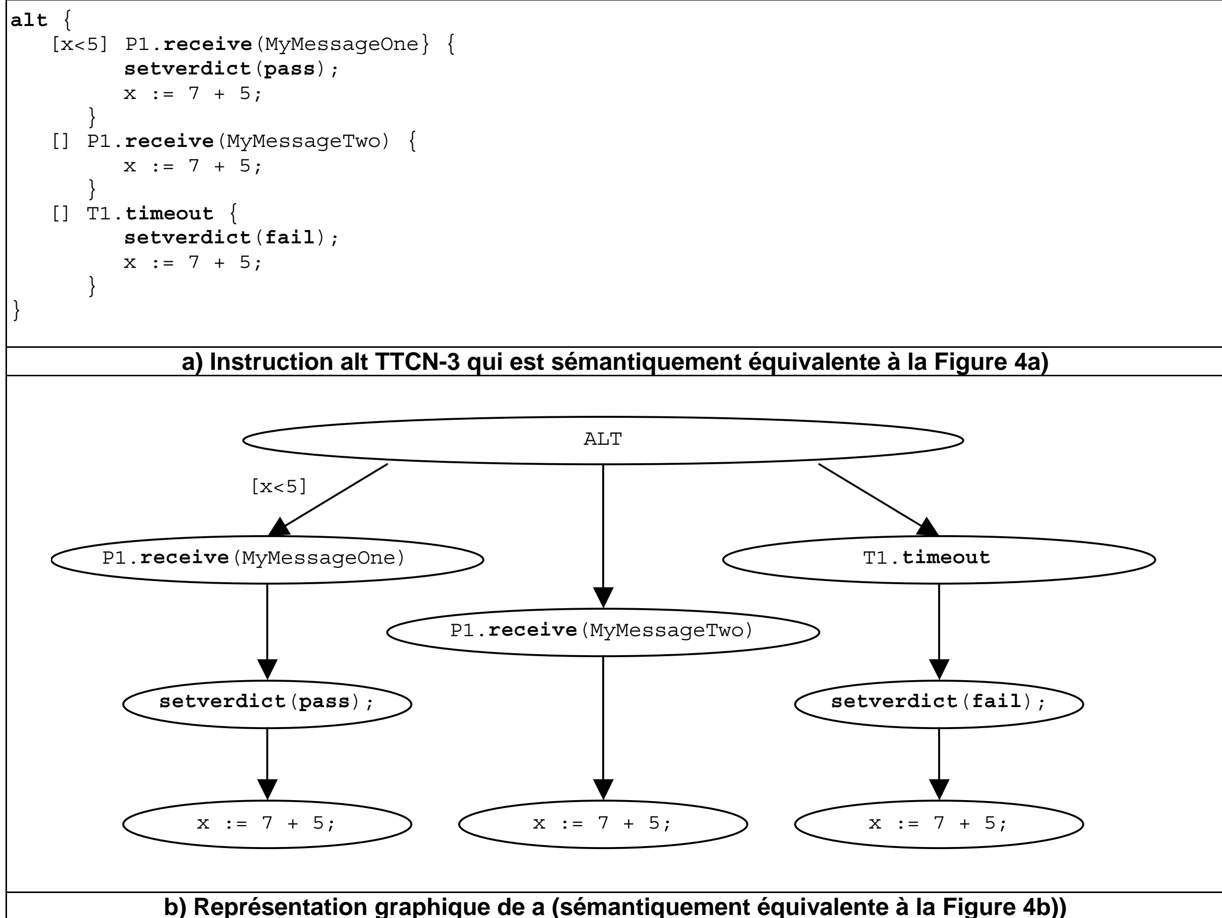


Figure 4/Z.143 – Représentation graphique d'une instruction alt TTCN-3

D'une manière générale, les représentations graphiques des instructions **`if-else`**, **`call`** *bloquantes* et **`alt`** correspondent à des graphes orientés sans boucle dans lesquels les arcs des différentes alternatives se rejoignent au moment de quitter l'instruction. La duplication permet de transformer ces graphes orientés en représentations arborescentes sémantiquement équivalentes. Cela est illustré sur la Figure 5 pour l'instruction **`alt`** représentée sur la Figure 4. L'algorithme décrit ci-dessous permet de construire ces représentations arborescentes.



a) Instruction alt TTCN-3 qui est sémantiquement équivalente à la Figure 4a)

b) Représentation graphique de a (sémantiquement équivalente à la Figure 4b))

Figure 5/Z.143 – Représentation graphique d'une instruction alt TTCN-3

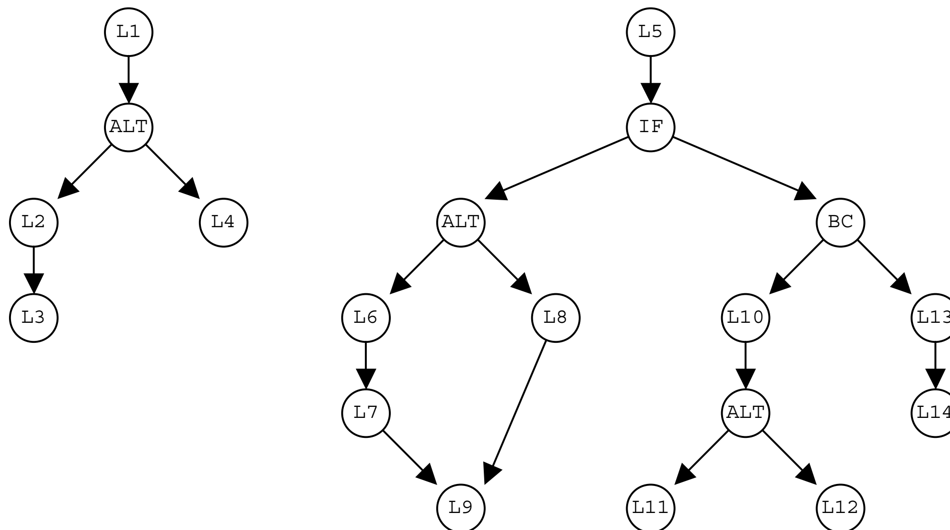
Une instruction **`interleave`** peut être décrite par un graphe constitué d'un ensemble de sous-graphes orientés, chacun étant construit à partir d'instructions autonomes et des instructions composites **`if-else`**, **`call`** *bloquantes* et **`alt`**. Les sous-graphes orientés décrivent les flux de commande entrelacés. Un exemple est illustré sur la Figure 6. Les inscriptions sur les nœuds de la Figure 6-b correspondent aux étiquettes des instructions TTCN-3 de la Figure 6-a.

```
interleave {
   [] P1.receive(M1} {                                  // L1
         alt {                                             // ALT
            [] P1.receive(M3) {                         // L2
                  setverdict(pass);                     // L3
               }
            [] T1.timeout { }                              // L4
         }


      }
   [] P2.receive(M2) {                                  // L5
         if (x < 5 ) {                                     // IF
            alt {                                          // ALT
               [] P2.receive(M4) {                      // L6
                     setverdict(pass);                  // L7
                  }
               [] Comp1.done { }                        // L8
            }
            x := 7 + 5;                                    // L9
         }
         else {
            P3.call(MyProcTempl, 20E-3) {    // BC (= BLOCKING CALL)

               [] P3.getreply(ReplyTempl) {     // L10
                     alt {                                 // ALT
                        [] P2.receive(M5) { }   // L11
                        [] P2.receive(M6) { }   // L12
                     }
                  }
               [] P3.catch(timeout) {           // L13
                     setverdict(fail);          // L14
                  }
            }
         }
      }
}
```

a) Instruction interleave TTCN-3



b) Représentation graphique de a

Figure 6/Z.143 – Représentation graphique d'une instruction interleave TTCN-3

Sur le plan formel, une instruction **`interleave`** peut être décrite par un graphe ***GI*** = (***St***, ***F***), où:

St est l'ensemble des instructions TTCN-3 autorisées;

$\boldsymbol{F} \subseteq (\boldsymbol{St} \times \boldsymbol{St})$ décrit la relation de flux.

Le terme *instructions TTCN-3 autorisées* renvoie aux restrictions statiques 1 à 5 ci-dessus.

Pour l'algorithme de construction, il faut définir les fonctions suivantes:

- La fonction <u>*REACHABLE*</u> retourne toutes les instructions qui peuvent être atteintes à partir d'une instruction $\boldsymbol{s}$ dans un graphe $\boldsymbol{GI} = (\boldsymbol{St}, \boldsymbol{F})$:

  <u>*REACHABLE*</u> $(\boldsymbol{s}, \boldsymbol{GI}) = \{ \boldsymbol{s} \} \cup$
  $\{ \boldsymbol{stmt} \mid \boldsymbol{stmt} \in \boldsymbol{St} \wedge \exists(\boldsymbol{s} = \boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n} = \boldsymbol{stmt})$ où $\boldsymbol{x_i} \in \boldsymbol{St}$,
  $i \in \{1 \ldots n\} \wedge (\boldsymbol{x_i}, \boldsymbol{x_{i+1}}) \in \boldsymbol{F}\}$

- La fonction <u>*SUCCESSORS*</u> retourne tous les successeurs d'une instruction $\boldsymbol{s}$ dans un graphe $\boldsymbol{GI} = (\boldsymbol{St}, \boldsymbol{F})$:

  <u>*SUCCESSORS*</u> $(\boldsymbol{s}, \boldsymbol{GI}) = \{ \boldsymbol{stmt} \mid \boldsymbol{stmt} \in \boldsymbol{St} \wedge (\boldsymbol{s}, \boldsymbol{stmt}) \in \boldsymbol{F}\}$

- La fonction <u>*ENABLED*</u> retourne toutes les instructions d'un graphe $\boldsymbol{GI} = (\boldsymbol{St}, \boldsymbol{F})$ qui n'ont pas de prédécesseur:

  <u>*ENABLED*</u> $(\boldsymbol{GI}) = \{ \boldsymbol{stmt} \mid \boldsymbol{stmt} \in \boldsymbol{St} \wedge (\boldsymbol{F} \cap (\boldsymbol{S} \times \{\boldsymbol{s}\}) = \emptyset)\}$

- La fonction <u>*KIND*</u> retourne la sorte ou le type d'une instruction TTCN-3 dans un graphe représentant une instruction **`interleave`**.

- La fonction <u>*DISCARD*</u> supprime une instruction $\boldsymbol{s}$ ou un ensemble d'instructions $\boldsymbol{S}$ d'un graphe $\boldsymbol{GI} = (\boldsymbol{St}, \boldsymbol{F})$ et retourne le graphe résultant $\boldsymbol{GI'} = (\boldsymbol{St'}, \boldsymbol{F'})$:

  Pour des nœuds uniques:

  <u>*DISCARD*</u> $(\boldsymbol{s}, \boldsymbol{GI}) = \boldsymbol{GI'}$ where: $\boldsymbol{GI'} = (\boldsymbol{St'}, \boldsymbol{F'})$, with $\boldsymbol{St'} = \boldsymbol{St} \backslash \{\boldsymbol{s}\}$ and
  $\boldsymbol{F'} = \boldsymbol{F} \cap (\boldsymbol{St} \backslash \{\boldsymbol{s}\} \times \boldsymbol{St} \backslash \{\boldsymbol{s}\})$.

  Pour des ensembles de nœuds:

  <u>*DISCARD*</u> $(\boldsymbol{S}, \boldsymbol{GI}) = \boldsymbol{GI'}$ où: $\boldsymbol{GI'} = (\boldsymbol{St'}, \boldsymbol{F'})$, avec $\boldsymbol{St'} = \boldsymbol{St} \backslash \boldsymbol{S}$ et $\boldsymbol{F'} = \boldsymbol{F} \cap (\boldsymbol{St} \backslash \boldsymbol{S} \times \boldsymbol{St} \backslash \boldsymbol{S})$.

- La fonction <u>*RECEIVING*</u> prend un ensemble d'instructions d'un graphe $\boldsymbol{GI}$ et retourne toutes les instructions de réception:

  <u>*RECEIVING*</u> $(S) = \{ stmt \mid stmt \in St \wedge$ <u>*KIND*</u>$(stmt) \in$ {receive, trigger, getcall, getreply, catch, check, done, timeout}}

- La fonction <u>*RANDOM*</u> choisit un élément $\boldsymbol{s}$ au hasard dans un ensemble donné $\boldsymbol{S}$ et retourne $\boldsymbol{s}$.

  <u>*RANDOM*</u> $(\boldsymbol{S}) = \boldsymbol{s}$ où $\boldsymbol{s} \in \boldsymbol{S}$

L'algorithme de construction (voir la Figure 7) de l'arbre est une procédure récursive pour laquelle, dans chaque appel récursif, les nœuds successeurs d'un nœud donné sont construits. La procédure est fournie dans un pseudo-code semblable au langage C qui utilise les fonctions définies ci-dessus et certaines notations mathématiques additionnelles.

```
CONSTRUCT-SUCCESSORS (statementType *predecessor, graphType GI) {
    // - statementType désigne le type d'un nœud de l'arbre qui est construit
    // - *predecessor désigne le dernier nœud qui a été créé
    // - graphType désigne le type du graphe d'instructions TTCN-3
    // - GI est appelé par valeur et désigne le sous-graphe constitué de toutes les instructions
    // TTCN-3 restantes qui doivent être prises en considération

    var graphType myGraph;
    var statementType i, myStmt;
    var statementType *newStmt, *firstInBranch; // pointeurs pour les nouveaux nœuds d'instruction
                                    // dans l'arbre qui est construit de façon récursive

    // Extraction des ensembles d'instructions TTCN-3 qui n'ont pas de prédécesseur dans 'GI'
    var statementSet enabStmts := ENABLED(GI);      // toutes les instructions sans prédécesseur
    var statementSet enabRecStmts := RECEIVING(enabStmts); // instructions de réception de
                                            // 'enabStmts'
    var statementSet enabNonRecStmts := enabStmts\enabRecStmts;
                                        // instructions autres que de réception de 'enabStmts'

    if (GI.St == ∅) {   // On suppose que GI.St désigne l'ensemble des instructions de GI
        return; // Il ne reste plus d'instruction, critère de terminaison de la procédure récursive
    }
    elseif (enabNonRecStmts != ∅) { // Traitement des instructions autres que de réception de
                                // 'enabStmts'

        myStmt := RANDOM(enabNonRecStmts);
                // Il ne peut y avoir qu'une seule instruction dans 'enabNonRec', car l'algorithme
                // continue la construction tant qu'il existe une branche qui contribue à
                // l'instruction interlave.
        newStmt := create(myStmt, predecessor);
                // Création d'un nouveau nœud représentant 'myStmt' dans l'arbre
                // et mise à jour des pointeurs dans 'newStmt' et 'predecessor'.

        if (KIND(myStmt) == IF || KIND(myStmt) == BLOCKING_CALL) {
            for each i in SUCCESSORS(myStmt, GI) {

                firstInBranch := create(i, newStmt);
                        // Création d'un deuxième nœud pour la première instruction d'une branche
                        // résultant d'une instruction if-else.
                        // Il est à noter que cette instruction create sera utilisée pour créer
                        // trois nœuds représentant les instructions de réception dans les
                        // opérations call bloquantes.

                myGraph := DISCARD({i, myStmt} ∪ REACHABLE(myStmt, GI)\REACHABLE(i, GI))
                        // Suppression de i, myStmt et toutes les instructions atteignables
                        // depuis myStmt mais pas celles qui sont atteignables depuis i. Dans
                        // le dernier cas, on considère la succession des branches d'un flux
                        // de commande dans un sous-graphe de GI.

                CONSTRUCT-SUCCESSORS(firstInBranch, myGraph);   // ETAPE SUIVANTE DE LA
                                                                // PROCEDURE RECURSIVE
            }
        }
        elseif (KIND(myStmt) == ALT) {
            for each (i in SUCCESSORS(myStmt, GI) {

            CONSTRUCT-SUCCESSORS(mystmt, DISCARD(REACHABLE(myStmt, GI)\REACHABLE(i, GI)));
                    // ETAPE SUIVANTE DE LA PROCEDURE RECURSIVE, dans l'argument
                        // DISCARD(REACHABLE(myStmt, GI)\REACHABLE(i, GI)), on considère
                        // la succession des branches dans un flux de commande résultant de
                        // différents événements de réception.
            }
        }
        else {      // mystmt est une instruction autonome
            CONSTRUCT-SUCCESSORS(newSonNode, DISCARD(myStmt, GI));
                    // ETAPE SUIVANTE DE LA PROCEDURE RECURSIVE
        }
    }
    else {  // Traitement des événements de réception qui s'entrelacent
        if (KIND(predecessor) != ALT) { // un nœud alt est manquant et doit être créé, si
                        // l'entrelacement n'est pas influencé par une instruction alt imbriquée
            predecessor := create(ALT, predecessor);
        }

        for each i in enabRecStmts) {
            newStmt := create(i, predecessor);              // Nouveau nœud dans l'arbre
            CONSTRUCT-SUCCESSORS(newStmt, DISCARD(i, GI));  // ETAPE(S) SUIVANTE(S) DE LA
                                                            // PROCEDURE RECURSIVE
        }
    }
}
```

Figure 7/Z.143 – Algorithme de remplacement des instructions interleave TTCN-3

Au départ, la fonction CONSTRUCT-SUCCESSORS (voir la Figure 7) est appelée avec un *nœud racine* d'un arbre vide et le graphe d'instructions TTCN-3 décrivant l'instruction **`interleave`** qui doit être remplacée. Une fois cette fonction terminée, le *nœud racine* peut être utilisé pour accéder à l'arbre construit.

L'application de la fonction CONSTRUCT-SUCCESSORS à l'instruction **`interleave`** illustrée sur la Figure 6 conduit à l'arbre illustré sur la Figure 8. Les étiquettes correspondent aux instructions de la Figure 6-a. Les étiquettes multiples sont le résultat de la duplication de code. Le code TTCN-3 qui correspond à l'arbre illustré sur la Figure 8 est présenté sur la Figure 9.

NOTE 5 – L'exemple d'application de l'algorithme de la Figure 7 (voir les Figures 6, 8 et 9) est très complet. Il permet d'illustrer la plupart des situations particulières, à savoir la succession de branches, des arcs qui se rejoignent, une instruction **`alt`** imbriquée, une instruction **`call`** *bloquante* et une instruction **`if-else`**.

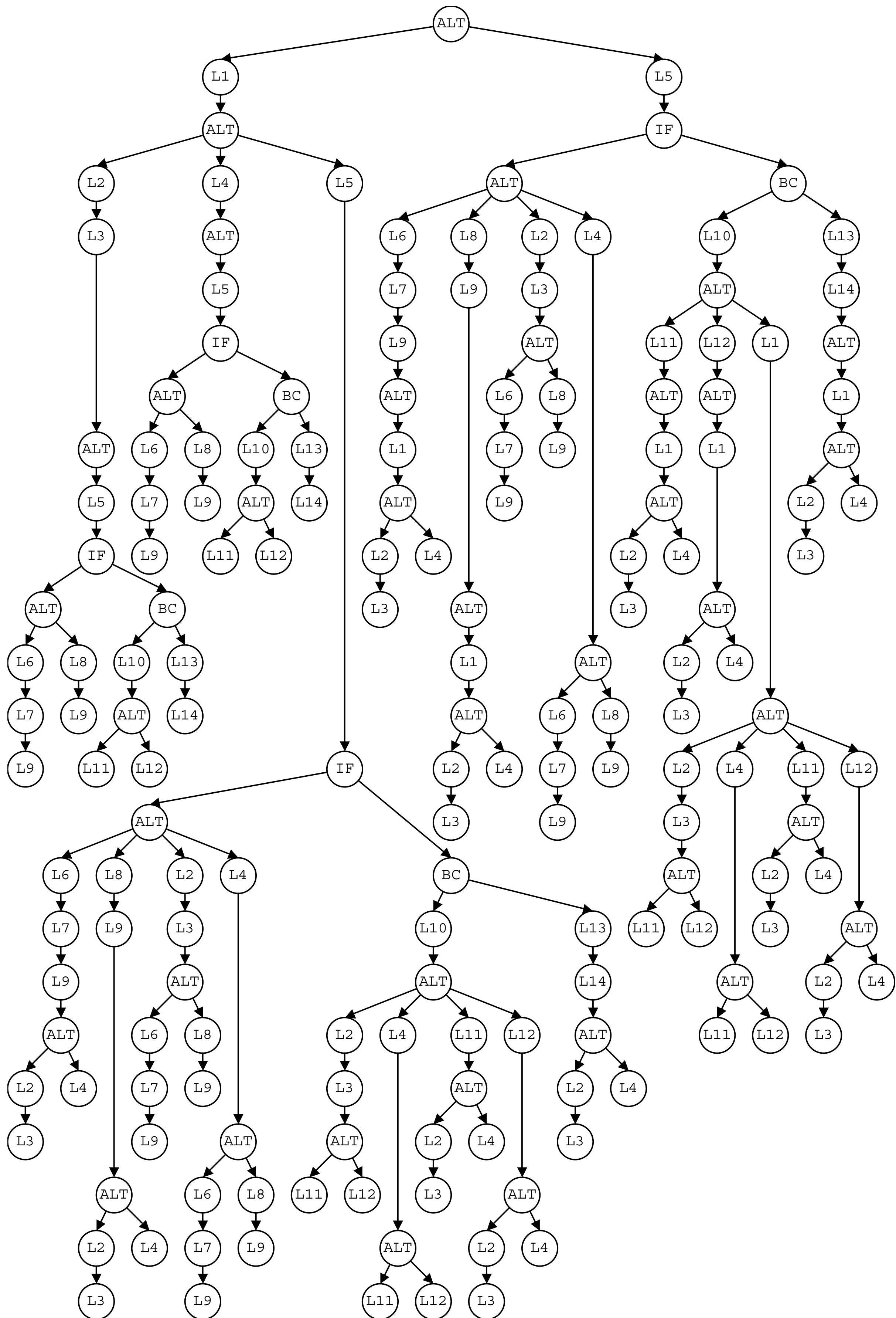


Figure 8/Z.143 – Résultat de l'application de l'algorithme de la Figure 7 à l'instruction interleave de la Figure 6

```
alt {                                                                                   // ALT
    []  P1.receive(M1} {                                                                // L1
            alt {                                                                       // ALT
                []  P1.receive(M3) {                                                    // L2
                        setverdict(pass);                                               // L3
                        alt {                                                           // ALT
                            []  P2.receive(M2) {                                        // L5
                                    if (x < 5 ) {                                       // IF
                                        alt {                                           // ALT
                                                []  P2.receive(M4) {                    // L6
                                                        setverdict(pass);               // L7
                                                        x := 7 + 5;                     // L9
                                                    }
                                                []  Comp1.done {                        // L8
                                                        x := 7 + 5;                     // L9
                                    }   }                   }
                                    else {
                                        P3.call(MyProcTempl, 20E-3) { // BC (= BLOCKING CALL)
                                            []  P3.getreply(ReplyTempl) {               // L10
                                                    alt {                               // ALT
                                                        []  P2.receive(M5) { }          // L11
                                                        []  P2.receive(M6) { }          // L12
                                                }   }
                                            []  P3.catch(timeout) {                     // L13
                                                    setverdict(fail);                   // L14
                    }  }        }   }   }       }
                []  T1.timeout {                                                        // L4
                        alt {                                                           // ALT
                            []  P2.receive(M2) {                                        // L5
                                    if (x < 5 ) {                                       // IF
                                        alt {                                           // ALT
                                                []  P2.receive(M4) {                    // L6
                                                        setverdict(pass);               // L7
                                                        x := 7 + 5;                     // L9
                                                    }
                                                []  Comp1.done {                        // L8
                                                        x := 7 + 5;                     // L9
                                    }   }                   }
                                    else {
                                        P3.call(MyProcTempl, 20E-3) { // BC (= BLOCKING CALL)
                                            []  P3.getreply(ReplyTempl) {               // L10
                                                    alt {                               // ALT
                                                        []  P2.receive(M5) { }          // L11
                                                        []  P2.receive(M6) { }          // L12
                                                }   }
                                            []  P3.catch(timeout) {                     // L13
                                                    setverdict(fail);                   // L14
                    }   }       }   }   }       }
                []  P2.receive(M2) {                                                    // L5
                        if (x < 5 ) {                                                   // IF
                            alt {                                                       // ALT
                                []  P2.receive(M4) {                                    // L6
                                        setverdict(pass);                               // L7
                                        x := 7 + 5;                                     // L9
                                        alt {                                           // ALT
                                            []  P1.receive(M3) {                        // L2
                                                    setverdict(pass);                   // L3
                                                }
                                            []  T1.timeout { }                          // L4
                                    }   }
                                []  Comp1.done {                                        // L8
                                        x := 7 + 5;                                     // L9
                                        alt {                                           // ALT
                                            []  P1.receive(M3) {                        // L2
                                                    setverdict(pass);                   // L3
                                                }
                                            []  T1.timeout { }                          // L4
                                    }   }
                                []  P1.receive(M3) {                                    // L2
                                        setverdict(pass);                               // L3
                                        alt {                                           // ALT
                                            []  P2.receive(M4) {                        // L6
                                                    setverdict(pass);                   // L7
                                                    x := 7 + 5;                         // L9
                                                }
                                            []  Comp1.done {                            // L8
                                                    x := 7 + 5;                         // L9
                                    }   }       }
                                []  T1.timeout {                                        // L4
                                        alt {                                           // ALT
```

```
                                            []  P2.receive(M4) {                        // L6
                                                    setverdict(pass);                   // L7
                                                    x := 7 + 5;                         // L9
                                                }
                                            []  Comp1.done {                            // L8
                                                    x := 7 + 5;                         // L9
                        }   }       }   }       }
                        else {
                            P3.call(MyProcTempl, 20E-3) {           // BC (= BLOCKING CALL)
                                []  P3.getreply(ReplyTempl) {                           // L10
                                        alt {                                           // ALT
                                            []  P1.receive(M3) {                        // L2
                                                    setverdict(pass);                   // L3
                                                    alt {                               // ALT
                                                        []  P2.receive(M5) { }          // L11
                                                        []  P2.receive(M6) { }          // L12
                                                }   }
                                            []  T1.timeout {                            // L4
                                                    alt {                               // ALT
                                                        []  P2.receive(M5) { }          // L11
                                                        []  P2.receive(M6) { }          // L12
                                                }   }
                                            []  P2.receive(M5) {                        // L11
                                                    alt {                               // ALT
                                                        []  P1.receive(M3) {            // L2
                                                                setverdict(pass);       // L3
                                                            }
                                                        []  T1.timeout { }              // L4
                                                }   }
                                            []  P2.receive(M6) {                        // L12
                                                    alt {                               // ALT
                                                        []  P1.receive(M3) {            // L2
                                                                setverdict(pass);       // L3
                                                            }
                                                        []  T1.timeout { }              // L4
                                    }   }       }   }
                                []  P3.catch(timeout) {                                 // L13
                                        setverdict(fail);                               // L14
                                        alt {                                           // ALT
                                            []  P1.receive(M3) {                        // L2
                                                    setverdict(pass);                   // L3
                                                }
                                            []  T1.timeout { }                          // L4
        }   }       }   }   }       }   }
    []  P2.receive(M2) {                                                                // L5
            if (x < 5 ) {                                                               // IF
                alt {                                                                   // ALT
                    []  P2.receive(M4) {                                                // L6
                            setverdict(pass);                                           // L7
                            x := 7 + 5;                                                 // L9
                            alt {                                                       // ALT
                                []  P1.receive(M1} {                                    // L1
                                        alt {                                           // ALT
                                            []  P1.receive(M3) {                        // L2
                                                    setverdict(pass);                   // L3
                                                }
                                            []  T1.timeout { }                          // L4
                        }   }       }   }
                    []  Comp1.done {                                                    // L8
                            x := 7 + 5;                                                 // L9
                            alt {                                                       // ALT
                                []  P1.receive(M1} {                                    // L1
                                        alt {                                           // ALT
                                            []  P1.receive(M3) {                        // L2
                                                    setverdict(pass);                   // L3
                                                }
                                            []  T1.timeout { }                          // L4
                        }   }       }   }
                    []  P1.receive(M3) {                                                // L2
                            setverdict(pass);                                           // L3
                            alt {                                                       // ALT
                                []  P2.receive(M4) {                                    // L6
                                        setverdict(pass);                               // L7
                                        x := 7 + 5;                                     // L9
                                    }
                                []  Comp1.done {                                        // L8
                                        x := 7 + 5;                                     // L9
                        }   }       }
                    []  T1.timeout {                                                    // L4
                            alt {                                                       // ALT
```

```
                        []  P2.receive(M4) {                                        // L6
                                setverdict(pass);                                   // L7
                                x := 7 + 5;                                         // L9
                            }
                        []  Comp1.done {                                            // L8
                                x := 7 + 5;                                         // L9
            }   }       }   }       }
            else {
                P3.call(MyProcTempl, 20E-3) {                   // BC (= BLOCKING CALL)
                    []  P3.getreply(ReplyTempl) {                                   // L10
                            alt {                                                   // ALT
                                []  P2.receive(M5) {                                // L11
                                        alt {                                       // ALT
                                            []  P1.receive(M1} {                    // L1
                                                    alt {                           // ALT
                                                        []  P1.receive(M3) {        // L2
                                                                setverdict(pass);   // L3
                                                            }
                                                        []  T1.timeout { }          // L4
                                    }   }       }   }
                                []  P2.receive(M6) {                                // L12
                                        alt {                                       // ALT
                                            []  P1.receive(M1} {                    // L1
                                                    alt {                           // ALT
                                                        []  P1.receive(M3) {        // L2
                                                                setverdict(pass);   // L3
                                                            }
                                                        []  T1.timeout { }          // L4
                                    }   }       }   }
                                []  P1.receive(M1} {                                // L1
                                        alt {                                       // ALT
                                            []  P1.receive(M3) {                    // L2
                                                    setverdict(pass);               // L3
                                                    alt {                           // ALT
                                                        []  P2.receive(M5) { }      // L11
                                                        []  P2.receive(M6) { }      // L12
                                                }   }
                                            []  T1.timeout {                        // L4
                                                    alt {                           // ALT
                                                        []  P2.receive(M5) { }      // L11
                                                        []  P2.receive(M6) { }      // L12
                                                }   }
                                            []  P2.receive(M5) {                    // L11
                                                    alt {                           // ALT
                                                        []  P1.receive(M3) {        // L2
                                                                setverdict(pass);   // L3
                                                            }
                                                        []  T1.timeout { }          // L4
                                                }   }
                                            []  P2.receive(M6) {                    // L12
                                                    alt {                           // ALT
                                                        []  P1.receive(M3) {        // L2
                                                                setverdict(pass);   // L3
                                                            }
                                                        []  T1.timeout { }          // L4
                        }   }       }   }       }   }
                    []  P3.catch(timeout) {                                         // L13
                            setverdict(fail);                                       // L14
                            alt {                                                   // ALT
                                []  P1.receive(M1} {                                // L1
                                        alt {                                       // ALT
                                            []  P1.receive(M3) {                    // L2
                                                    setverdict(pass);               // L3
                                                }
                                            []  T1.timeout { }                      // L4
}       }   }   }       }   }       }   }
```

Figure 9/Z.143 – Code TTCN-3 sémantiquement équivalent à l'instruction interleave de la Figure 6

7.6 Remplacement des opérations trigger

L'opération **trigger** filtre les messages avec un certain critère d'appariement à partir d'un flux de messages à un port donné. La sémantique de l'opération **trigger** peut être décrite par son remplacement par deux opérations **receive** et une instruction **goto**. Dans la sémantique opérationnelle, on suppose que ce remplacement est opéré au niveau syntaxique.

EXEMPLE 1:

```
// L'opération trigger suivante ...

        alt {
         []  MyCL.trigger (MyType:?) { }
        }

// doit être remplacée par ...

    alt {
        []   MyCL.receive (MyType:?) { }
        []   MyCL.receive {
             repeat
         }
    }
```

Si l'instruction **trigger** est utilisée dans une instruction **alt** plus complexe, le remplacement est opéré de la même manière.

EXEMPLE 2:

```
// L'instruction alt suivante inclut une instruction trigger ...

    alt {
        []   PCO2.receive {
             stop;
         }
        []   MyCL.trigger (MyType:?) { }
        []   PCO3.catch {
             setverdict(fail);
             stop;
         }
    }

// qui sera remplacée par

    alt {
        []   PCO2.receive {
             stop;
         }
        []   MyCL.receive (MyType:?) { }
        []   MyCL.receive {
             repeat;
         }
        []   PCO3.catch {
             setverdict(fail);
             stop;
         }
    }
```

8 Sémantique de la notation TTCN-3 sur la base de graphes de flux

La sémantique opérationnelle de la notation TTCN-3 est fondée sur l'interprétation de graphes de flux. Dans le présent paragraphe, on expose les graphes de flux (voir § 8.1), on explique la construction des graphes de flux représentant la commande de module, les tests élémentaires, les variantes, les fonctions et les définitions de type de composant TTCN-3 (voir § 8.2), on définit les états de module et de composant pour la description des états d'exécution d'un module TTCN-3 (voir § 8.3), on décrit le traitement des messages, des appels de procédure distante, des réponses aux appels de procédure distante et des exceptions (voir § 8.4) et on explique la procédure d'évaluation de la commande de module et des tests élémentaires (voir § 8.6).

8.1 Graphes de flux

Un graphe de flux est un graphe orienté constitué de nœuds étiquetés et d'arcs étiquetés. La traversée d'un graphe de flux décrit le flux de commande possible pendant l'exécution d'une description de comportement représentée.

8.1.1 Cadre d'un graphe de flux

Un graphe de flux doit être placé dans un cadre définissant la limite du graphe de flux. Le nom du graphe de flux suit les mots clés **flow graph** (il ne s'agit pas de mots clés du langage noyau TTCN-3) et doit être placé dans le coin en haut à gauche du graphe de flux. Par convention, on suppose que le nom du graphe de flux renvoie à la description de comportement TTCN représentée par le graphe de flux. Un graphe de flux simple est illustré sur la Figure 10.

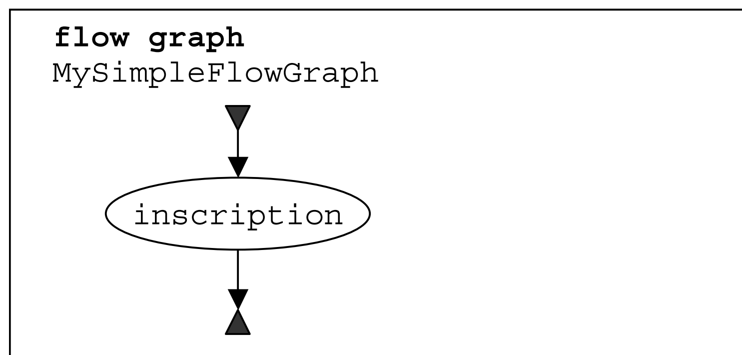


Figure 10/Z.143 – Graphe de flux simple

8.1.2 Nœuds d'un graphe de flux

Un graphe de flux est constitué d'un *nœud de début*, de *nœuds de fin*, de *nœuds de base* et de *nœuds de référence*.

8.1.2.1 Nœud de début

Le nœud de début correspond au point de départ d'un graphe de flux. Un graphe de flux possède un seul nœud de début. Un nœud de début est illustré sur la Figure 11-a.

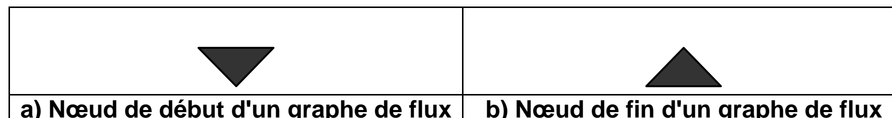


Figure 11/Z.143 – Nœuds de début et de fin

8.1.2.2 Nœuds de fin

Les nœuds de fin correspondent aux points de terminaison d'un graphe de flux. Un graphe de flux peut avoir plusieurs nœuds de fin ou, dans le cas de boucles, aucun nœud de fin. Les nœuds de base (voir § 8.1.2.3) et les nœuds de référence (voir § 8.1.2.4) qui n'ont pas de nœud successeur doivent être raccordés à un nœud de fin pour indiquer qu'ils décrivent la dernière action d'un chemin à travers un graphe de flux. Un nœud de fin est illustré sur la Figure 11-b.

8.1.2.3 Nœuds de base

Un nœud de base correspond à une unité d'exécution, autrement dit il est exécuté en une seule étape. Un nœud de base a un type et, suivant le type, il peut avoir une liste d'attributs associés. Deux nœuds de base sont illustrés sur la Figure 12.

Dans l'inscription d'un nœud de base, les attributs du nœud suivent le type de nœud et sont placés entre parenthèses. Le type et les attributs servent à déterminer l'action à réaliser pendant l'exécution de la construction de langage représentée. Les attributs décrivent les informations à extraire de la construction TTCN-3 correspondante.

Les attributs ont des valeurs, que la sémantique opérationnelle extraira en faisant référence au nom de l'attribut. Si nécessaire, il est autorisé d'affecter des valeurs explicites dans les nœuds de base au moyen de l'affectation ':='. Un exemple est illustré sur la Figure 12-b.

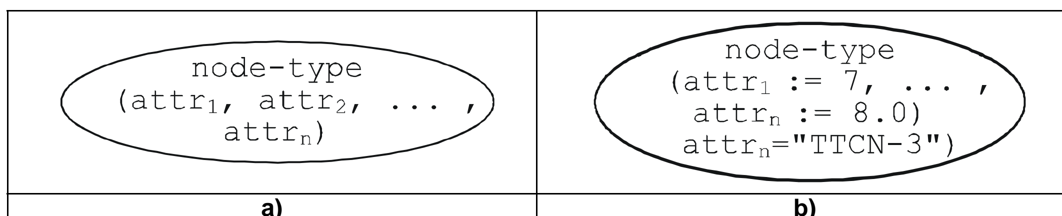


Figure 12/Z.143 – Nœuds de base avec attributs

8.1.2.4 Nœuds de référence

Les nœuds de référence font référence à des segments de graphe de flux (voir § 8.1.4) qui sont des sous-graphes de flux. La signification d'un nœud de référence est définie par son remplacement dans le graphe de flux par le segment de graphe de flux référencé. L'inscription du nœud de référence donne la référence à un segment de graphe de flux. Un nœud de référence est illustré sur la Figure 13-a.

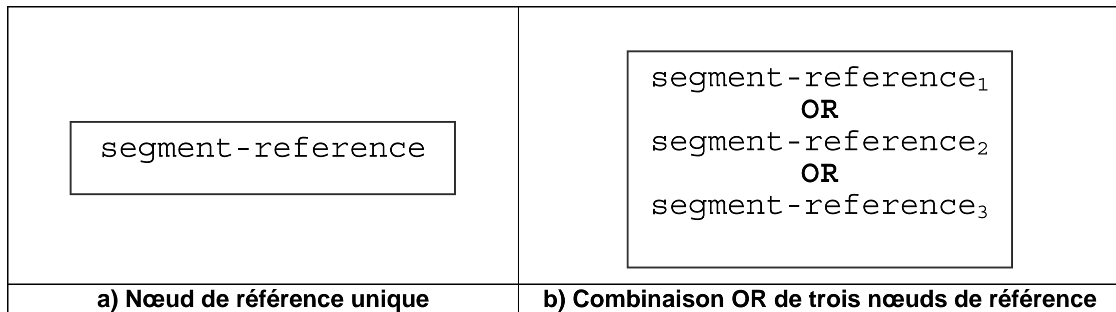


Figure 13/Z.143 – Nœuds de référence

8.1.2.4.1 Combinaison OR de nœuds de référence

Dans certains cas, plusieurs segments de graphe de flux peuvent remplacer un nœud de référence. Dans ces cas, on peut utiliser un opérateur **OR** pour faire référence à plusieurs segments de graphe de flux (voir Figure 13-b). Dans le graphe de flux effectif représentant la commande de module, un test élémentaire ou une fonction, une seule alternative est déterminée par la construction représentée.

8.1.2.4.2 Occurrences multiples de nœuds de référence

Dans certains cas, la même sorte de nœud de référence peut apparaître zéro, une ou plusieurs fois dans un graphe de flux. Dans les expressions régulières, la répétition possible de parties d'une expression régulière est décrite grâce à l'utilisation des symboles d'opérateur '+' (une ou plusieurs répétitions) et '*' (zéro, une ou plusieurs répétitions). Comme indiqué sur la Figure 14, ces opérateurs ont été adoptés dans les graphes de flux par l'introduction de nœuds de référence à double cadre avec un symbole d'opérateur associé. Un arc unique (voir § 8.1.3) doit remplacer un nœud de référence, lorsque le nombre d'occurrences est nul (nœud de référence à double cadre avec opérateur '*').

Figure 14/Z.143 – Répétition de nœuds de référence

Il est possible de donner 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 cadre. La référence de segment illustrée sur la Figure 15 peut se produire de zéro à 5 fois.

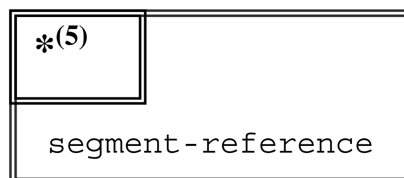
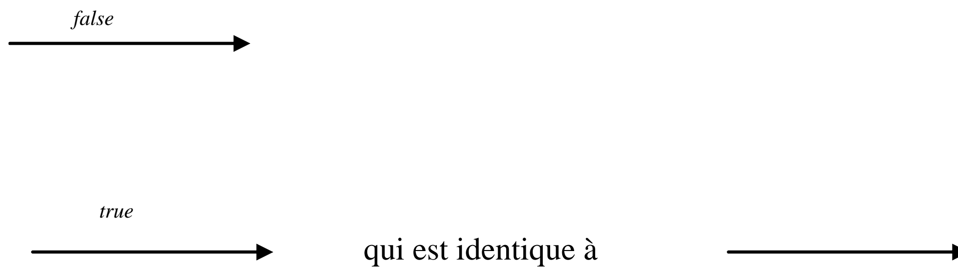


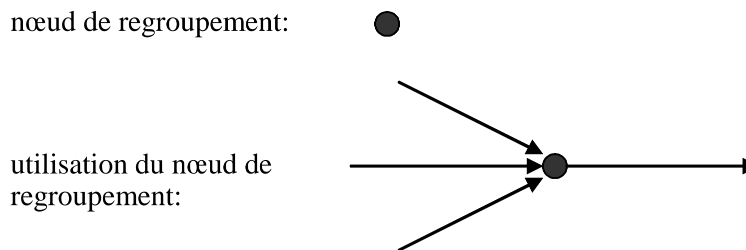
Figure 15/Z.143 – Répétition restreinte d'un nœud de référence

8.1.3 Arcs

Les arcs sont représentés par des flèches. Un arc a une inscription *true* ou *false* qui indique une condition dans laquelle l'arc est choisi pendant l'interprétation du graphe de flux. Dans une notation abrégée, il est autorisé d'omettre l'inscription *true*. Des exemples d'arcs sont donnés ci-dessous:



Pour pouvoir regrouper plusieurs arcs vers un seul arc au niveau graphique, un nœud spécial de regroupement est introduit. Ce nœud ainsi qu'un exemple d'utilisation sont illustrés ci-dessous:



Il est difficile de tracer de longs arcs dans les grands diagrammes comme c'est par exemple nécessaire pour modéliser les constructions TTCN-3 **`goto`** et **`label`**. Dans ce cas, on peut alors utiliser des étiquettes pour des arcs sortants et des arcs entrants. Des exemples sont donnés ci-dessous.

Un arc sortant avec une étiquette est raccordé à un arc entrant avec une étiquette, si les deux étiquettes sont identiques. Les étiquettes d'arc entrant doivent être uniques. Si plusieurs arcs sortants ont la même étiquette, on considère qu'il s'agit d'un regroupement d'arcs vers l'arc entrant ayant la même étiquette.

8.1.4 Segments de graphe de flux

Les segments de graphe de flux sont des sous-graphes de flux, auxquels il est fait référence dans des nœuds de référence et qui définissent la signification de ces nœuds de référence. Ils peuvent inclure d'autres nœuds de référence.

Comme indiqué sur la Figure 16, les segments de graphe de flux ont des interfaces précises constituées d'arcs entrants et d'arcs sortants. Un seul arc entrant et un ou zéro arc sortant ne sont pas étiquetés. En outre, il peut y avoir plusieurs arcs entrants et plusieurs arcs sortants étiquetés. Par exemple, des arcs entrants et sortants étiquetés sont nécessaires pour décrire la signification des instructions TTCN-3 **`goto`** et **`alt`**.

Un segment de graphe de flux est placé dans un cadre et le nom du segment doit suivre le mot clé **`segment`** dans le coin en haut à gauche du cadre. Les arcs décrivant l'interface du segment de graphe de flux doivent franchir le cadre du segment de graphe de flux.

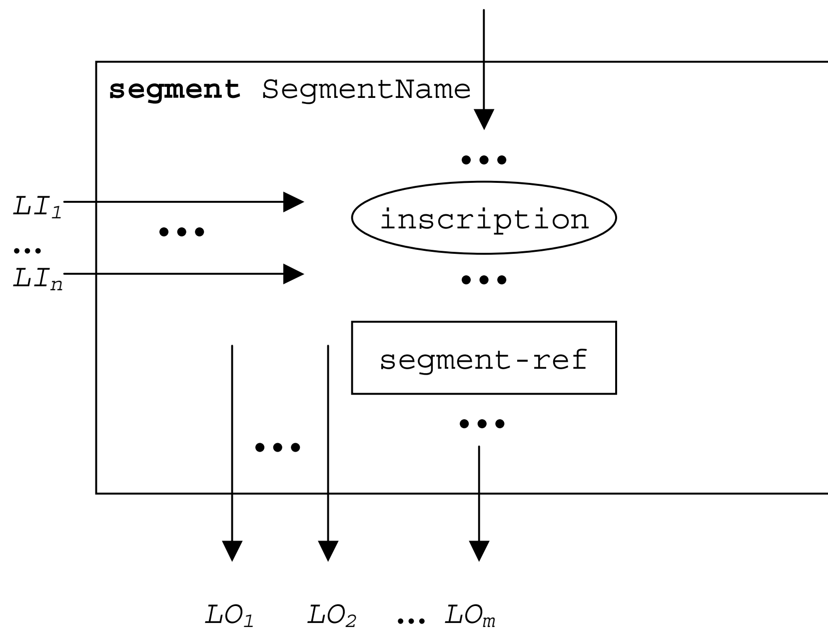


Figure 16/Z.143 – Structure d'une description de segment de graphe de flux

8.1.5 Commentaires

Pour améliorer la lisibilité et la cohérence, on peut utiliser un symbole spécial de commentaire pour associer des commentaires aux nœuds et aux arcs d'un graphe de flux. Le symbole de commentaire et son utilisation sont illustrés sur la Figure 17.

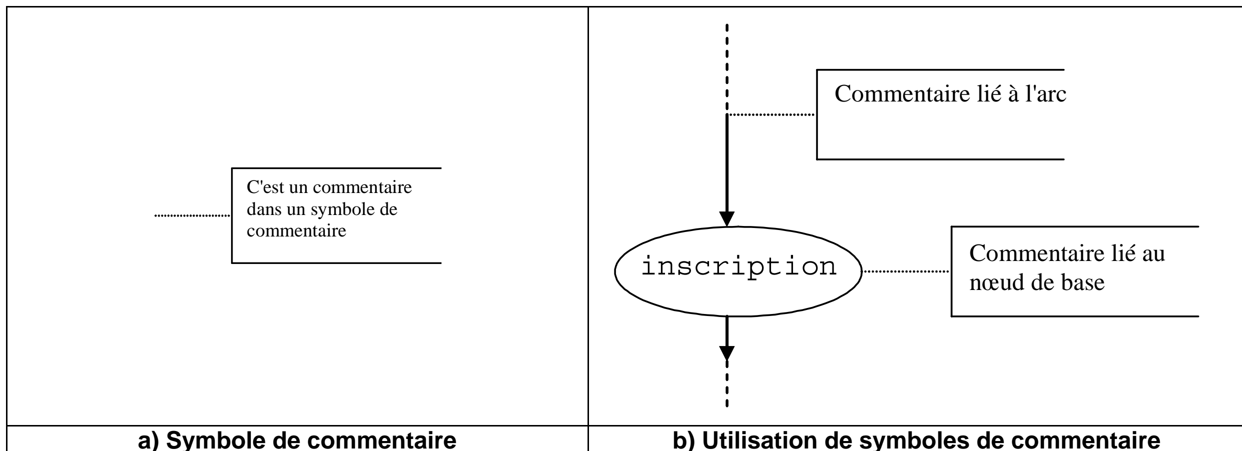


Figure 17/Z.143 – Représentation des commentaires dans les graphes de flux

8.1.6 Traitement des descriptions de graphe de flux

La procédure d'évaluation de la sémantique opérationnelle traverse des graphes de flux comportant uniquement des nœuds de base, autrement dit tous les nœuds de référence doivent être développés au moyen des définitions de segment de graphe de flux correspondantes. La fonction *NEXT* est nécessaire pour la prise en charge de cette traversée. Elle est définie comme suit:

actualNodeRef.*NEXT*(*bool*) := `successorNodeRef` où:

- *actualNodeRef* est la référence d'un nœud de graphe de flux de base;
- *successorNodeRef* est la référence d'un nœud successeur du nœud désigné par *actualNodeRef*;
- *bool* est un booléen spécifiant si le successeur *true* ou *false* est retourné (voir § 8.1.3).

8.2 Représentation du comportement TTCN-3 sous forme de graphes de flux

Dans la sémantique opérationnelle, on suppose que les descriptions de comportement TTCN-3 sont données sous la forme d'un ensemble de graphes de flux, autrement dit pour chaque description de comportement TTCN-3, un graphe de flux distinct doit être construit.

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 variante;

e) définitions de type de composant.

La commande de module spécifie la campagne de tests, c'est-à-dire l'ordre d'exécution (éventuellement répétitif) des tests élémentaires effectifs. Les définitions de test élémentaire définissent le comportement du composant MTC. Les fonctions structurent le comportement. Elles sont exécutées par la commande de module ou par les composants de test. Les variantes sont utilisées pour la définition d'un comportement par défaut ou sous forme de fonctions pour structurer le comportement. Les définitions de type de composant sont supposées être des descriptions de comportement car elles spécifient la création, la déclaration et l'initialisation de ports, de constantes, de variables et de temporisations pendant la création d'une instance d'un type de composant.

8.2.1 Procédure de construction d'un graphe de flux

Les graphes de flux présentés sur les Figures 18 à 22 et les segments de graphe de flux présentés au § 8 sont uniquement des modèles. Ils incluent des crochets '<' et '>' à l'intérieur desquels une information doit être fournie afin de produire un graphe de flux ou un segment de graphe de flux concret.

La construction de la représentation d'un module TTCN-3 sous forme de graphe de flux se fait en trois étapes:

1) pour chaque instruction TTCN-3 dans la commande de module et dans les définitions de test élémentaire, de variante, de fonction et de type de composant, un segment de graphe de flux concret est construit;

2) pour la commande de module et pour chaque définition de test élémentaire, de variante, de fonction et de type de composant, un graphe de flux concret (avec les nœuds de référence) est construit;

3) dans une procédure par étapes, tous les nœuds de référence des graphes de flux concrets sont remplacés par les définitions de segment de graphe de flux correspondantes jusqu'à ce que tous les graphes de flux incluent uniquement un nœud de début, des nœuds de fin et des nœuds de base.

NOTE 1 – Les nœuds de base des graphes de flux décrivent des unités d'exécution indivisibles de base. La sémantique opérationnelle du comportement TTCN-3 est fondée sur l'interprétation de ces nœuds. Le paragraphe 8.6 présente les méthodes d'exécution applicables uniquement à ces nœuds.

Le remplacement d'un nœud de référence par la définition de segment de graphe de flux correspondante peut conduire à la présence de parties non connectées dans un graphe de flux, c'est-à-dire de parties qui ne peuvent pas être atteintes depuis le nœud de début en suivant les arcs du graphe de flux. La sémantique opérationnelle ignorera les parties non connectées d'un graphe de flux.

NOTE 2 – Une partie non connectée d'un graphe de flux résulte de la procédure de remplacement mécanique. Pour la construction d'une représentation de graphe de flux optimale, il faut aussi prendre en considération les différentes combinaisons d'instructions TTCN-3. Toutefois, le but de la présente Recommandation est de définir une sémantique correcte et complète mais pas une représentation de graphe de flux optimale.

8.2.2 Représentation de la commande de module sous forme de graphe de flux

Schématiquement, la structure syntaxique d'un module TTCN-3 est la suivante:

```
module <identifier> <module-definitions-part> control <statement-block>
```

Pour la représentation du comportement sous forme de graphe de flux, seules les informations suivantes sont utiles:

```
module <identifier> <statement-block>
```

Ces informations sont comparables à la définition d'une fonction et la représentation de la commande de module sous forme de graphe de flux est analogue à la représentation d'une fonction sous forme de graphe de flux (voir § 8.2.4). La sémantique accédera au graphe de flux représentant la commande de module au moyen du nom du module.

NOTE – La signification de la partie définitions d'un module sort du cadre de cette sémantique opérationnelle. Les paramètres de module sont définis comme étant des constantes globales au moment de l'exécution. Les références aux paramètres de module doivent être remplacées par leurs valeurs concrètes au niveau syntaxique (voir § 8.3).

Le schéma de la représentation de la commande de module sous forme de graphe de flux est illustré sur la Figure 18. Le nom de graphe de flux `control` identifie le graphe de flux représentant la commande de module. Les nœuds du graphe de flux ont des commentaires associés en décrivant la signification. Le nœud de référence `<stop-entity-op>` couvre le cas où aucune opération **`stop`** explicite n'est spécifiée, autrement dit, dans la sémantique opérationnelle, on suppose qu'une opération **`stop`** est implicitement ajoutée.

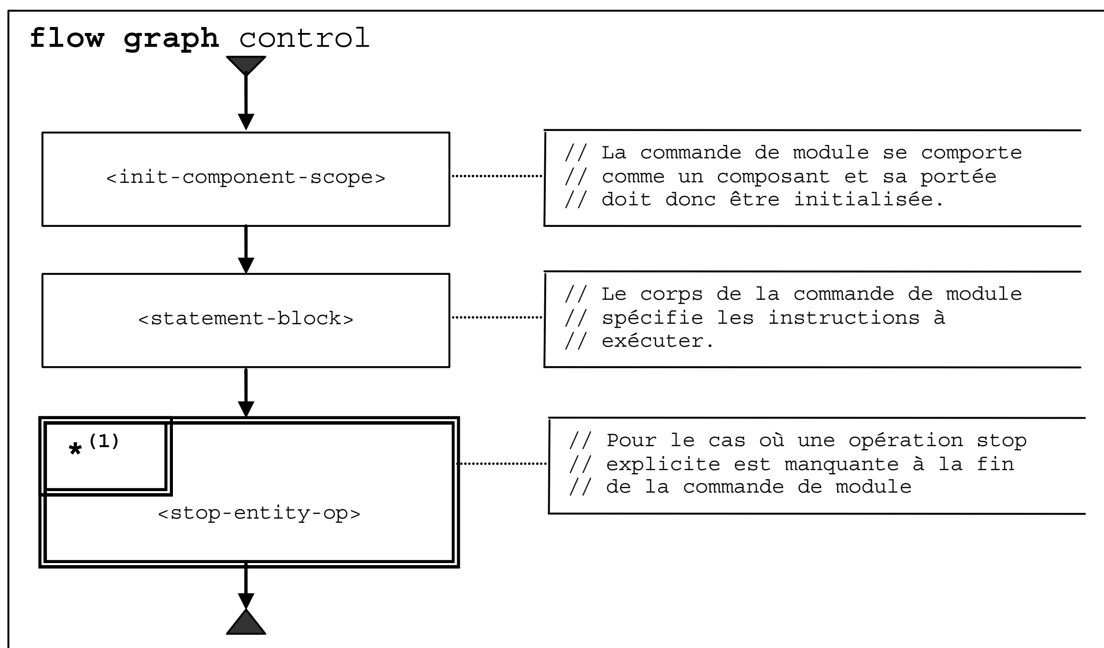


Figure 18/Z.143 – Représentation de la commande de module sous forme de graphe de flux

8.2.3 Représentation d'un test élémentaire sous forme de graphe de flux

Schématiquement, la structure syntaxique d'une définition de test élémentaire TTCN-3 est la suivante:

```
testcase <identifier> (<parameter>) <testcase-interface> <statement-block>
```

La partie `<testcase-interface>` ci-dessus renvoie aux clauses **`runs on`** (obligatoire) et **`system`** (facultative) de la définition de test élémentaire. La description d'un test élémentaire sous forme de graphe de flux décrit le comportement du composant MTC. Les informations fournies par la partie `<testcase-interface>` ne sont pas utiles pour le composant MTC. Elles seront utilisées par l'instruction **`execute`**, mais n'ont pas besoin d'être représentées dans le graphe de flux du test élémentaire. Par conséquent, pour le graphe de flux, seules les informations suivantes sont utiles:

```
testcase <identifier> (<parameter>) <statement-block>
```

Le schéma de la représentation d'un test élémentaire sous forme de graphe de flux est illustré sur la Figure 19. Le nom de graphe de flux `<identifier>` renvoie au nom du test élémentaire représenté. Les nœuds du graphe de flux ont des commentaires associés en décrivant la signification. Le nœud de référence `<stop-entity-op>` couvre le cas où aucune opération **`stop`** explicite pour le composant MTC n'est spécifiée, autrement dit, dans la sémantique opérationnelle, on suppose qu'une opération **`stop`** est implicitement ajoutée.

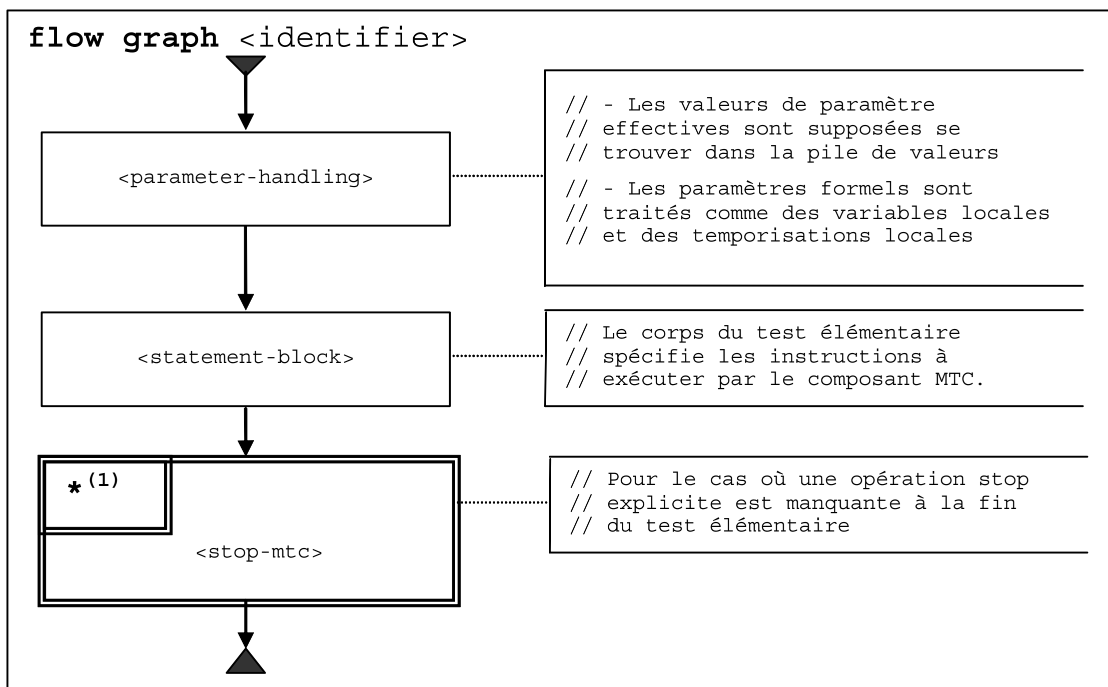


Figure 19/Z.143 – Représentation d'un test élémentaire sous forme de graphe de flux

8.2.4 Représentation d'une fonction sous forme de graphe de flux

Schématiquement, la structure syntaxique d'une fonction TTCN-3 est la suivante:

```
function <identifier> (<parameter>) [<function-interface>] <statement-block>
```

La partie `<function-interface>` facultative ci-dessus renvoie aux clauses **`runs on`** et **`return`** de la définition de la fonction. Les informations fournies par la partie `<function-interface>` ne sont pas utiles pour la description du comportement. Elles seront utilisées pour les vérifications de la sémantique statique, mais n'ont pas besoin d'être représentées dans le graphe de flux. Par conséquent, pour le graphe de flux, seules les informations suivantes sont utiles:

```
function <identifier> (<parameter>) <statement-block>
```

La sémantique accédera aux graphes de flux représentant les fonctions au moyen du nom des fonctions.

Le schéma de la représentation d'une fonction sous forme de graphe de flux est illustré sur la Figure 20. Le nom de graphe de flux `<identifier>` renvoie au nom de la fonction représentée. Le nœud de référence `<return-without-value>` couvre le cas où aucune instruction **`return`** explicite n'est spécifiée, autrement dit, dans la sémantique opérationnelle, on suppose qu'une instruction **`return`** est implicitement ajoutée.

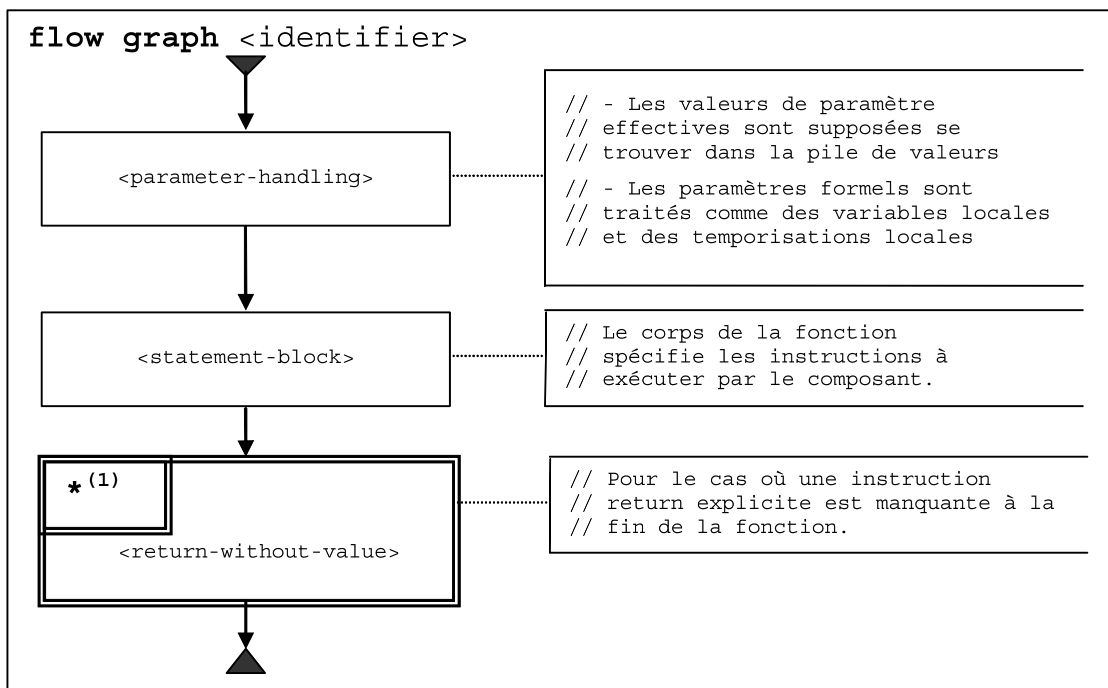


Figure 20/Z.143 – Représentation d'une fonction sous forme de graphe de flux

8.2.5 Représentation d'une variante sous forme de graphe de flux

Schématiquement, la structure syntaxique d'une variante TTCN-3 est la suivante:

```
altstep <identifier> (<parameter>) [<altstep-interface>]
                  <constant-variable-timer-declarations>
                        { <receiving-branch> | <else-branch> }*
```

La partie `<altstep-interface>` facultative ci-dessus renvoie à la clause **`runs on`** de la définition de la variante. Les informations fournies par la partie `<altstep-interface>` ne sont pas utiles pour la description du comportement. Elles seront utilisées pour les vérifications de la sémantique statique, mais n'ont pas besoin d'être représentées dans le graphe de flux. Par conséquent, pour le graphe de flux, seules les informations suivantes sont utiles:

```
altstep <identifier> (<parameter>) [<altstep-interface>]
                  <constant-variable-timer-declarations>
                        { <receiving-branch> }*
                        [ <else-branch> ]
```

NOTE – Seules les alternatives jusqu'à la première branche else, y compris cette branche, sont prises en considération. Les branches qui suivent la première branche else ne peuvent pas être atteintes.

La sémantique accédera aux graphes de flux représentant les variantes au moyen du nom des variantes.

Le schéma de la représentation d'une variante sous forme de graphe de flux est illustré sur la Figure 21. Le nom de graphe de flux `<identifier>` renvoie au nom de la variante représentée. Le nœud de référence `<successful-altstep-termination>` couvre le cas où la variante se termine après la sélection et l'exécution d'une alternative. Le nœud de référence `<unsuccessful-altstep-termination>` spécifie le cas où aucune alternative de la variante n'a été exécutée.

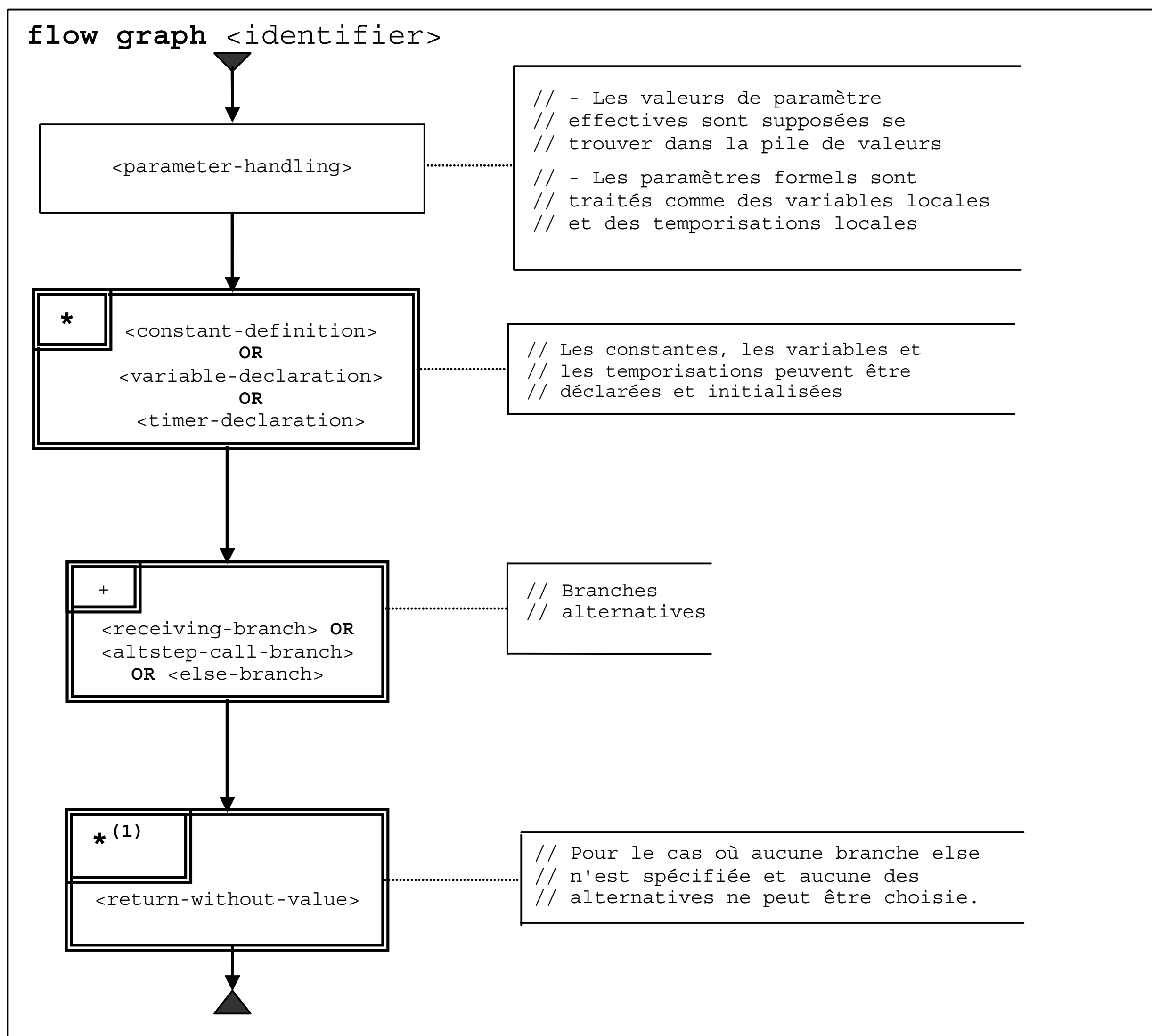


Figure 21/Z.143 – Représentation d'une variante sous forme de graphe de flux

8.2.6 Représentation d'une définition de type de composant sous forme de graphe de flux

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 de flux représentant les types de comportement au moyen du nom de ces types.

Le schéma de la représentation d'une définition de type de composant sous forme de graphe de flux est illustré sur la Figure 22. Le nom de graphe de flux `<identifier>` renvoie au nom du type de composant représenté.

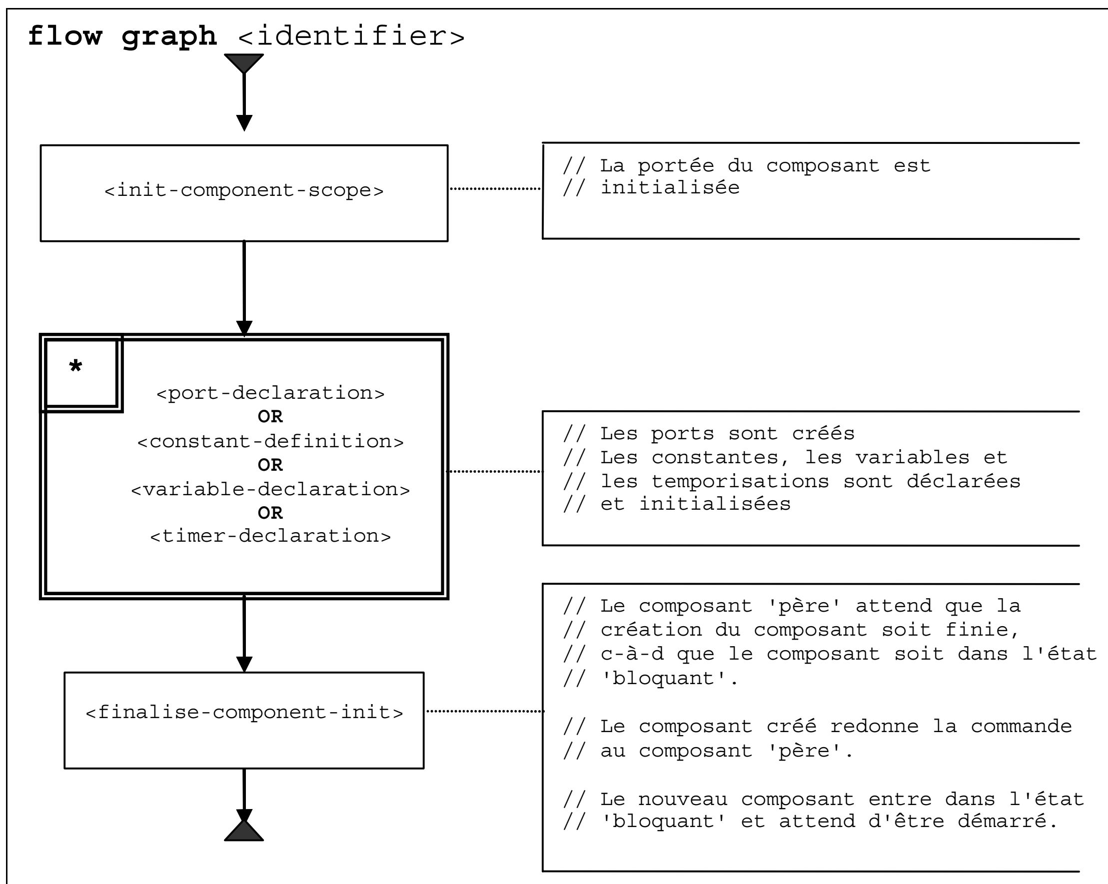


Figure 22/Z.143 – Représentation d'une définition de type de composant sous forme de graphe de flux

8.2.7 Extraction du nœud de début d'un graphe de flux

Pour l'extraction de la référence au nœud de début d'un graphe de flux, la fonction suivante est requise:

```
La fonction GET-FLOW-GRAPH:          GET-FLOW-GRAPH (flow-graph-identifier)
```

La fonction retourne une référence au nœud de début du graphe de flux dénommé *flow-graph-identifier*. L'identificateur *flow-graph-identifier* renvoie au nom du module de commande, aux noms de test élémentaire, aux noms de fonctions et aux définitions de type de composant.

8.3 Définition des états concernant les modules TTCN-3

Pendant l'interprétation des graphes de flux représentant le comportement TTCN-3, des *états de module* sont manipulés. Un état de module est un état structuré comportant plusieurs sous-états qui décrivent les états des composants de test et des ports. Le présent paragraphe décrit l'état de module, les états de composant et les états de port. Il définit en outre les fonctions permettant d'extraire des informations des états et de manipuler les états.

8.3.1 Etat de module

Comme indiqué sur la Figure 23, un état de module est structuré en *ALL-ENTITY-STATES*, *ALL-PORT-STATES*, *MTC*, *TC-VERDICT*, *DONE* et *SNAP-ACTIVE*. *ALL-ENTITY-STATES* décrit l'état de la commande de module et, pendant l'exécution d'un test élémentaire, les états des composants de test instanciés. *ALL-PORT-STATES*, *MTC* et *TC-VERDICT* s'appliquent uniquement pendant l'exécution d'un test élémentaire. *ALL-PORT-STATES* décrit les états des différents ports, *MTC* contient une référence au composant de test principal (MTC), *TC-VERDICT* contient le verdict de test global effectif d'un test élémentaire. *DONE* est une liste de tous les composants de test arrêtés pendant l'exécution d'un test élémentaire et *SNAP-ACTIVE* est utilisé dans le cadre de l'instantané du composant MTC. *SNAP-ACTIVE* contient le nombre de composants de test actifs lorsque le composant MTC prend un instantané. Il est utilisé pour l'évaluation des opérations **`all component.done`** et **`all component.running`**.

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 commande de module (*M-CONTROL* sur la Figure 23) est traité comme un composant de test normal et son état est le premier élément de *ALL-ENTITY-STATES* de l'état de module.

ALL-ENTITY-STATES						ALL-PORT-STATES					MTC	TC-VERDICT	DONE	SNAP-ACTIVE
	M-CONTROL	ES_1	...	ES_n			P_1	...	P_n					

Figure 23/Z.143 – Structure d'un état de module

NOTE 2 – Les états de port peuvent être considérés comme faisant partie des états d'entité. Grâce à `connect` et `map`, les ports sont visibles par les autres composants et cette sémantique opérationnelle traite donc les ports au niveau sommital d'un état de module.

8.3.1.1 Accès à l'état de module

MTC, *TC-VERDICT* et *SNAP-ACTIVE* font partie d'un état de module et sont traitées comme des variables globales, autrement dit les mots clés *MTC* et *TC-VERDICT* peuvent être utilisés pour extraire et modifier les valeurs de l'état de module correspondant.

NOTE 1 – Il existe un seul état de module pendant l'interprétation d'un module TTCN-3. Par conséquent, les mots clés *MTC* et *TC-VERDICT* peuvent être considérés comme des identificateurs uniques à l'échelle globale pour la procédure d'évaluation.

Pour le traitement des listes *ALL-ENTITY-STATES*, *ALL-PORT-STATES* et *DONE*, on peut utiliser les opérations applicables aux listes *add*, *append*, *delete*, *member*, *first*, *length*, *next*, *random* et *change*. Ces opérations ont la signification suivante:

- *myList*.*add*(*item*) ajoute *item* comme premier élément dans la liste *myList*;
- *myList*.*append*(*item*) ajoute *item* comme dernier élément dans la liste *myList*;
- *myList*.*delete*(*item*) supprime *item* de la liste *myList*;
- *myList*.*member*(*item*) retourne **`true`** si *item* est un élément de la liste *myList*, et **`false`** dans le cas contraire;
- *myList*.*first*() retourne le premier élément de *myList*;
- *myList*.*length*() retourne la longueur de *myList*;
- *myList*.*next*(*item*) retourne l'élément qui suit *item* dans *myList*, ou **`NULL`** si *item* est le dernier élément de *myList*;
- *MyList*.*random*(<*condition*>) retourne un élément de *myList* pris aléatoirement parmi les éléments qui remplissent la condition booléenne <*condition*> ou **`NULL`** si aucun élément de *myList* ne remplit la condition <*condition*>;
- *MyList*.*change*(<*operation*>) permet d'appliquer l'opération <*operation*> à tous les éléments de *myList*.

NOTE 2 – Les opérations *random* et *change* ne sont pas des opérations courantes applicables aux listes. Elles sont introduites pour expliquer la signification des mots clés `all` et `any` dans les opérations TTCN-3.

8.3.2 Etats d'entité

Les états d'entité servent à décrire les états effectifs de la commande de module et des composants de test. Dans l'état de module, les états d'entité sont traités dans la liste *ALL-ENTITY-STATES*. La structure d'un état d'entité est illustrée sur la Figure 24.

<identifier>	STATUS	CONTROL-STACK	DEFAULT-LIST	DEFAULT-POINTER	VALUE-STACK	E-VERDICT	TIMER-GUARD	DATA-STATE	TIMER-STATE	SNAP-DONE

Figure 24/Z.143 – Structure d'un état d'entité

<*identifier*> est l'identificateur unique d'une entité du système de test, à savoir la commande de module ou un composant de test. Ces identificateurs uniques sont créés implicitement pour la commande de module, le composant **`mtc`** et le système de test **`system`** lorsqu'un module commence à être exécuté ou qu'un test élémentaire est exécuté au moyen de l'instruction **`execute`**. Ils servent à identifier et à adresser les entités du système de test, par exemple dans le cas d'opérations **`send`** avec des clauses **`to`** ou d'opérations **`receive`** avec des clauses **`from`**.

STATUS décrit si la commande de module ou un composant de test est **`ACTIVE`**, **`SNAPSHOT`**, **`REPEAT`** ou **`BLOCKED`**. La commande de module est bloquée pendant l'exécution d'un test élémentaire. Les composants de test peuvent être bloqués pendant la création d'autres composants de test, à savoir pendant l'exécution d'une opération **`create`**. Le statut **`SNAPSHOT`** indique que le composant est actif, mais qu'il se trouve dans la phase d'évaluation d'un instantané. Le statut

REPEAT indique que le composant est actif et qu'il se trouve dans une instruction **alt** qui doit être réévaluée en raison d'une instruction **repeat**.

CONTROL-STACK est une pile de références à des nœuds de graphe de flux. L'élément sommital de la pile *CONTROL-STACK* est le nœud suivant du graphe de flux qui doit être interprété. La pile est nécessaire pour modéliser les appels de fonction de manière adéquate.

DEFAULT-LIST est une liste de comportements par défaut activés, autrement dit une liste de pointeurs qui renvoient aux nœuds de début des comportements par défaut activés. La liste est dans l'ordre inverse de l'activation: le comportement par défaut qui a été activé en premier est le dernier élément de la liste.

Pendant l'exécution du mécanisme par défaut, *DEFAULT-POINTER* renvoie au comportement par défaut suivant qui doit être évalué en cas d'échec de la terminaison du comportement par défaut effectif.

VALUE-STACK est une pile de valeurs de tous les types possibles qui permet un stockage 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 l'opération **mtc** seront placés dans la pile *VALUE-STACK*. Outre les valeurs de tous les types de données connus dans un module, on définit la valeur spéciale **MARK** comme faisant partie de l'alphabet de la pile. Au moment de quitter une unité de portée, la valeur **MARK** sert à nettoyer la pile *VALUE-STACK*.

E-VERDICT contient le verdict local effectif d'un composant de test. Il est ignoré si l'état d'entité représente la commande de module.

TIMER-GUARD représente la temporisation de protection spéciale requise pour la durée d'exécution des tests élémentaires et la durée des opérations d'appel. Il est modélisé sous la forme d'un lien de temporisation (voir § 8.3.2.4 et Figure 28).

DATA-STATE est considéré comme une liste de listes de liens de variable. La structure de la liste des listes reflète les unités de portée imbriquées dues aux appels de fonction imbriqués. Chaque liste présente dans la liste de listes de liens de variable décrit les variables connues et leurs valeurs dans une certaine unité de portée. L'entrée dans une unité de portée et la sortie d'une unité de portée correspondent respectivement à l'ajout et à la suppression d'une liste de liens de variable dans *DATA-STATE*. On trouvera au § 8.3.2.2 une description de la partie *DATA-STATE* d'un état d'entité.

TIMER-STATE est considéré comme une liste de listes de liens de temporisation. La structure de la liste des listes reflète les unités de portée imbriquées dues aux appels de fonction imbriqués. Chaque liste présente dans la liste de listes de liens de temporisation décrit les temporisations connues et leurs statuts dans une certaine unité de portée. L'entrée dans une unité de portée et la sortie d'une unité de portée correspondent respectivement à l'ajout et à la suppression d'une liste d'états de temporisation dans *timer-state*. On trouvera au § 8.3.2.4 une description de la partie *timer-state* d'un état d'entité.

SNAP-DONE prend en charge la sémantique d'instantané des composants de test. Lorsqu'un instantané est pris, une copie de la liste *DONE* de l'état de module sera affectée à *SNAP-DONE*, autrement dit *SNAP-DONE* est la liste des identificateurs des composants arrêtés.

8.3.2.1 Accès aux états d'entité

<*identifier*> est l'identificateur unique d'un état d'entité, qui peut servir à accéder au composant représenté par l'état d'entité et aux différentes parties de l'état d'entité.

Les parties *STATUS*, *DEFAULT-POINTER*, *E-VERDICT* et *TIMER-GUARD* d'un état d'entité sont traitées comme des variables qui sont visibles au niveau global, autrement dit les valeurs de *STATUS*, *DEFAULT-POINTER* et *E-VERDICT* peuvent être extraites ou modifiées au moyen de la notation à points, par exemple *myEntity*.*STATUS*, *myEntity*.*DEFAULT-POINTER* et *myEntity*.*E-VERDICT*, où *myEntity* désigne un état d'entité.

NOTE – Dans ce qui suit, on suppose qu'on peut utiliser la notation à points avec des références et des identificateurs uniques. Par exemple, dans *myEntity*.*STATUS*, *myEntityState* peut être un pointeur vers un état d'entité ou être la valeur du champ <*identifier*>.

Les parties *CONTROL-STACK*, *DEFAULT-LIST* et *VALUE-STACK* d'un état d'entité *myEntity* peuvent être adressées au moyen de la notation à points *myEntity*.*CONTROL-STACK*, *myEntity*.*DEFAULT-LIST* et *myEntity*.*VALUE-STACK*.

On peut accéder aux parties *CONTROL-STACK* et *VALUE-STACK* et les manipuler au moyen des opérations applicables aux piles *push*, *pop*, *top*, *clear* et *clear-until*. Ces opérations ont la signification suivante:

- myStack.push(item) place item dans myStack;
- myStack.pop() supprime l'élément sommital de myStack;
- myStack.top() retourne l'élément sommital de myStack ou **NULL** si myStack est vide;
- myStack.clear() nettoie myStack, autrement dit supprime tous les éléments de myStack;

- myStack.clear-until(item) supprime les éléments de myStack jusqu'à l'élément item ou jusqu'à ce que myStack soit vide.

On peut accéder à la partie *DEFAULT-LIST* et la manipuler au moyen des opérations applicables aux listes *add*, *append*, *delete*, *member*, *first*, *length*, *next*, *random* et *change*. La signification de ces opérations est donnée au § 8.3.1.1.

Pour la création d'un nouvel état d'entité, la fonction *NEW-ENTITY* est supposée être disponible:

- *NEW-ENTITY* (*entityIdentifier, flow-graph-node-reference*);

crée un nouvel état d'entité et retourne sa référence. Les parties du nouvel état d'entité ont les valeurs suivantes:

- <identifier> est mis à *entityIdentifier* et doit être un identificateur unique au niveau global;
- *STATUS* est mis à **ACTIVE**;
- *flow-graph-node-reference* est le seul élément (sommital) de la pile *CONTROL-STACK*;
- *DEFAULT-LIST* est une liste vide;
- *DEFAULT-POINTER* a la valeur **NULL**;
- *VALUE-STACK* est une pile vide;
- *E-VERDICT* est mis à **none**;
- *TIMER-GUARD* est un nouveau lien de temporisation (voir § 8.3.2.4) dont le nom est GUARD, dont le statut est **IDLE** et pour lequel aucune durée par défaut n'est définie;
- *DATA-STATE* est une liste vide;
- *TIMER-STATE* est une liste vide;
- *SNAP-DONE* est une liste vide.

Pendant la traversée d'un graphe de flux, la pile *CONTROL-STACK* change souvent de valeur de la même manière: l'élément sommital est supprimé de la pile *CONTROL-STACK* et le nœud successeur du nœud supprimé est placé dans la pile *CONTROL-STACK*. Cette série d'opérations applicables aux piles est encapsulée dans la fonction *NEXT-CONTROL*:

```
myEntity.NEXT-CONTROL(myBool) {
      successorNode := myEntity.CONTROL-STACK.NEXT(myBool).top();
      myEntity.CONTROL-STACK.pop();
      myEntity.CONTROL-STACK.push(successorNode);
}
```

8.3.2.2 Etat de données et lien de variable

Comme indiqué sur la Figure 25, l'état de données *DATA-STATE* d'un état d'entité est une liste de listes de liens de variable. Chaque liste de liens de variable définit les liens de variable dans une certaine unité de portée. L'ajout d'une nouvelle liste de liens de variable correspond à l'entrée dans une nouvelle unité de portée, par exemple lorsqu'une fonction est appelée. La suppression d'une liste de liens de variable correspond à la sortie d'une unité de portée, par exemple lorsqu'une fonction exécute une instruction **return**.

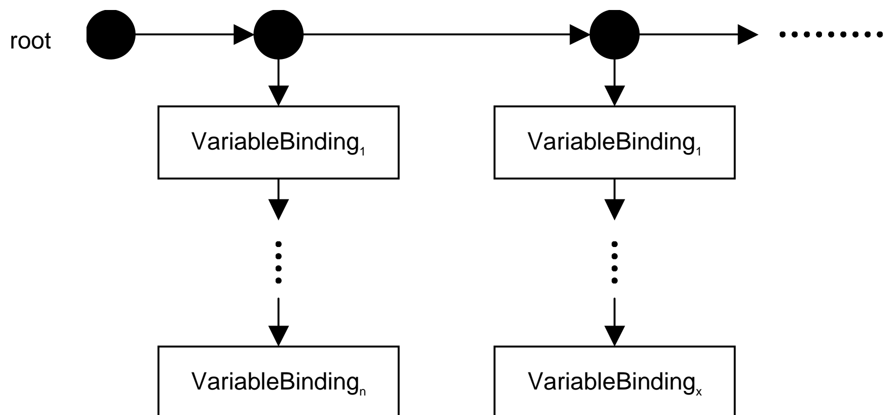


Figure 25/Z.143 – Structure de la partie *DATA-STATE* d'un état d'entité

La structure d'un lien de variable est illustrée sur la Figure 26. Une variable a un nom, un emplacement <*location*> et une valeur *VALUE*. *VAR NAME* identifie une variable dans une unité de portée. <*location*> est un identificateur unique de l'endroit où est stockée la valeur de la variable. La partie *VALUE* d'un lien de variable décrit la valeur effective d'une variable.

NOTE – Un identificateur d'emplacement unique doit être fourni automatiquement lorsqu'une variable est déclarée.

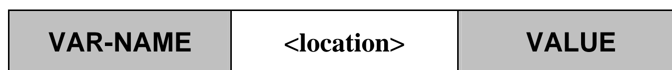


Figure 26/Z.143 – Structure d'un lien de variable

La distinction entre nom de variable et emplacement a été faite afin de pouvoir modéliser les appels de fonction et l'exécution des tests élémentaires avec paramétrage par valeur ou par référence de façon appropriée:

a) un paramètre transmis par valeur est traité comme la déclaration d'une nouvelle variable, autrement dit un nouveau lien de variable est ajouté dans la liste des liens de variable de la portée de la fonction appelée ou du test élémentaire exécuté. Le nouveau lien de variable utilise le nom de paramètre formel *VAR-NAME*, reçoit un nouvel emplacement et obtient la valeur qui est transmise dans la fonction ou le test élémentaire.

b) un paramètre transmis par référence conduit aussi à un nouveau lien de variable dans la portée de la fonction appelée ou du test élémentaire exécuté. Le nouveau lien de variable utilise également le nom de paramètre formel *VAR-NAME*, mais ne reçoit aucun nouvel emplacement et aucune nouvelle valeur. Le nouveau lien de variable obtient une copie de l'emplacement <*location*> et de la valeur *VALUE* de la variable qui est transmise par référence.

Lors de la mise à jour de la valeur d'une variable, par exemple dans le cas d'une affectation faite à une variable, le nom de la variable sert à identifier un emplacement et tous les liens de variable associés au même emplacement sont mis à jour simultanément. Par conséquent, au moment de quitter l'unité de portée, la liste des variables appartenant à cette unité de portée peut être supprimée sans autre mise à jour. Du fait de la procédure de mise à jour, les variables transmises par référence ont automatiquement la valeur correcte.

8.3.2.3 Accès aux états de données

La valeur d'une variable peut être extraite au moyen de la notation à points *myEntity.myVar.VALUE*, où *myEntity* renvoie à un état d'entité et *myVar* est le nom d'une variable.

Pour le traitement des variables et des portées de variable, on considère que les fonctions suivantes sont définies:

a) la fonction *VAR-SET*: *myEntity.VAR-SET* (*myVar*, *myVal*)

met la partie *VALUE* de la variable *myVar* se trouvant dans la portée effective d'une entité *myEntity* à *myVal*. En outre, la partie *VALUE* de toutes les variables associées au même emplacement que la variable *myVar* sera également mise à *myVal*;

b) la fonction *INIT-VAR*: *myEntity.INIT-VAR* (*myVar, myVal*)

crée un nouveau lien de variable pour une variable *myVar* avec la valeur initiale *myVal* dans l'unité de portée effective d'une entité *myEntity*. L'utilisation du mot clé **NONE** pour *myVal* signifie qu'une variable avec une valeur initiale non définie est créée. Une valeur nouvelle et unique d'emplacement <*location*> est créée automatiquement;

c) la fonction *GET-VAR-LOC*: *myEntity.GET-VAR-LOC* (*myVar*)

extrait l'emplacement de la variable *myVar* détenue par *myEntity*;

d) la fonction *INIT-VAR-LOC*: *myEntity.INIT-VAR-LOC* (*myVar,myLoc*)

crée un nouveau lien de variable pour une variable *myVar* avec à l'emplacement *myLoc* dans l'unité de portée effective de *myEntity*. La variable sera initialisée avec la valeur d'une autre variable associée à l'emplacement *myLoc*;

NOTE – Les variables associées au même emplacement résultent d'un paramétrage par référence. Compte tenu du traitement des paramètres par référence décrit au § 8.3.2.2, toutes les variables associées au même emplacement auront des valeurs identiques pendant leur durée de vie.

e) la fonction *INIT-VAR-SCOPE*: *myEntity.INIT-VAR-SCOPE* ()

initialise une nouvelle portée de variable dans l'état de données de l'entité *myEntity*, autrement dit une liste vide est ajoutée comme première liste dans la liste de listes de liens de variable;

f) la fonction *DEL-VAR-SCOPE*: *myEntity.DEL-VAR-SCOPE* ()

supprime une portée de variable de l'état de données de *myEntity*, autrement dit la première liste figurant dans la liste de listes de liens de variable est supprimée.

8.3.2.4 Etat de temporisation et lien de temporisation

Comme indiqué sur les Figures 27 et 25, l'état de temporisation *TIMER-STATE* et l'état de données *DATA-STATE* d'un état d'entité sont comparables. Dans les deux cas, il s'agit d'une liste de listes de liens et chaque liste de liens définit les liens valides dans une certaine portée. L'ajout d'une nouvelle liste correspond à l'entrée dans une nouvelle unité de portée et la suppression d'une liste de liens correspond à la sortie d'une unité de portée.

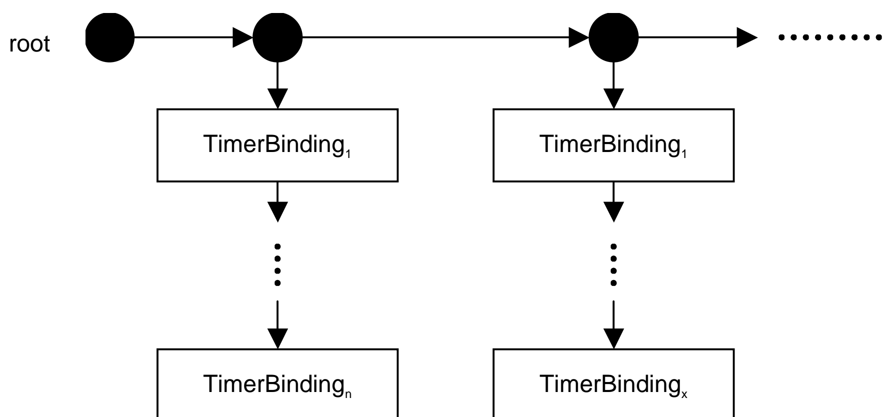


Figure 27/Z.143 – Structure de la partie *TIMER-STATE* d'un état d'entité

La structure d'un lien de temporisation est illustrée sur la Figure 28. La signification du nom *TIMER-NAME* et de l'emplacement <*location*> est analogue à la signification du nom *VAR-NAME* et de l'emplacement <*location*> pour un lien de variable (Figure 26).

TIMER-NAME	<location>	STATUS	DEF-DURATION	ACT-DURATION	TIME-LEFT	SNAP-VALUE	SNAP-STATUS

Figure 28/Z.143 – Structure d'un lien de temporisation

STATUS indique si une temporisation est active, inactive ou a expiré. Les valeurs correspondantes de *STATUS* sont **`IDLE`**, **`RUNNING`** et **`TIMEOUT`**. *DEF-DURATION* décrit la durée par défaut d'une temporisation. *ACT-DURATION* contient la durée effective avec laquelle une temporisation en cours a été lancée. *TIME-LEFT* décrit la durée effective qu'il reste à une temporisation en cours jusqu'à son expiration.

NOTE – *DEF-DURATION* est non défini si une temporisation est déclarée sans durée par défaut. *ACT-DURATION* et *TIME-LEFT* sont mis à 0.0 si une temporisation est arrêtée ou expire. Si une temporisation est déclenchée sans durée, la valeur de *DEF-DURATION* est copiée dans *ACT-DURATION*. Une erreur dynamique se produit si une temporisation est déclenchée dans durée définie.

SNAP-VALUE et *SNAP-STATUS* sont nécessaires pour prendre en charge la sémantique d'instantané de la notation TTCN-3. Lors de la prise d'un instantané, *SNAP-VALUE* reçoit la valeur effective de *ACT-DURATION* – *TIME-LEFT* et *SNAP-STATUS* reçoit la même valeur que *STATUS*. L'évaluation d'un instantané sera uniquement fondée sur les valeurs de *SNAP-VALUE* et *SNAP-STATUS*.

Une temporisation peut uniquement être transmise par référence dans des fonctions, autrement dit le mécanisme est analogue au mécanisme pour les variables décrit au § 8.3.2.2. Cela signifie qu'un nouveau lien de temporisation (avec le nom de paramètre formel) est créé et qu'il reçoit les copies de <*location*>, *STATUS*, *DEF-DURATION*, *ACT-DURATION*, *TIME-LEFT*, *SNAP-VALUE* et *SNAP-STATUS* relatives à la temporisation qui est transmise par référence. Lors de la mise à jour d'une temporisation, tous les liens de temporisation associés à la même valeur d'emplacement <*location*> sont mis à jour simultanément.

8.3.2.5 Accès aux états de temporisation

Les valeurs de *STATUS*, *DEF-DURATION*, *ACT-DURATION*, *TIME-LEFT*, *SNAP-VALUE* et *SNAP-STATUS* d'une temporisation *myTimer* peuvent être extraites au moyen de la notation à points:

- myEntity.myTimer.STATUS;
- myEntity.myTimer.DEF-DURATION;
- myEntity.myTimer.ACT-DURATION;
- myEntity.myTimer.TIME-LEFT;
- myEntity.myTimer.SNAP-VALUE;
- myEntity.myTimer.SNAP-STATUS.

L'élément *myEntity* figurant dans la notation à points renvoie à un état d'entité représentant l'état d'un composant de test ou de la commande de module qui détient la temporisation *myTimer*.

Pour changer les valeurs de *STATUS*, *DEF-DURATION*, *ACT-DURATION*, *TIME-LEFT*, *SNAP-VALUE* et *SNAP-STATUS* d'une temporisation *timer-name*, il faut utiliser l'opération générique *TIMER-SET*, par exemple:

- *myEntity.TIMER-SET(myTimer, STATUS, myVal)*.

met le statut *STATUS* de la temporisation *myTimer* se trouvant dans la portée effective de *myEntity* à la valeur *myVal*. En outre, le statut *STATUS* de toutes les temporisations associées au même emplacement que la temporisation *myTimer* sera également mis à *myVal*. La fonction *TIMER-SET* peut également servir à changer les valeurs de *DEF-DURATION*, *ACT-DURATION*, *TIME-LEFT*, *SNAP-VALUE* et *SNAP-STATUS.*

Pour le traitement des temporisations, des portées de temporisation et des instantanés, il faut définir les fonctions suivantes:

a) la fonction *INIT-TIMER*: *myEntity.INIT-TIMER* (*myTimer, myDuration*)

   crée un nouveau lien de temporisation pour une temporisation *myTimer* avec la durée par défaut *myDuration* dans la portée effective d'une entité *myEntity*. L'utilisation du mot clé **NONE** pour *myDuration* signifie qu'une temporisation sans durée par défaut est créée;

b) la fonction *GET-TIMER-LOC*: *myEntity.GET-TIMER-LOC* (*myTimer*)

   extrait l'emplacement de la temporisation *myTimer* détenue par *myEntity*;

c) la fonction *INIT-TIMER-LOC*: *myEntity.INIT-TIMER-LOC* (*myTimer, myLocation*)

   crée un nouveau lien de temporisation pour une temporisation *myTimer* avec l'emplacement *myLocation* dans l'unité de portée effective de *myEntity*. La temporisation sera initialisée avec les valeurs de *STATUS*, *DEF-DURATION*, *ACT-DURATION* et *TIME-LEFT* d'une autre temporisation associée à l'emplacement <*location*>.

   NOTE – Les temporisations associées au même emplacement résultent d'un paramétrage par référence. Compte tenu du traitement des paramètres par référence de temporisation décrit au 8.3.2.3, toutes les temporisations associées au même emplacement auront des valeurs identiques de *STATUS*, *DEF-DURATION*, *ACT-DURATION* et *TIME-LEFT* pendant leur durée de vie.

d) la fonction *INIT-TIMER-SCOPE*: *myEntity.INIT-TIMER-SCOPE* ()

   initialise une nouvelle portée de temporisation dans l'état de temporisation de l'entité *myEntity*, autrement dit une liste vide est ajoutée comme première liste dans la liste de listes de liens de temporisation;

e) la fonction *DEL-TIMER-SCOPE*: *myEntity.DEL-TIMER-SCOPE* ()

   supprime une portée de temporisation de l'état de temporisation de l'entité *myEntity*, autrement dit la première liste figurant dans la liste de listes de liens de temporisation est supprimée;

f) la fonction *SNAP-TIMER*: *myEntity.SNAP-TIMER* ()

   fait une mise à jour de *SNAP-VALUE* et *SNAP-STATUS* dans toutes les temporisations détenues par *myEntity*, à savoir:

```
myEntity.SNAP-TIMERS () {
      for all myTimer in TIMER-STATE {
            myEntity.myTimer.SNAP-VALUE := myEntity.myTimer.ACT-DURATION –
            myEntity.myTimer.TIME-LEFT;
            myEntity.myTimer.SNAP-STATUS := myEntity.myTimer.STATUS;
      }
```

8.3.3 Etats de port

Les états de port servent à décrire les états effectifs des ports. Dans un état de module, les états de port sont traités dans la liste *ALL-PORT-STATES* (voir Figure 23). La structure d'un état de port est illustrée sur la Figure 29. *PORT-NAME* renvoie au nom de port qui sert à identifier le port détenu par le composant de test *OWNER*. *STATUS* contient le statut effectif du port. Un port peut être **STARTED** ou **STOPPED**.

NOTE – Un port d'un système de test est identifié sans équivoque par le composant de test détenteur <owner> et par le nom de port <port-name> au niveau local du détenteur <owner>.

La partie *CONNECTIONS-LIST* d'un état de port conserve une trace des connexions entre les différents ports du système de test. Le mécanisme est expliqué au § 8.3.3.1.

La partie *VALUE-QUEUE* d'un état de port stocke les messages, les appels, les réponses et les exceptions qui sont reçus à ce port mais qui n'ont pas encore été consommés.

SNAP-VALUE prend en charge le mécanisme d'instantané TTCN-3. Lorsqu'un instantané est pris, le premier élément de *VALUE-QUEUE* est copié dans *SNAP-VALUE*. *SNAP-VALUE* recevra la valeur **NULL** si *VALUE-QUEUE* est vide ou si *STATUS* vaut **STOPPED**.

PORT-NAME	OWNER	STATUS	CONNECTIONS-LIST	VALUE-QUEUE	SNAP-VALUE

Figure 29/Z.143 – Structure d'un état de port

8.3.3.1 Traitement des connexions entre les ports

Pour réaliser une connexion entre deux composants de test, on connecte deux de leurs ports au moyen d'une opération **connect**. Par conséquent, un composant peut ensuite utiliser son nom de port local pour adresser la file d'attente distante. Comme indiqué sur la Figure 30, la *connexion* est représentée dans les états des deux files d'attentes connectées par une paire *REMOTE-ENTITY* et *REMOTE-PORT-NAME*. *REMOTE-ENTITY* contient l'identificateur unique du composant de test qui détient le port distant. *REMOTE-PORT-NAME* renvoie au nom local utilisé par l'élément *REMOTE-ENTITY* pour adresser la file d'attente. La notation TTCN-3 prend en charge les connexions d'un port à plusieurs ports et, par conséquent, toutes les connexions d'un port sont organisées dans une liste.

NOTE 1 – Les connexions réalisées par des opérations **map** sont également traitées dans la liste de connexions. L'opération **map**: **map**(*PTC1*:*MyPort*, **system**.*PCO1*) conduit à une nouvelle connexion (**system**, *PCO1*) dans l'état de port de *MyPort* détenu par *PTC1*. L'extrémité distante de la connexion avec *PCO1* réside dans le système SUT. Son comportement sort du cadre de cette sémantique.

NOTE 2 – La sémantique opérationnelle traite le mot clé **system** comme une adresse symbolique. Une connexion (**system**, *myPort*) figurant dans la liste des connexions d'un port indique que le port est mappé vers le port *myPort* dans l'interface du système de test.

REMOTE-ENTITY	REMOTE-PORT-NAME

Figure 30/Z.143 – Structure d'une connexion

8.3.3.2 Traitement des états de port

On peut accéder à la file d'attente des valeurs d'un état de port et la manipuler en utilisant les opérations connues applicables aux files d'attente *enqueue*, *dequeue*, *first* et *clear*. L'utilisation de la fonction *GET-PORT* ou *GET-REMOTE* permet de faire référence à la file d'attente à laquelle on doit accéder.

NOTE 1 – Les opérations applicables aux files d'attente *enqueue*, *dequeue*, *first* et *clear* ont la signification suivante:

- *myQueue*.*enqueue*(*item*) place *item* comme dernier élément dans *myQueue*;
- *myQueue*.*dequeue*() supprime le premier élément de *myQueue*;
- *myQueue*.*first*() retourne le premier élément de *myQueue* ou **NULL** si *myQueue* est vide;
- *myQueue*.*clear*() supprime tous les éléments de *myQueue*.

Le traitement des états de port est pris en charge par les fonctions suivantes:

a) la fonction *NEW-PORT*: *NEW-PORT* (*myEntity, myPort*)

crée un nouveau port et retourne sa référence. Le nouveau port est détenu par *myEntity* et porte le nom *myPort* du port identifié par le composant de test *myEntity* et le nom de port *myPort*. Le statut du nouveau port est **STARTED**. *CONNECTIONS-LIST* et *VALUE-QUEUE* sont vides. *SNAP-VALUE* a la valeur **NULL** (autrement dit la file d'attente d'entrée du nouveau port est vide);

b) la fonction *GET-PORT*: *GET-PORT* (*myEntity, myPort*)

retourne une référence au port identifié par le composant de test *myEntity* qui détient le port et le nom de port *myPort*;

c) la fonction *GET-REMOTE-PORT*: *GET-REMOTE-PORT* (*myEntity, myPort, myRemoteEntity*)

retourne la référence au port qui est détenu par le composant de test *myRemoteEntity* et qui est connecté à un port identifié par *myEntity* et *myPort*. L'adresse symbolique **SYSTEM** est retournée, si le port distant est mappé vers un port de l'interface du système de test;

NOTE 2 – La fonction *GET-REMOTE-PORT* retourne **NULL** s'il n'y a pas de port distant ou si le port distant ne peut pas être identifié sans équivoque. La valeur spéciale **NONE** peut servir de valeur pour le paramètre *myRemoteEntity* si l'entité distante n'est pas connue ou pas requise, autrement dit s'il existe une seule connexion de un à un pour ce port.

d) Le statut *STATUS* d'un port est traité comme une variable. Pour l'adresser, on peut qualifier *STATUS* avec un appel *GET-PORT*:

GET-PORT(*myEntity*, *myPort*).*STATUS*

e) la fonction *ADD-CON*: *ADD-CON* (*myEntity, myPort, myRemoteEntity*, *myRemotePort*)

ajoute la connexion (*myRemoteEntity*, *myRemotePort*) à la liste des connexions du port *myPort* détenu par *myEntity*;

f) la fonction *DEL-CON*: *DEL-CON* (*myEntity, myPort, myRemoteEntity*, *myRemotePort*)

supprime la connexion (*myRemoteEntity*, *myRemotePort*) de la liste des connexions du port *myPort* détenu par *myEntity*;

g) la fonction *SNAP-PORTS*: *SNAP-PORTS* (*myEntity*)

met à jour *SNAP-VALUE* pour tous les ports détenus par *myEntity*, à savoir:

```
SNAP-PORTS (myEntity) {
    for all ports p         /* dans l'état de module */ {
        if (p.OWNER == myEntity) {
            if (p.STATUS == STOPPED) {
             p. SNAP-VALUE := NULL;
            }
            else {
             p. SNAP-VALUE := p.first()
            }
        }
    }
}
```

8.3.4 Fonctions générales pour le traitement des états de module

Dans la sémantique opérationnelle, on suppose l'existence des fonctions suivantes pour le traitement des états de module.

NOTE 1 – Pendant l'interprétation d'un module TTCN-3, il existe un seul état de module. On suppose que les parties de l'état de module sont stockées dans des variables globales et non dans un objet de données complexe. Par conséquent, les fonctions suivantes sont supposées s'appliquer à des variables globales et non à un objet d'état de module particulier.

a) La fonction *DEL-ENTITY*: *DEL-ENTITY*(*myEntity*)

supprime une entité ayant l'identificateur unique *myEntity*. La suppression inclut:

- la suppression de l'état d'entité de *myEntity*;
- la suppression de tous les ports détenus par *myEntity*;
- la suppression de toutes les connexions dans lesquelles *myEntity* intervient.

b) La fonction *UPDATE-REMOTE-REFERENCES*:

UPDATE-REMOTE-REFERENCES (*source, target*)

met à jour les variables et les temporisations associées au même emplacement dans les deux entités. Les valeurs utilisées pour la mise à jour sont les valeurs des variables et des temporisations détenues par *source*.

NOTE 2 – La fonction *UPDATE-REMOTE-REFERENCES* est utilisée pendant la terminaison des tests élémentaires. Elle permet de mettre à jour les variables de la commande de module, qui sont transmises sous forme de paramètres par référence aux tests élémentaires.

8.4 Messages, appels de procédure, réponses et exceptions

L'échange d'informations entre composants de test ainsi qu'entre des composants de test et le système SUT repose sur des *messages*, *appels de procédure*, *réponses aux appels de procédure* et *exceptions*, qui, aux fins de communication, doivent être construits, codés et décodés. Le codage concret, à savoir le mappage de types de données TTCN-3 vers des bits et des octets, et le décodage, à savoir le mappage de bits et d'octets vers des types de données TTCN-3, sortent du cadre de la sémantique opérationnelle. Dans la présente Recommandation, les *messages*, *appels de procédure*, *réponses aux appels de procédure* et *exceptions* sont traités au niveau conceptuel.

8.4.1 Messages

Les messages sont utilisés dans la communication en mode message. Les valeurs de tous les types de données (prédéfinis ou définis par l'utilisateur) peuvent être échangées entre les entités qui communiquent. Comme indiqué sur la Figure 31, la sémantique opérationnelle traite un message comme un objet structuré comportant trois parties: *sender*, *type* et *value*. La partie *sender* identifie l'expéditeur du message, la partie *type* spécifie le type du message et la partie *value* définit la valeur du message.

sender	type	value

Figure 31/Z.143 – Structure d'un message

NOTE – La sémantique opérationnelle présente uniquement un modèle pour les concepts de la notation TTCN-3. La question de savoir si les informations *sender* doivent ou non être envoyées et/ou reçues ainsi que les modalités d'envoi et de réception dépendent de l'implémentation du système de test, par exemple dans certains cas les informations relatives à l'expéditeur peuvent être contenues dans la partie *value* du message et, par conséquent, ne pas constituer une partie à part entière de la structure du message.

8.4.2 Appels de procédure et réponses

Les appels de procédure et les réponses aux appels de procédure sont utilisés dans la communication en mode procédure. Ils sont définis comme des valeurs d'un enregistrement dont les parties représentent les paramètres. La sémantique opérationnelle traite également les appels de procédure et les réponses aux appels de procédure comme des valeurs de types structurés. La structure d'un appel de procédure et la structure d'une réponse sont présentées sur les Figures 32 et 33.

<table>
<tr><th>sender</th><th>procedure-reference</th><th colspan="5">parameter-part</th></tr>
<tr><td></td><td></td><td></td><td>in-or-inout-parameter$_1$</td><td>…</td><td>in-or-inout-parameter$_n$</td><td></td></tr>
</table>

Figure 32/Z.143 – Structure d'un appel de procédure

<table>
<tr><th>sender</th><th>procedure-reference</th><th colspan="5">parameter-part</th><th>value</th></tr>
<tr><td></td><td></td><td></td><td>inout-or-out-parameter$_1$</td><td>…</td><td>inout-or-out-parameter$_n$</td><td></td><td></td></tr>
</table>

Figure 33/Z.143 – Structure d'une réponse à un appel de procédure

Les parties *sender* et *procedure-reference* ont la même signification sur les deux figures. La partie *sender* indique l'expéditeur d'un appel ou de la réponse à un appel de procédure. La partie *procedure-reference* indique la procédure applicable à l'appel et à la réponse. La partie *parameter-part* de l'appel de procédure sur la Figure 32 indique les paramètres `in` et les paramètres `inout` et la partie *parameter-part* de la réponse sur la Figure 33 indique les paramètres `inout` et les paramètres `out` de la procédure applicable à l'appel et à la réponse. En outre, la réponse a une partie *value* pour la valeur de retour incluse dans la réponse à un appel de procédure.

NOTE 1 – Comme indiqué dans la note précédente (voir § 8.4.1), la sémantique opérationnelle présente uniquement un modèle pour les concepts de la notation TTCN-3. La question de savoir si les informations décrites sur les Figures 32 et 33 doivent ou non être envoyées et/ou reçues ainsi que les modalités d'envoi et de réception dépendent de l'implémentation du système de test.

NOTE 2 – Pour un appel de procédure, les paramètres **out** ne sont pas applicables et sont omis sur la Figure 32. Pour une réponse à un appel de procédure, les paramètres **in** ne sont pas applicables et sont omis sur la Figure 33.

NOTE 3 – Les types des paramètres et le type de la valeur de retour peuvent toujours être obtenus unanimement à partir de la définition de la signature associée.

8.4.3 Exceptions

Les exceptions sont également utilisées dans la communication en mode procédure. La structure d'une exception est illustrée sur la Figure 34. Elle comporte quatre parties. La partie *sender* identifie l'expéditeur de l'exception; la partie *procedure-reference* indique la procédure applicable à l'exception, la partie *type* identifie le type de l'exception et la partie *value* donne la valeur de l'exception. La signature de la procédure à laquelle il est fait référence dans la partie *procedure-reference* définit la liste des types d'exception autorisés. Une exception reçue doit être conforme à l'un des types énumérés. En général, il peut s'agir de n'importe quel type de données TTCN-3 prédéfini ou défini par l'utilisateur.

Figure 34/Z.143 – Structure d'une exception

8.4.4 Construction des messages, des appels de procédure, des réponses et des exceptions

Les opérations d'envoi d'un message, d'un appel de procédure, d'une réponse à un appel de procédure et d'une exception sont **send**, **call**, **reply** et **raise**. Toutes ces opérations d'envoi sont construites de la même manière:

```
<port-name>.<sending-operation>(<send-specification>) [to <receiver>]
```

Les parties <port-name> et <sending-operation> définissent le port et l'opération utilisés pour l'envoi d'un item. Dans le cas de connexions de un à plusieurs, il faut spécifier une entité <receiver>. L'item à envoyer est construit au moyen de la spécification <send-specification>, qui peut utiliser des valeurs concrètes, des références de modèle, des valeurs de variable, des constantes, des expressions, des fonctions, etc., pour construire et coder l'item à envoyer.

Dans la sémantique opérationnelle, on suppose qu'il existe une fonction générique *CONSTRUCT-ITEM*:

CONSTRUCT-ITEM (*myEntity,* <sending-operation>*,* <send-specification>)

retourne un message, un appel de procédure, une réponse à un appel de procédure ou une exception, en fonction des parties `<sending-operation>` et `<send-specification>` (les deux parties `<sending-operation>` et `<send-specification>` renvoient aux parties correspondantes de l'opération d'envoi TTCN-3). La référence d'entité *myEntity* correspond à l'expéditeur de l'item à envoyer. On suppose que ces informations *sender* font également partie de l'item à envoyer (Figures 31 à 34).

8.4.5 Appariement des messages, des appels de procédure, des réponses et des exceptions

Les opérations de réception d'un message, d'un appel de procédure, d'une réponse à un appel de procédure et d'une exception sont **receive**, **getcall**, **getreply** et **catch**. Toutes ces opérations de réception sont construites de la même manière:

```
<port-name>.<receiving-operation>(<matching-part>) [from <sender>] [<assignment-part>]
```

Les parties `<port-name>` et `<receiving-operation>` définissent le port et l'opération utilisés pour la réception d'un item. Dans le cas de connexions de un à plusieurs, une clause **from** peut être utilisée pour spécifier l'expéditeur `<sender>`. L'item à recevoir doit remplir les conditions d'appariement spécifiées dans la partie `<matching-part>`. La partie `<matching-part>` peut utiliser des valeurs concrètes, des références de modèle, des valeurs de variable, des constantes, des expressions, des fonctions, etc., pour spécifier les conditions d'appariement.

Dans la sémantique opérationnelle, on suppose qu'il existe une fonction générique *MATCH-ITEM*:

MATCH-ITEM (*myItem,* `<matching-part>`, `<sender>`)

retourne **true** si *myItem* remplit les conditions de `<matching-part>` et si *myItem* a été envoyé par `<sender>`; dans les autres cas, elle retourne **false**.

8.4.6 Extraction d'informations des items reçus

Les informations contenues dans les messages, appels de procédure, réponses aux appels de procédure et exceptions reçus peuvent être extraites dans la partie `<assignment-part>` (voir § 8.4.5) des fonctions de réception **`receive`**, **`getcall`**, **`getreply`** et **`catch`**. La partie `<assignment-part>` décrit comment les paramètres des appels de procédure et des réponses, les valeurs de retour codées dans les réponses, les messages, les exceptions et l'identificateur de l'entité `<sender>` sont affectés aux variables.

Dans la sémantique opérationnelle, on suppose qu'il existe une fonction générique *RETRIEVE-INFO*:

RETRIEVE-INFO (*myItem*, `<assignment-part>`)

> toutes les valeurs devant être extraites conformément à la partie `<assignment-part>` sont extraites et sont affectées aux variables énumérées dans la partie affectation. Les affectations sont réalisées au moyen de l'opération *VAR-SET*, autrement dit les variables associées au même emplacement sont mises à jour simultanément.

8.5 Enregistrements d'appel pour les fonctions, les variantes et les tests élémentaires

Les fonctions, les variantes et les tests élémentaires sont appelés (ou exécutés) par leur nom et une liste de paramètres effectifs. Les paramètres effectifs contiennent des références s'il s'agit de paramètres par référence et des valeurs concrètes s'il s'agit de paramètres par valeur comme défini par les paramètres formels dans la définition de la fonction ou du test élémentaire. La sémantique opérationnelle traite les appels de fonction, de variante et de test élémentaire au moyen d'*enregistrements d'appel* comme indiqué sur la Figure 35. La valeur de *BEHAVIOUR-ID* est le nom d'une fonction ou d'un test élémentaire, les paramètres par valeur contiennent des valeurs concrètes $\langle parId_1 \rangle$ … $\langle parId_n \rangle$ pour les paramètres formels $\langle parId_1 \rangle$ … $\langle parId_n \rangle$. Les paramètres par référence contiennent des références aux emplacements des variables et des temporisations existantes. Avant qu'une fonction ou qu'un test élémentaire puisse être exécuté, un enregistrement d'appel approprié doit être construit.

behaviour-id	value-parameter					reference-parameter				
		$parId_1$	…	$parId_n$			$parId_1$	…	$parId_n$	
		$value_1$	…	$value_n$			loc_1	…	loc_n	

Figure 35/Z.143 – Structure d'un enregistrement d'appel

8.5.1 Traitement des enregistrements d'appel

Le nom de la fonction ou du test élémentaire et les valeurs des paramètres effectifs peuvent être extraits au moyen de la notation à points, par exemple $myCallRecord.parId_n$ ou *myCallRecord.behaviour-id*, où *myCallRecord* est un pointeur vers un enregistrement d'appel.

Pour la construction d'un appel, la fonction *NEW-CALL-RECORD* est supposée être disponible:

NEW-CALL-RECORD(*myBehaviour*)

> crée un nouvel enregistrement d'appel pour la fonction ou le test élémentaire *myBehaviour* et retourne un pointeur vers le nouvel enregistrement. Les champs de paramètre du nouvel enregistrement d'appel ont des valeurs non définies.

myEntity.INIT-CALL-RECORD(*myCallRecord*)

> crée des variables et des temporisations pour le traitement des paramètres par valeur et par référence dans la portée effective du composant de test ou de la commande de module *myEntity*. Les variables pour le traitement des paramètres par valeur sont initialisées avec les valeurs correspondantes fournies dans l'enregistrement d'appel. Les variables et les temporisations pour le traitement des paramètres par référence reçoivent l'emplacement fourni. En outre, elles reçoivent une valeur d'une variable ou d'une temporisation existante dans une autre unité de portée du composant où l'enregistrement d'appel a été créé.

8.6 Procédure d'évaluation d'un module TTCN-3

8.6.1 Phases d'évaluation

La procédure d'évaluation d'un module TTCN-3 comporte les phases suivantes:

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 commande de module se termine. La procédure d'évaluation est décrite au moyen d'un mélange de texte informel, de pseudo-code et des fonctions introduites dans les paragraphes précédents.

8.6.1.1 Phase I: initialisation

La phase d'initialisation inclut les actions suivantes:

a) **Déclaration et initialisation des variables**:

– *INIT-FLOW-GRAPHS*(); // Initialisation du traitement des graphes de flux.
// *INIT-FLOW-GRAPHS* est expliqué au § 8.6.2

– *Entity* := **NULL**; // *Entity* servira à désigner un état d'entité. Un état d'entité
// représente la commande de module ou un composant de test.

NOTE – Les variables globales suivantes *ALL-ENTITY-STATES*, *ALL-PORT-STATES*, *MTC*, *TC-VERDICT* et *DONE* constituent l'état de module qui est manipulé pendant l'interprétation d'un module TTCN-3 (voir § 8.3.1).

– ALL-ENTITY-STATES := **NULL**;
– ALL-PORT-STATES := **NULL**;
– MTC := **NULL**;
– TC-VERDICT := **none**;
– DONE := **NULL**;
– SNAP-DONE := 0;

b) **Création et initialisation de la 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 de flux représentant le comportement de la
// commande du module dont le nom est <moduleId>.
// GET-UNIQUE-ID est expliqué au 8.6.2.

– Entity.INIT-VAR-SCOPE(); // Nouvelle portée de variable
– Entity.INIT-TIMER-SCOPE(); // Nouvelle portée de temporisation
– Entity.VALUE-STACK.push(**MARK**); // Une marque est placée dans la pile de valeurs
– ALL-ENTITY-STATES.append(Entity); // La nouvelle entité est placée dans l'état de module.

8.6.1.2 Phase II: mise à jour

La phase de mise à jour concerne toutes les actions qui sortent du cadre de la sémantique opérationnelle mais qui influent sur l'interprétation d'un module TTCN-3. La phase de mise à jour comporte les actions suivantes:

a) **progression temporelle**: toutes les temporisations en cours sont mises à jour, autrement dit les valeurs *TIME-LEFT* des temporisations en cours sont (éventuellement) diminuées et si, par suite de la mise à jour, une temporisation expire, les liens de temporisation correspondants sont mis à jour, à savoir *TIME-LEFT* est mis à 0.0 et *STATUS* est mis à **`TIMEOUT`**;

NOTE 1 – La mise à jour des temporisations inclut la mise à jour de toutes les temporisations *TIMER-GUARD* en cours dans les états de module. Les temporisations *TIMER-GUARD* servent à protéger l'exécution des tests élémentaires et des opérations d'appel;

b) **comportement du système SUT**: les messages, les appels de procédure distante, les réponses aux appels de procédure distante et les exceptions (éventuellement) reçus en provenance du système SUT sont placés dans les files d'attente des ports auxquels les réceptions correspondantes se font.

NOTE 2 – Dans cette sémantique opérationnelle, aucune hypothèse n'est faite quant à la progression temporelle et au comportement du système SUT.

8.6.1.3 Phase III: sélection

La phase de sélection comporte les deux actions suivantes:

a) **sélection**: sélectionner une entité non bloquée, à savoir une entité dont la valeur de *STATUS* est **ACTIVE** ou **SNAPSHOT**;

b) **stockage**: stocker l'identificateur de l'entité sélectionnée dans la variable globale *Entity*.

8.6.1.4 Phase IV: exécution

La phase d'exécution comporte les deux actions suivantes:

a) **étape d'exécution de l'entité sélectionnée**: exécuter le nœud de graphe de flux se trouvant au sommet de la pile *CONTROL-STACK* de *Entity*;

b) **vérification du critère de terminaison**: arrêter l'exécution si la commande de module est terminée, à savoir si la liste d'états d'entité est vide; dans les autres cas, poursuivre avec la phase II.

NOTE – L'étape d'exécution de l'entité sélectionnée peut être considérée comme un appel de procédure. La vérification du critère de terminaison est faite lorsque l'étape d'exécution est terminée, à savoir lorsque la commande est redonnée.

8.6.2 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 des graphes de flux. Le traitement peut inclure la création des graphes de flux et le traitement des pointeurs vers les graphes de flux et vers leurs nœuds.

b) *GET-UNIQUE-ID* est supposée être une fonction qui retourne 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 augmentée et retournée chaque fois que *GET-UNIQUE-ID* est appelée.

Le pseudo-code utilisé dans les paragraphes qui suivent pour décrire l'exécution de nœuds de graphe de flux utilise les fonctions *CONTINUE-COMPONENT*, *RETURN*, `***DYNAMIC-ERROR***`:

a) *CONTINUE-COMPONENT*: le composant de test effectif continue son exécution avec le nœud se trouvant au sommet de la pile de commande, autrement dit la commande n'est pas redonnée à la procédure d'évaluation de module décrite dans le présent paragraphe.

b) *RETURN*: redonne la commande à la procédure d'évaluation de module décrite dans le présent paragraphe. La fonction *RETURN* est la dernière action de l'*étape d'exécution de l'entité sélectionnée* de la phase d'*exécution*.

c) `***DYNAMIC-ERROR***` indique l'occurrence d'une erreur dynamique. La procédure de traitement des erreurs proprement dite sort du cadre de la sémantique opérationnelle. Si une erreur dynamique se produit, l'ensemble du comportement qui suit du test élémentaire est censé être non défini. Dans ce cas, les ressources attribuées au test élémentaire doivent être supprimées et le verdict **error** est affecté au test élémentaire. La commande est donnée à l'instruction de la partie commande qui suit l'instruction d'exécution dans laquelle l'erreur s'est produite. Cela est modélisé par le segment de graphe de flux <dynamic-error> (voir § 9.18b).

   NOTE – L'occurrence d'une erreur dynamique concerne le comportement de test. Une erreur dynamique telle que spécifiée par la sémantique opérationnelle indique un problème dans l'utilisation de la notation TTCN-3, par exemple une mauvaise utilisation ou une situation de concurrence.

d) *`APPLY-OPERATOR`* est utilisée en tant que fonction générique pour décrire l'évaluation d'opérateurs (par exemple +, *, / ou –) dans des expressions (voir § 9.18.4).

9 Segments de graphe de flux pour les constructions TTCN-3

La sémantique opérationnelle représente le comportement TTCN-3 sous forme de graphes de flux. L'algorithme de construction des graphes de flux représentant le comportement est décrit au § 8.2. Il est fondé sur des modèles de graphes de flux et de segments de graphes de flux à utiliser pour la construction de graphes de flux concrets pour la commande de module, les tests élémentaires, les variantes, les fonctions et les types de composant définis dans un module TTCN-3. Le présent paragraphe définit les modèles des segments de graphe de flux. Ces modèles sont présentés par ordre alphabétique et non dans un ordre logique.

Les segments de graphe de flux sont définis dans des figures. Les nœuds de graphe de flux sont présentés dans la partie gauche des figures et les commentaires associés aux nœuds et aux arcs sont indiqués dans la partie droite. Les figures contiennent des commentaires descriptifs pour les nœuds de référence et des commentaires en pseudo-code pour les nœuds de base. Le pseudo-code décrit comment un nœud de base est interprété, à savoir comment il modifie l'état de

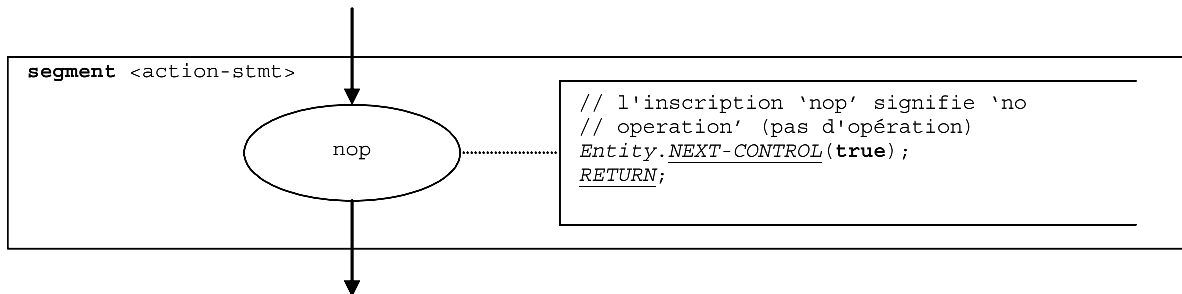
module. Il utilise les fonctions définies au paragraphe 8 et les variables globales déclarées et initialisées dans la procédure d'évaluation des modules TTCN-3 (voir § 8.6). Le paragraphe 8 contient une vue d'ensemble de la totalité des fonctions et des mots clés utilisés par le pseudo-code.

9.1 Instruction action

La structure syntaxique d'une instruction **`action`** est la suivante:

```
action (<informal description>)
```

Le segment de graphe de flux <action-stmt> sur la Figure 36 définit l'exécution de l'instruction **`action`**.



NOTE – Le paramètre <informal description> de l'instruction **`action`** n'a pas de signification pour la sémantique opérationnelle et n'est donc pas représenté dans le segment de graphe de flux.

Figure 36/Z.143 – Segment de graphe de flux <action-stmt>

9.2 Instruction activate

La structure syntaxique de l'instruction **`activate`** est la suivante:

```
activate(<altstep-name>([<act-par-desc1>, ... , <act-par-descn>]))
```

Le nom <altstep-name> est le nom d'une variante qui est activée en tant que comportement par défaut et les descriptions `<act-par-descr`$_1$`>`, … , `<act-par-descr`$_n$`>` décrivent les valeurs des paramètres effectifs de la variante au moment de son activation.

On suppose que pour chaque description `<act-par-desc`$_1$`>`, l'identificateur de paramètre formel correspondant `<f-par-Id`$_1$`>` est connu, autrement dit il est possible d'étendre la structure syntaxique ci-dessus comme suit:

```
activate(<altstep-name>((<f-par-Id1>,<act-par-desc1>), ... , (<f-par-Idn>,<act-par-descn>)))
```

Le segment de graphe de flux `<activate-stmt>` sur la Figure 37 définit l'exécution de l'instruction **`activate`**. L'exécution est structurée en trois étapes. Dans la première étape, un enregistrement d'appel pour la variante `<function-name>` est créé. Dans la deuxième étape, les valeurs des paramètres effectifs sont calculées et affectées au champ correspondant dans l'enregistrement d'appel. Dans la troisième étape, l'enregistrement d'appel est placé comme premier élément dans la liste *DEFAULT-LIST* de l'entité qui active le comportement par défaut.

NOTE – Pour les variantes qui sont activées en tant que comportement par défaut, seuls les paramètres par valeur sont autorisés. Sur la Figure 37, le traitement des paramètres par valeur est décrit par le segment de graphe de flux <value-par-calculation>, qui est défini au § 9.24.1.

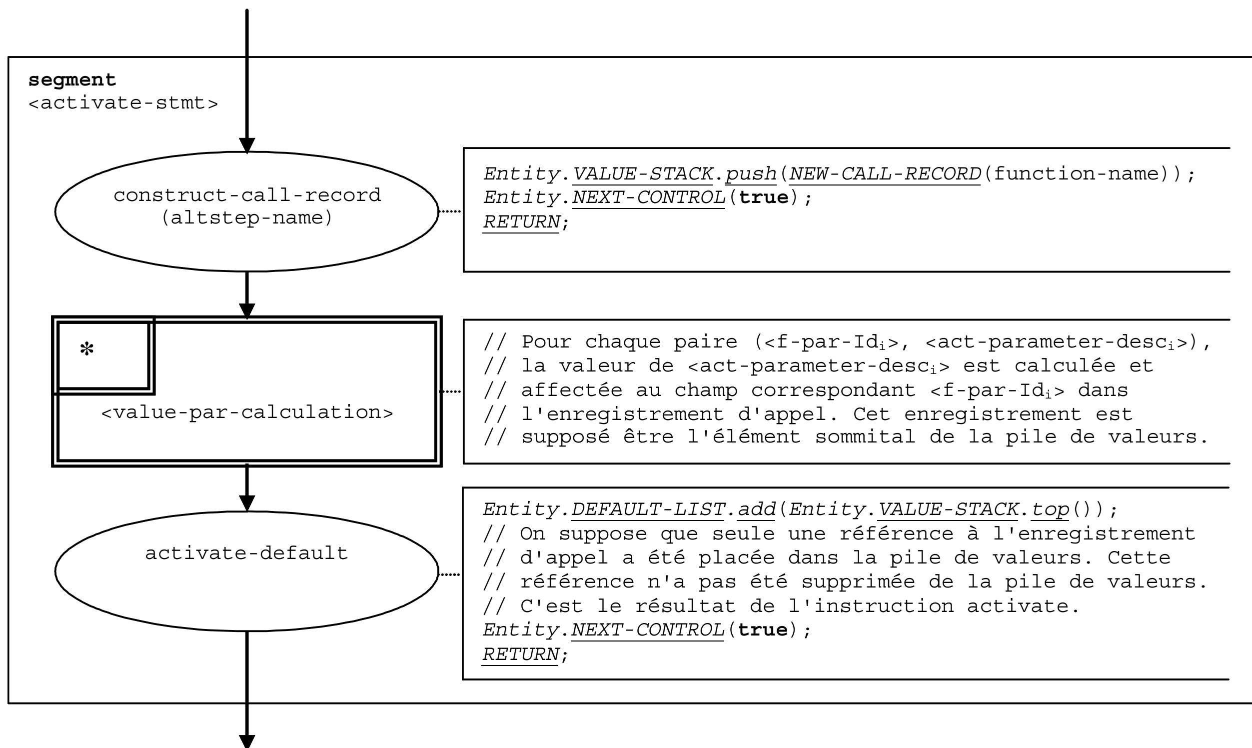


Figure 37/Z.143 – Segment de graphe de flux <activate-stmt>

9.3 Instruction alt

L'instruction **`alt`** est l'instruction la plus compliquée et la plus importante de la notation TTCN-3. Elle implémente la sémantique d'instantané et spécifie la succession des branches résultant de la réception de messages, de réponses, d'appels et d'exceptions, résultant de l'occurrence d'expirations de temporisation et résultant de la terminaison de composants. En outre, l'évocation du mécanisme par défaut TTCN-3 est également liée à l'instruction **`alt`**.

La représentation de l'instruction **`alt`** sous forme de graphe de flux est illustrée sur la Figure 38. Les différentes alternatives résultant de la réception de messages, de réponses, d'appels et d'exceptions, résultant de l'occurrence d'expirations de temporisation et résultant de la terminaison de composants sont cachées dans le segment de graphe de flux `<receiving-branch>`.

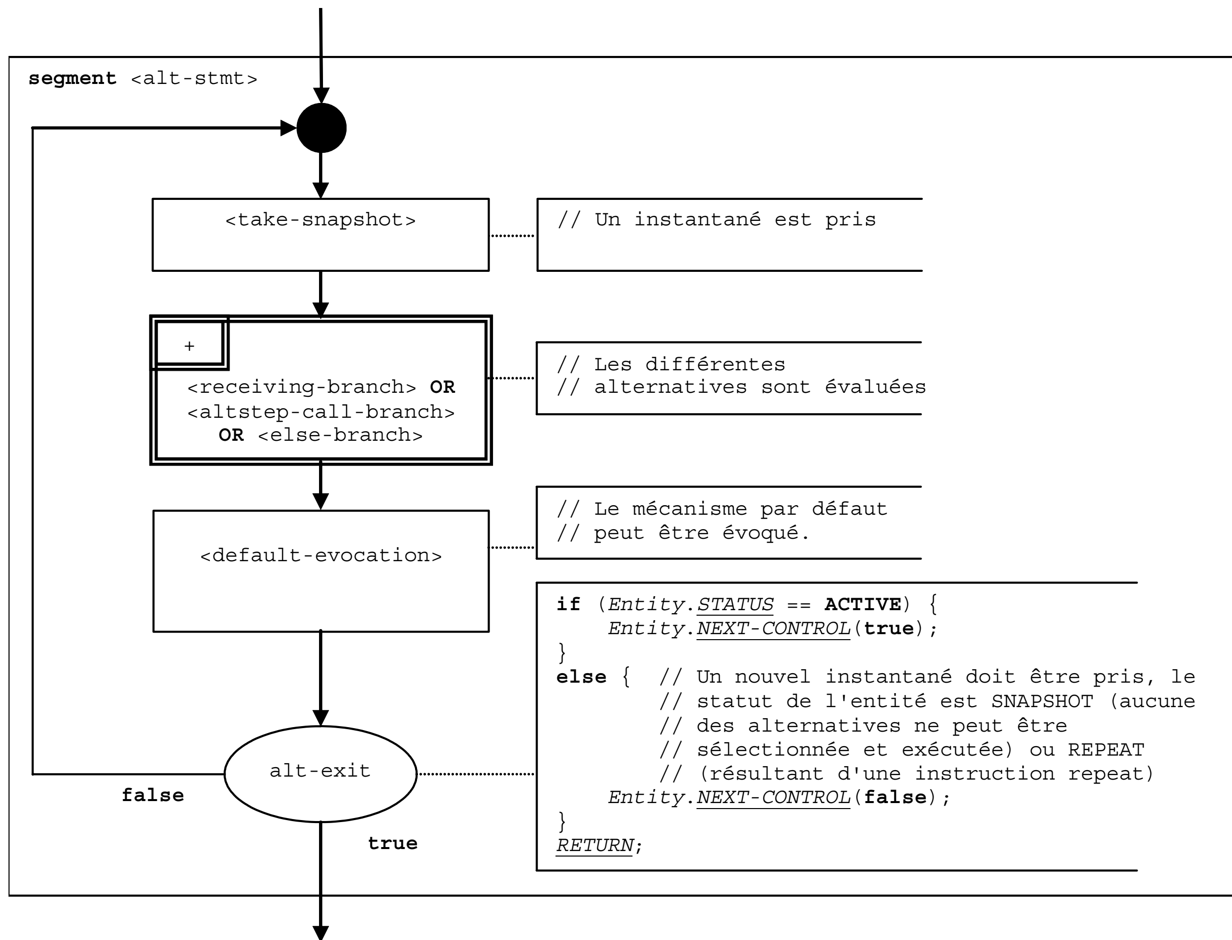


Figure 38/Z.143 – Segment de graphe de flux <alt-stmt>

9.3.1 Segment de graphe de flux <take-snapshot>

Le segment de graphe de flux `<take-snapshot>` sur la Figure 39 décrit la procédure de prise d'un instantané. L'instantané enregistre les valeurs des ports, des temporisations et des composants arrêtés.

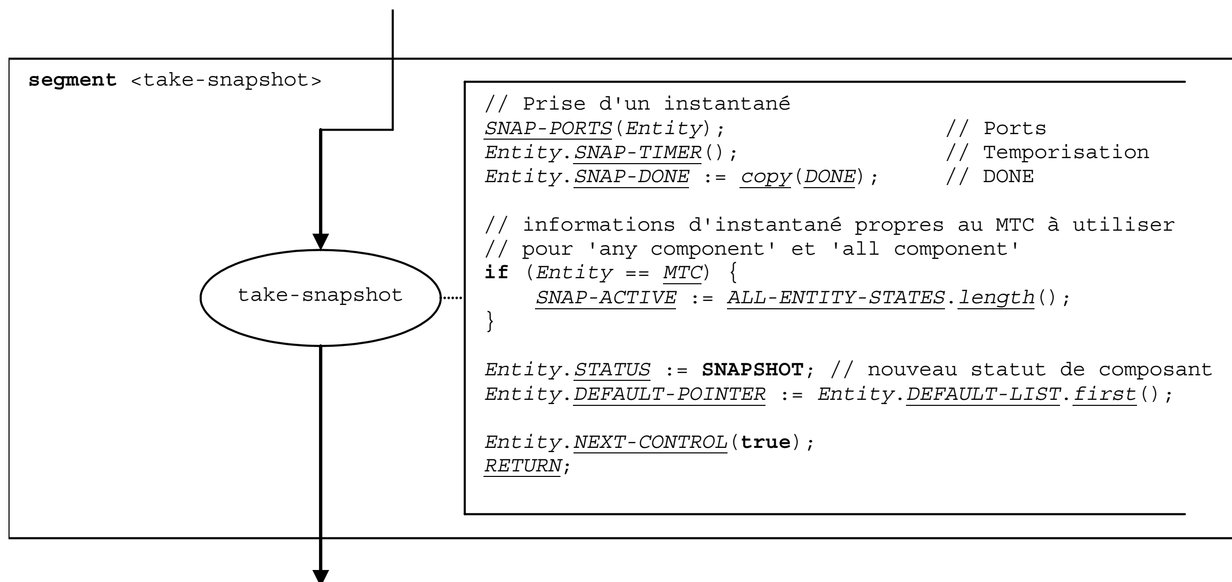


Figure 39/Z.143 – Segment de graphe de flux <take-snapshot>

9.3.2 Segment de graphe de flux <receiving-branch>

L'exécution du segment de graphe de flux `<receiving-branch>` est illustrée sur la Figure 40.

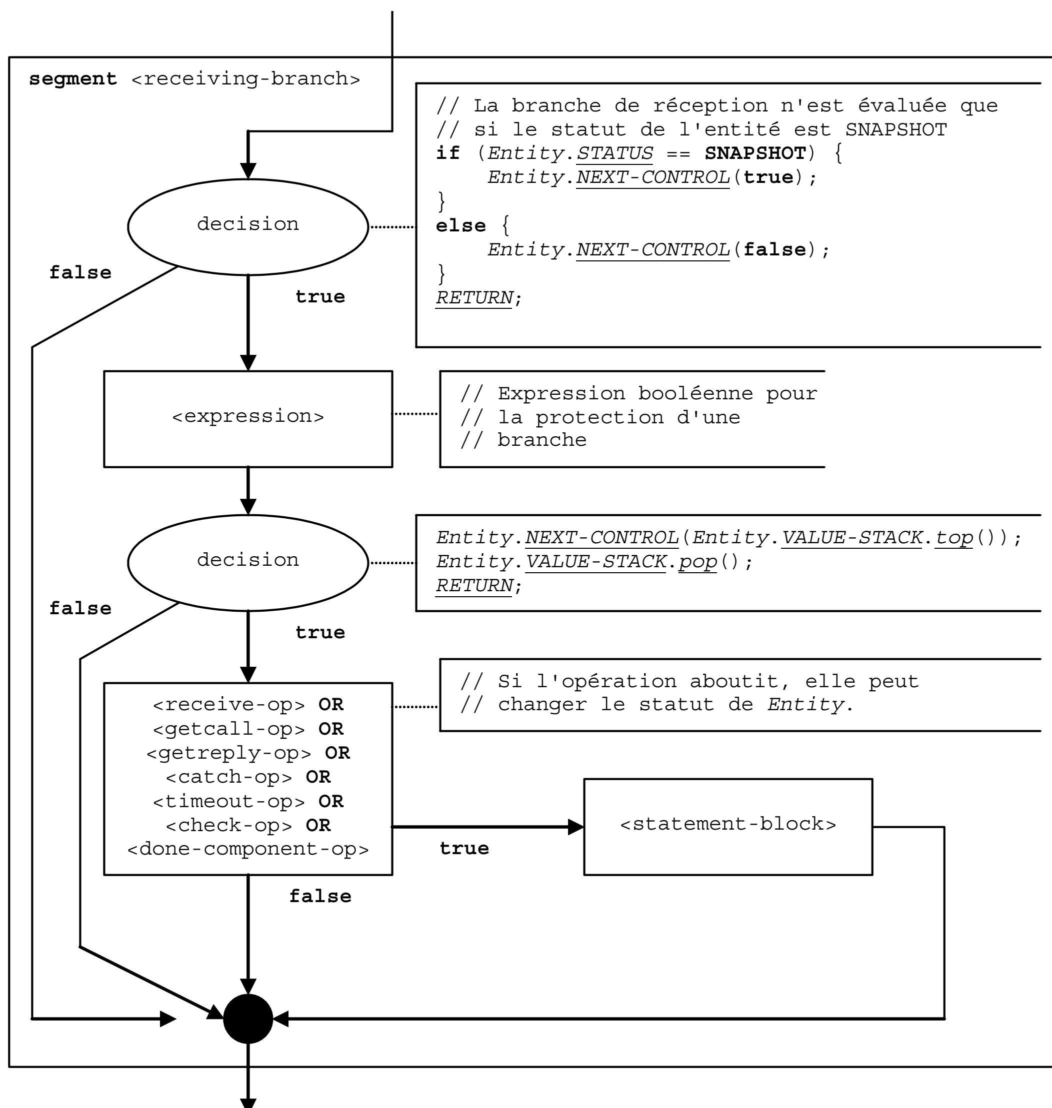


Figure 40/Z.143 – Segment de graphe de flux <receiving-branch>

9.3.3 Segment de graphe de flux <altstep-call-branch>

L'invocation d'une variante dans une instruction **`alt`** est décrite par le segment de graphe de flux`<altstep-call-branch>` sur la Figure 41.

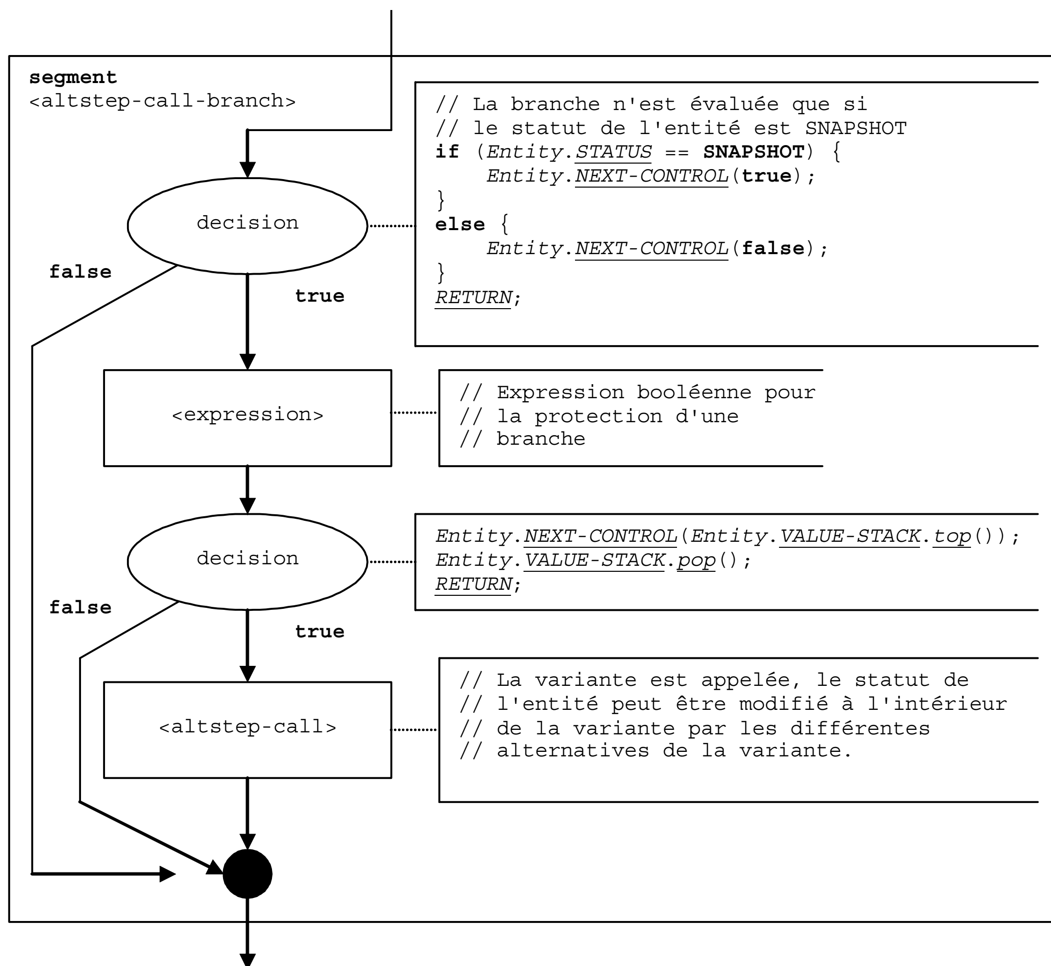


Figure 41/Z.143 – Segment de graphe de flux <altstep-call-branch>

9.3.4 Segment de graphe de flux <else-branch>

L'exécution d'une branche **else** dans une instruction **alt** est décrite par le segment de graphe de flux <else-branch> sur la Figure 42.

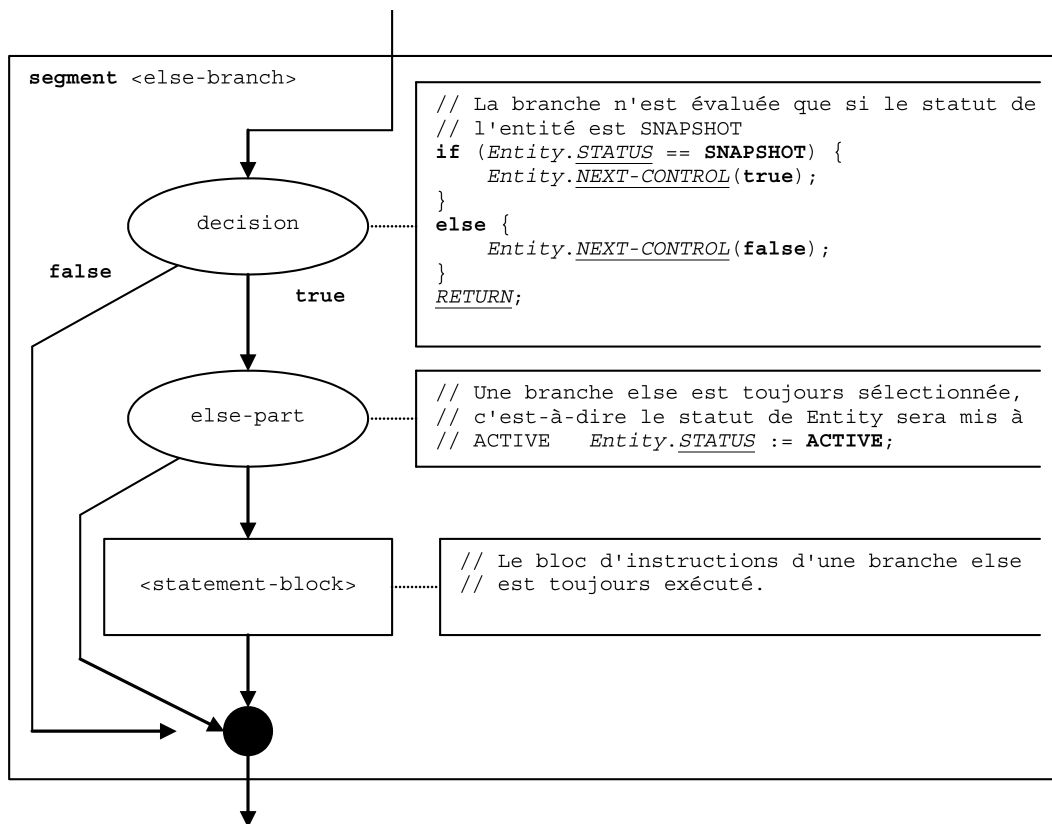


Figure 42/Z.143 – Segment de graphe de flux <else-branch>

9.3.5 Segment de graphe de flux <default-evocation>

L'évocation du comportement par défaut à la fin d'instructions **`alt`** est décrite par le segment de graphe de flux `<default-evocation>` sur la Figure 43.

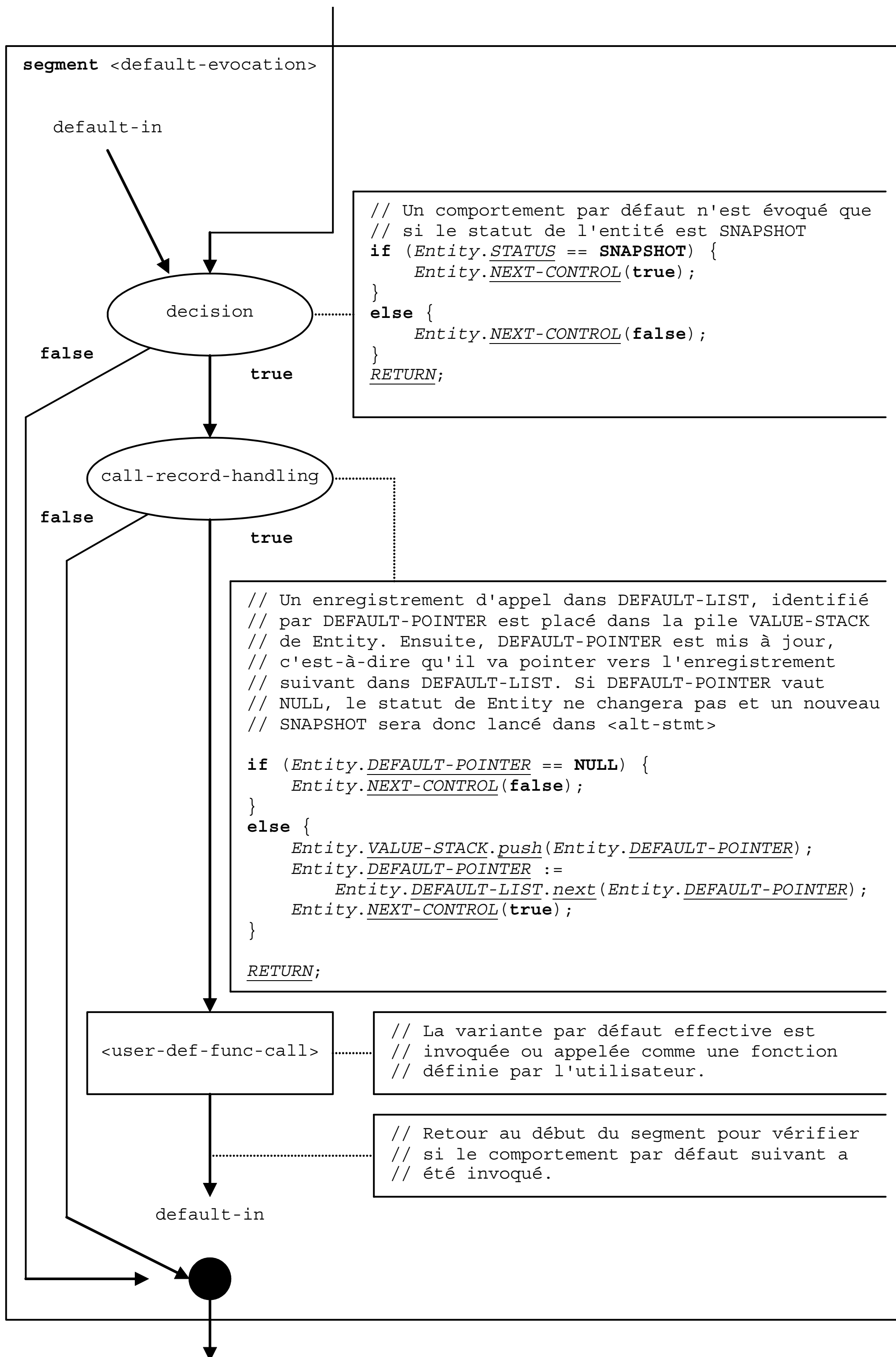


Figure 43/Z.143 – Segment de graphe de flux <default-evocation>

9.4 Appel de variante

Comme indiqué sur la Figure 44, l'appel d'une variante est traité comme un appel de fonction.

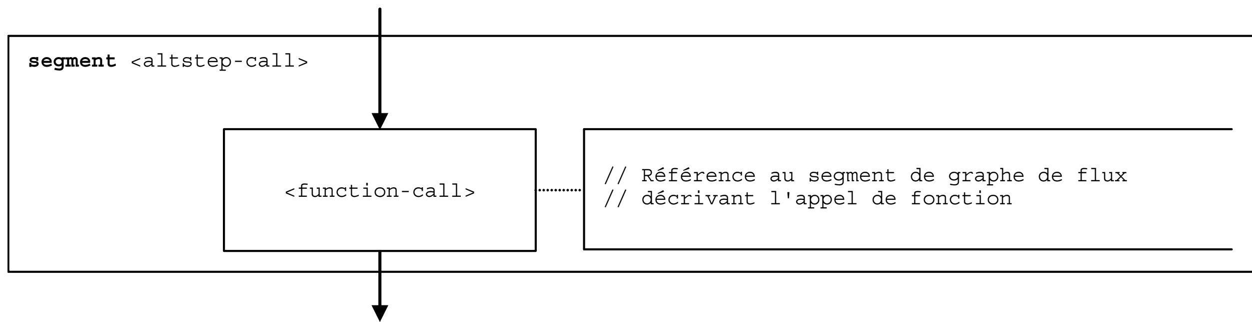


Figure 44/Z.143 – Segment de graphe de flux <altstep-call>

9.5 Instruction assignment

La structure syntaxique d'une instruction **assignment** est la suivante:

```
<varId> := <expression>
```

La valeur de l'expression <expression> est affectée à la variable <varId>. L'exécution d'une instruction **assignment** est définie par le segment de graphe de flux <assignment-stmt> sur la Figure 45.

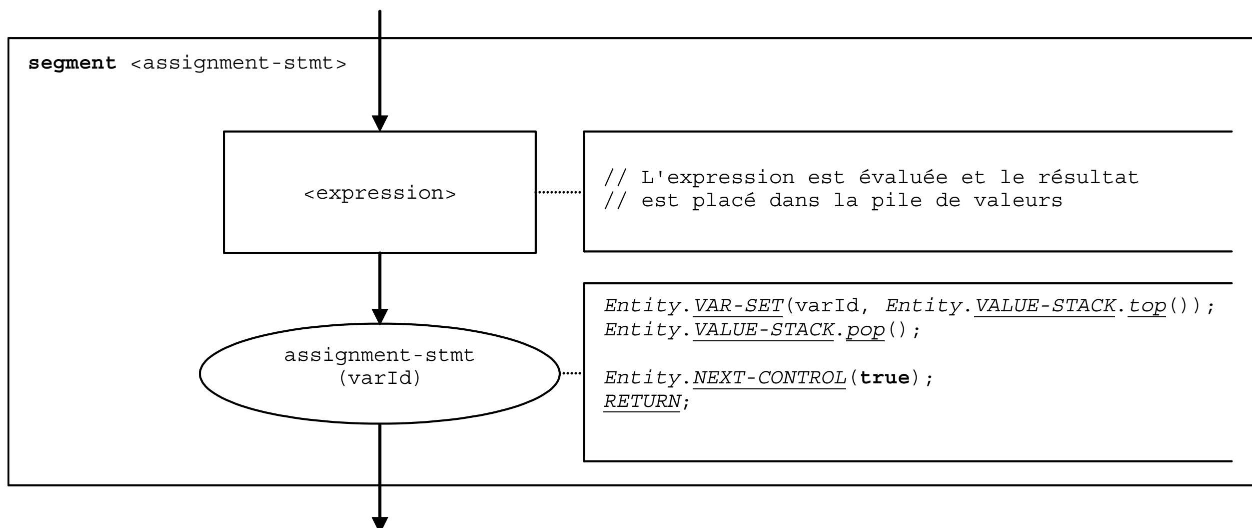


Figure 45/Z.143 – Segment de graphe de flux <assignment-stmt>

9.6 Opération call

La structure syntaxique de l'opération **call** est la suivante:

```
<portId>.call (<callSpec> [<blocking-info>]) [to <component-expression>] [<call-reception-part>]
```

La partie facultative <blocking-info> comporte soit le mot clé **nowait** soit une durée pour une exception d'expiration de temporisation. La partie facultative <component-expression> dans la clause **to** indique l'entité réceptrice. Il peut s'agir d'une valeur de variable ou de la valeur de retour d'une fonction. La partie facultative <call-reception-part> indique les différentes réceptions possibles en cas d'opération **call** bloquante.

La sémantique opérationnelle fait la distinction entre opération **call** *bloquante* et opération **call** *non bloquante*. Une opération **call** est non bloquante si le mot clé **nowait** est utilisé dans l'opération **call**, ou si la procédure appelée est non bloquante, c'est-à-dire si elle est définie au moyen du mot clé **noblock**. Une opération **call** bloquante a une partie <call-reception-part>.

Le segment de graphe de flux <call-op> sur la Figure 46 définit l'exécution d'une opération **call**. Il reflète la distinction entre opération d'appel bloquante et opération d'appel non bloquante.

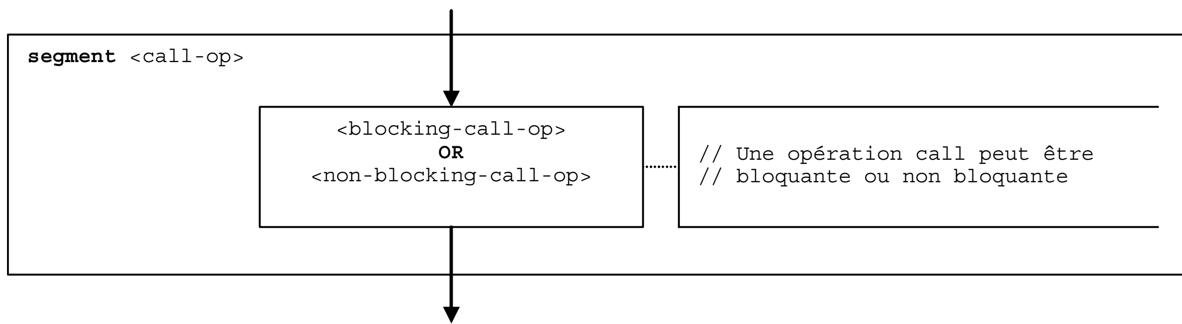


Figure 46/Z.143 – Segment de graphe de flux <call-op>

Pour les opérations d'appel bloquantes comme pour les opérations d'appel non bloquantes, une entité réceptrice peut être spécifiée sous la forme d'une expression. Les possibilités sont montrées sur les Figures 47 et 48.

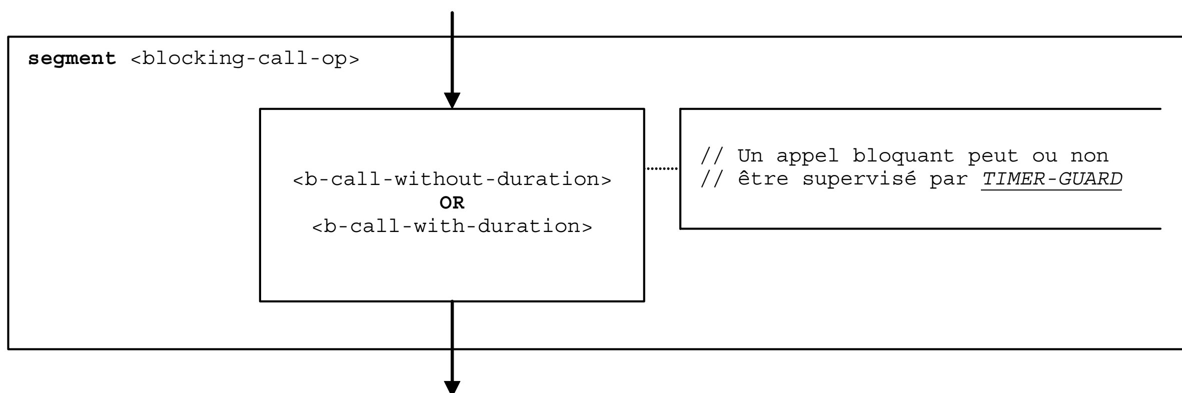


Figure 47/Z.143 – Segment de graphe de flux <blocking-call-op>

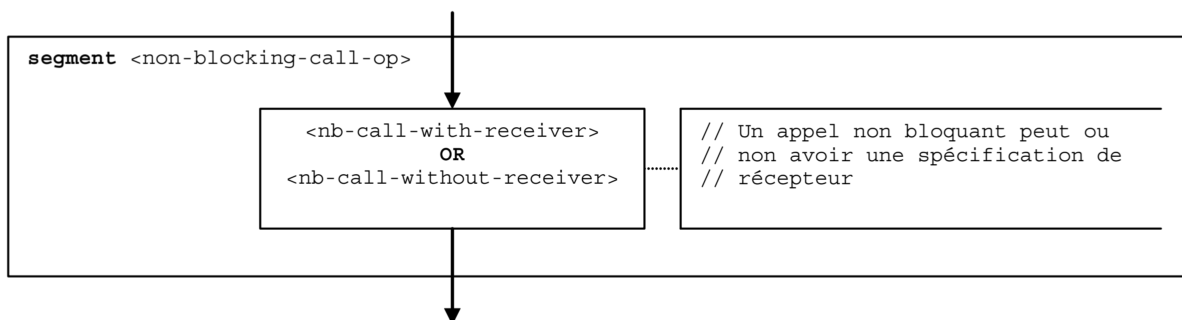


Figure 48/Z.143 – Segment de graphe de flux <non-blocking-call-op>

9.6.1 Segment de graphe de flux <nb-call-with-receiver>

Le segment de graphe de flux <nb-call-with-receiver> sur la Figure 49 définit l'exécution d'une opération **call** non bloquante pour laquelle le récepteur est spécifié sous la forme d'une expression.

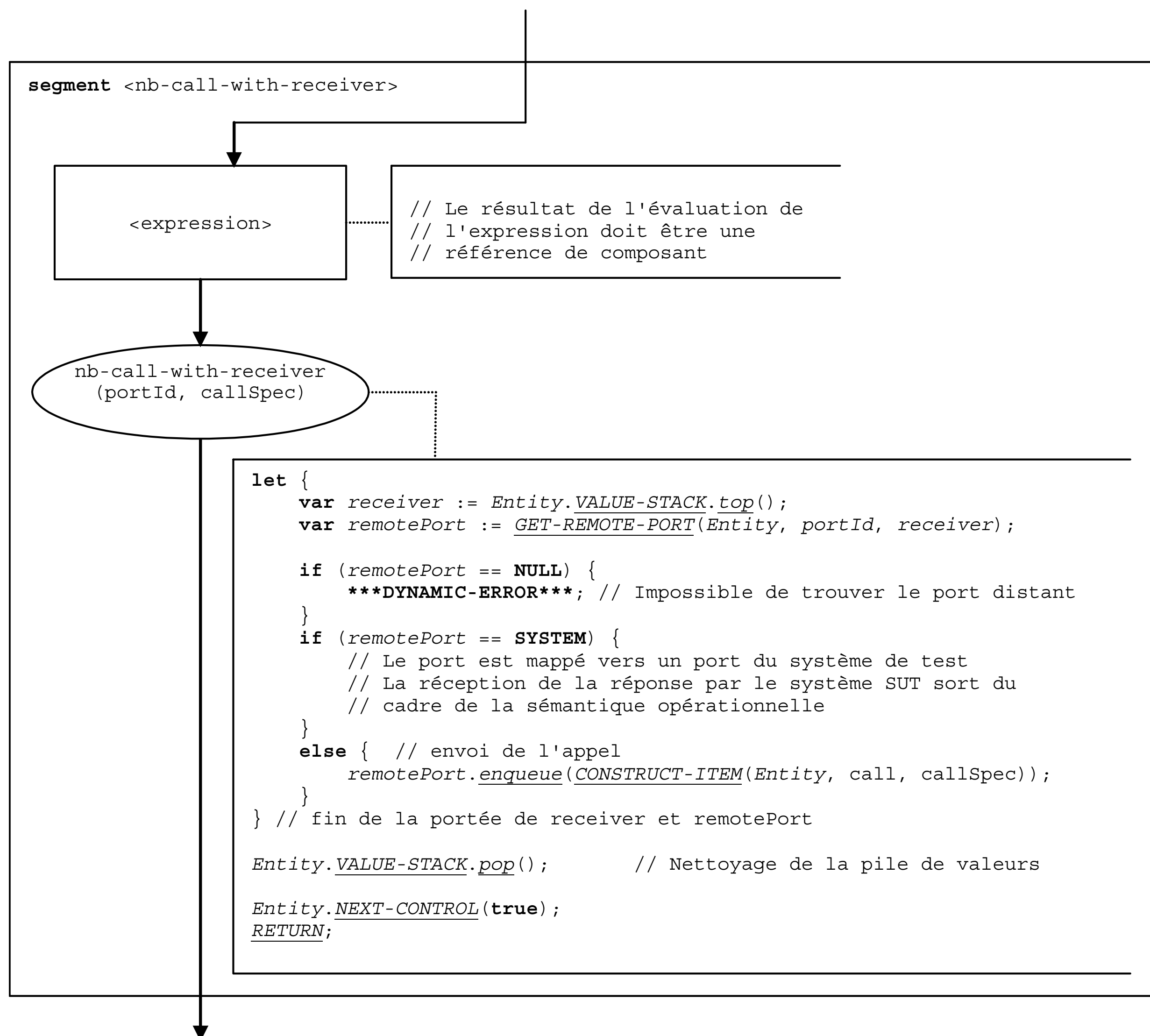


Figure 49/Z.143 – Segment de graphe de flux <nb-call-with-receiver>

9.6.2 Segment de graphe de flux <nb-call-without-receiver>

Le segment de graphe de flux `<nb-call-without-receiver>` sur la Figure 50 définit l'exécution d'une opération **`call`** non bloquante sans clause **`to`**.

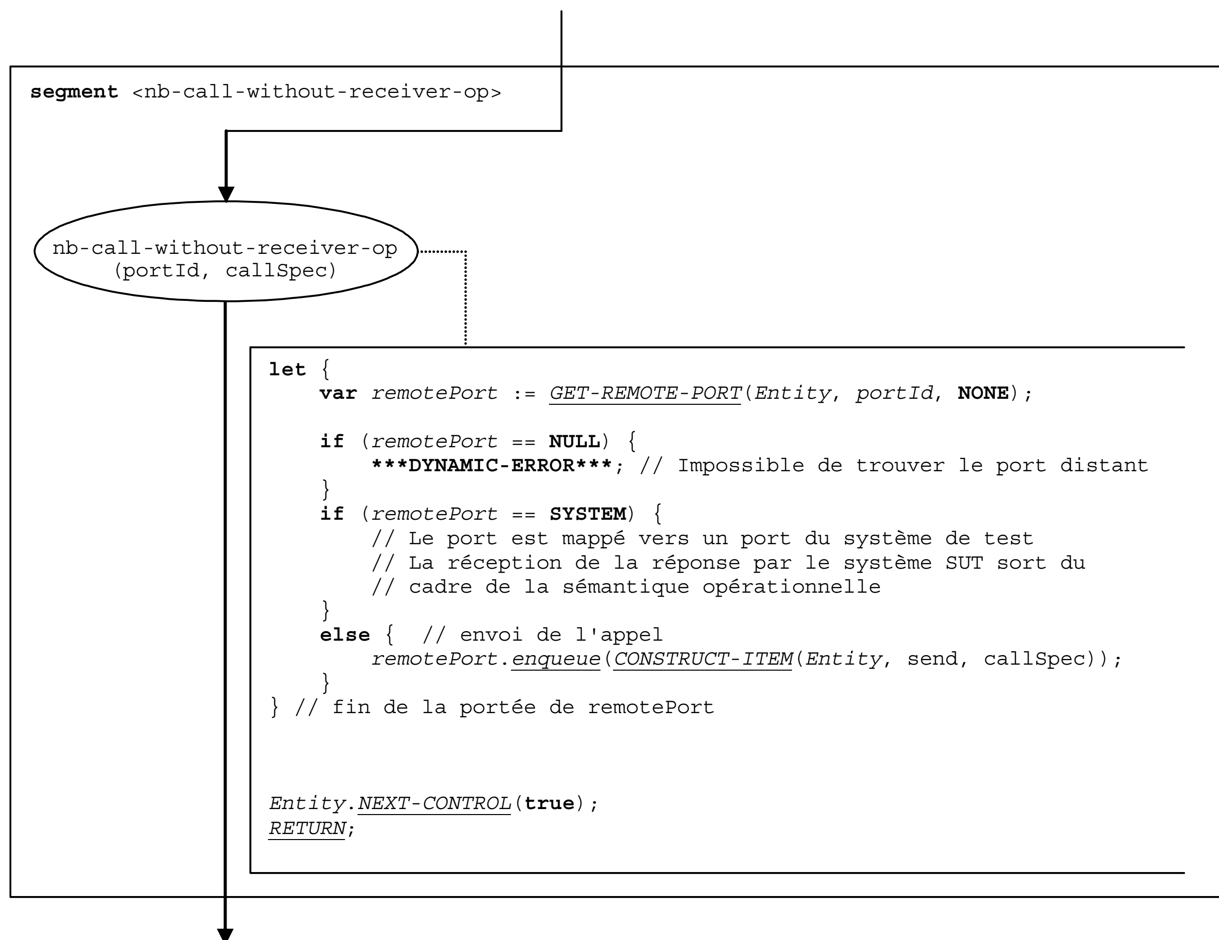


Figure 50/Z.143 – Segment de graphe de flux <nb-call-without-receiver>

9.6.3 Segment de graphe de flux <b-call-without-duration>

Les appels bloquants sont modélisés par un appel non bloquant suivi par le corps de l'appel, qui traite les réponses et les exceptions. Le segment de graphe de flux `<b-call-without-duration>` illustré sur la Figure 51 décrit l'exécution d'un appel bloquant pour laquelle aucune durée n'est donnée pour la temporisation de protection.

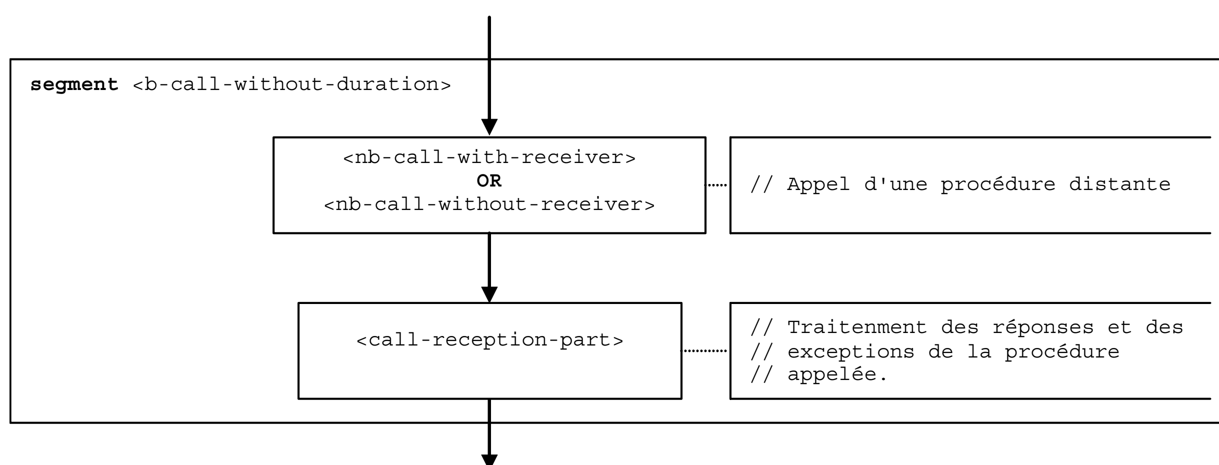


Figure 51/Z.143 – Segment de graphe de flux <b-call-without-duration>

9.6.4 Segment de graphe de flux <b-call-with-duration>

Le segment de graphe de flux `<b-call-with-duration>` (voir la Figure 52) décrit l'exécution d'un appel bloquant avec une durée de temporisation de protection.

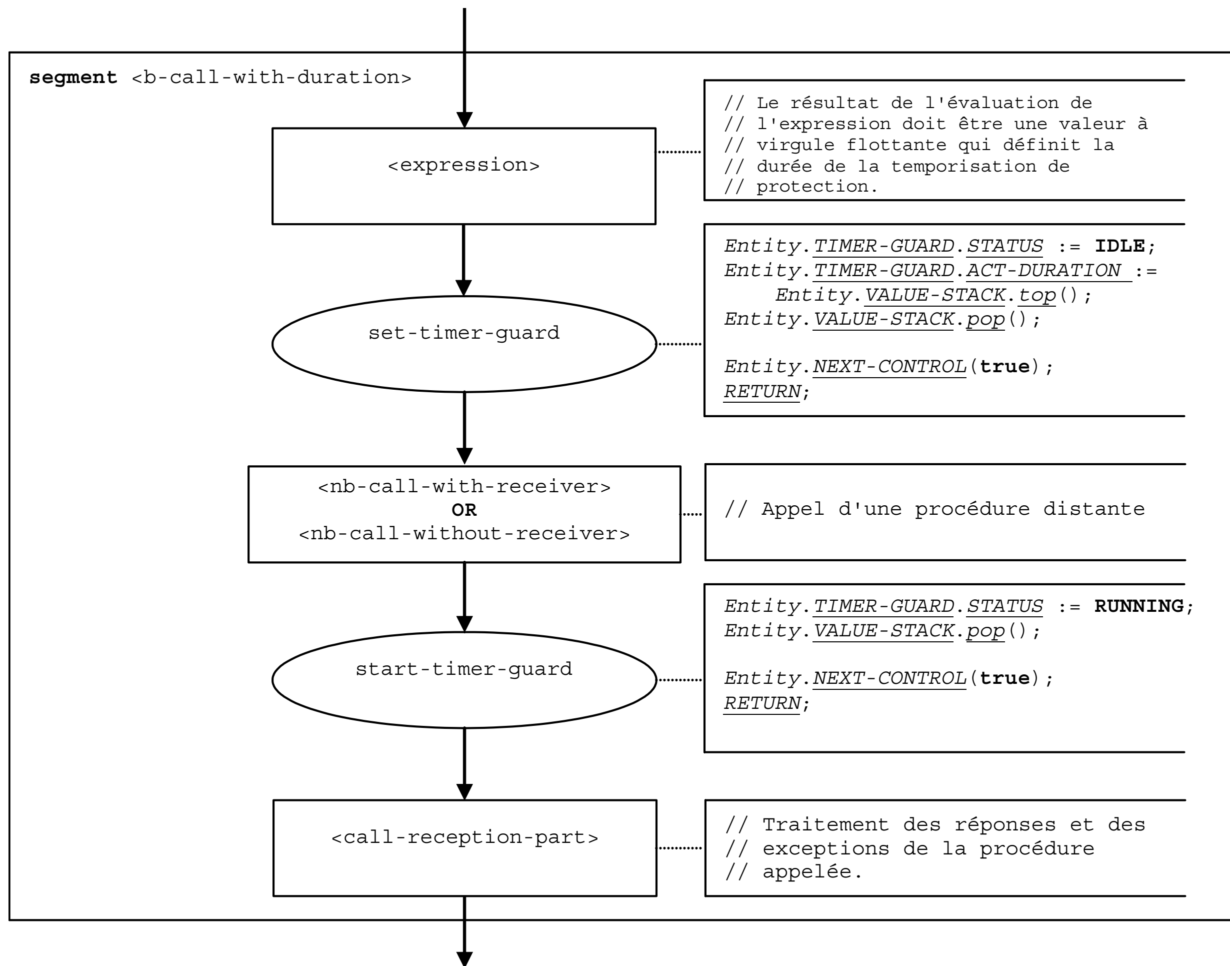


Figure 52/Z.143 – Segment de graphe de flux <b-call-with-duration>

9.6.5 Segment de graphe de flux <call-reception-part>

Le segment de graphe de flux `<call-reception-part>` (voir la Figure 53) décrit le traitement des réponses, des exceptions et de l'exception d'expiration de temporisation d'une opération **`call`** bloquante.

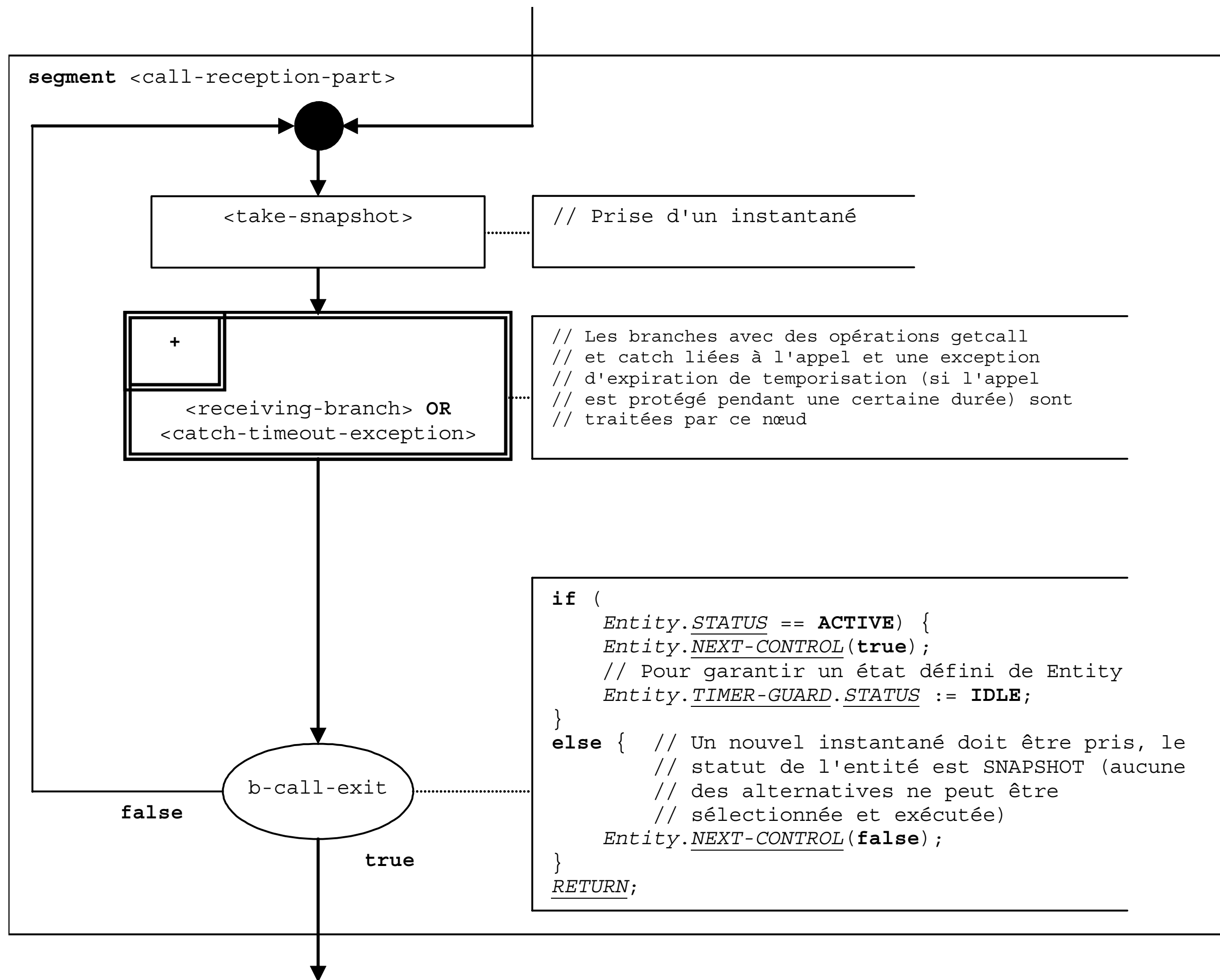


Figure 53/Z.143 – Segment de graphe de flux <call-reception-part>

9.6.6 Segment de graphe de flux <catch-timeout-exception>

Le segment de graphe de flux `<catch-timeout-exception>` (voir la Figure 54) concerne le traitement d'une exception d'expiration de temporisation pour une opération d'appel bloquante qui est protégée par une durée.

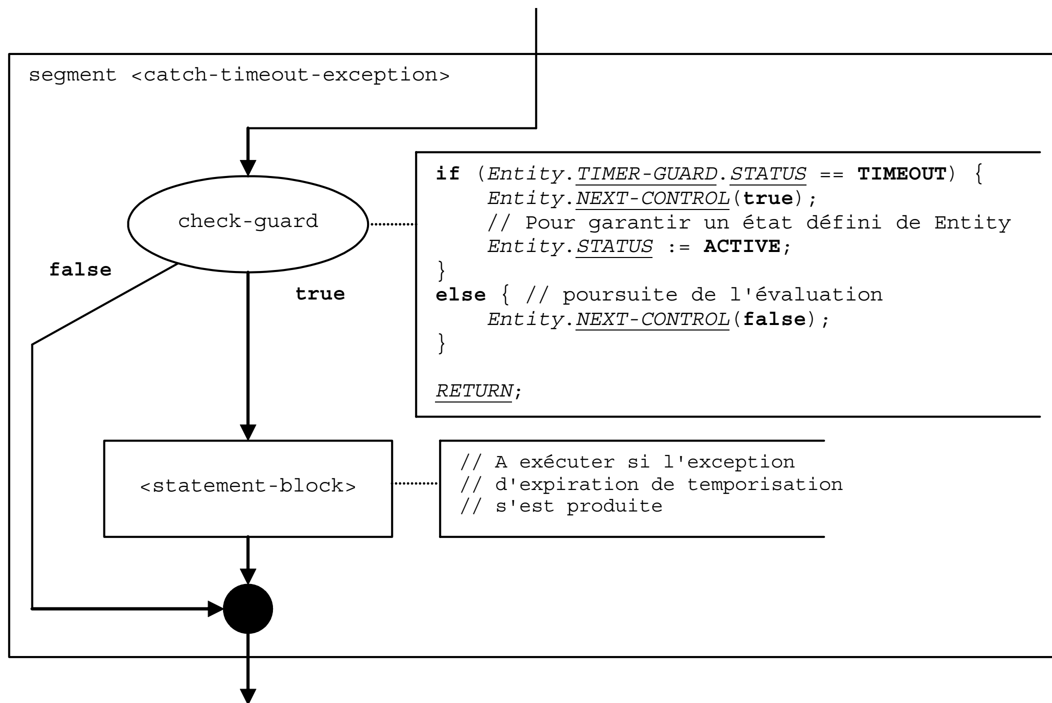


Figure 54/Z.143 – Segment de graphe de flux <catch-timeout-exception>

9.7 Opération catch

La structure syntaxique de l'opération **catch** est la suivante:

```
<portId>.catch (<matchingSpec>) [from <component_expression>] -> [<assignmentPart>]
```

Mis à part le mot clé **`catch`**, cette structure syntaxique est identique à la structure syntaxique de l'opération **`receive`**. Par conséquent, la sémantique opérationnelle traite l'opération **`catch`** de la même manière que l'opération **`receive`**, ce qui est illustré dans le segment de graphe de flux `<catch-op>` (Figure 55), qui définit l'exécution d'une opération **`catch`**. La figure fait référence aux segments de graphe de flux liés à l'opération **`receive`** (voir § 9.37).

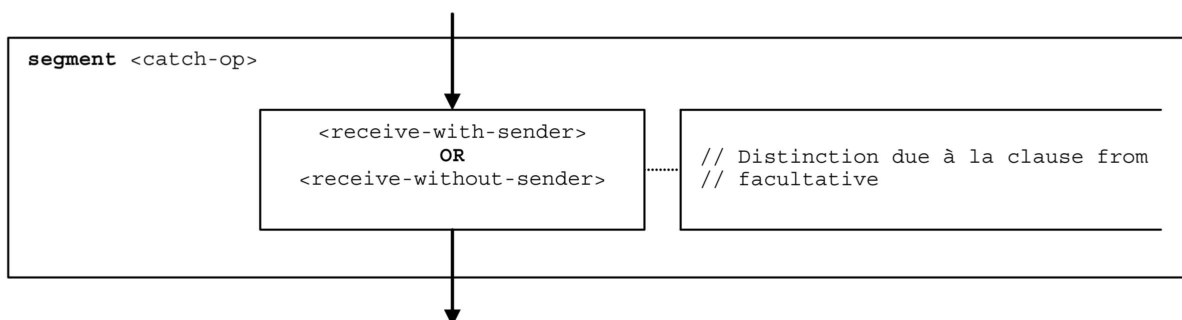


Figure 55/Z.143 – Segment de graphe de flux <catch-op>

9.8 Opération check

La structure syntaxique de l'opération **`check`** est la suivante:

```
<portId>.check( receive|getcall|catch|getreply (<matchingSpec>)
          [from <component-expression>]) [-> <assignmentPart>]
```

La partie facultative `<component-expression>` dans la clause **`from`** indique l'entité expéditrice. Il peut s'agir d'une valeur de variable ou de la valeur de retour d'une fonction, autrement dit on suppose qu'il s'agit d'une expression. La partie facultative `<assignmentPart>` indique l'affectation des informations reçues si celles-ci correspondent à la spécification d'appariement `<matchingSpec>` et à la clause (facultative) **`from`**.

La sémantique opérationnelle traite les opérations **receive**, **getcall**, **catch** et **getreply** de la même manière, c'est-à-dire qu'elles sont décrites en faisant référence aux mêmes segments de graphe de flux <receive-with-sender> et <receive-without-sender>. L'opération **check** traite aussi les différentes opérations de la même manière. Par conséquent, le segment de graphe de flux <check-op> sur la Figure 56, qui définit l'exécution de l'opération **check**, ne fait également référence qu'à ces deux segments de graphe de flux. La seule différence en ce qui concerne les segments de graphe de flux <receive-with-sender> et <receive-without-sender> est que les items reçus ne sont pas supprimés après l'appariement.

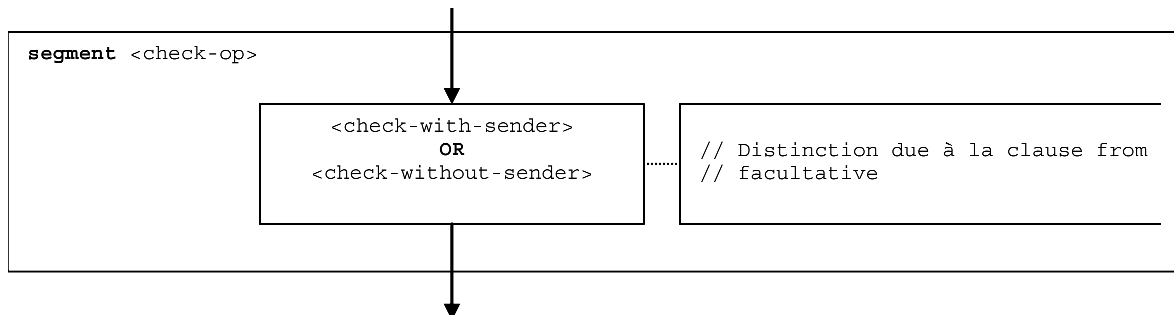


Figure 56/Z.143 – Segment de graphe de flux <check-op>

9.8.1 Segment de graphe de flux <check-with-sender>

Le segment de graphe de flux <check-with-sender> sur la Figure 57 définit l'exécution d'une opération **check** pour laquelle l'expéditeur est spécifié sous la forme d'une expression.

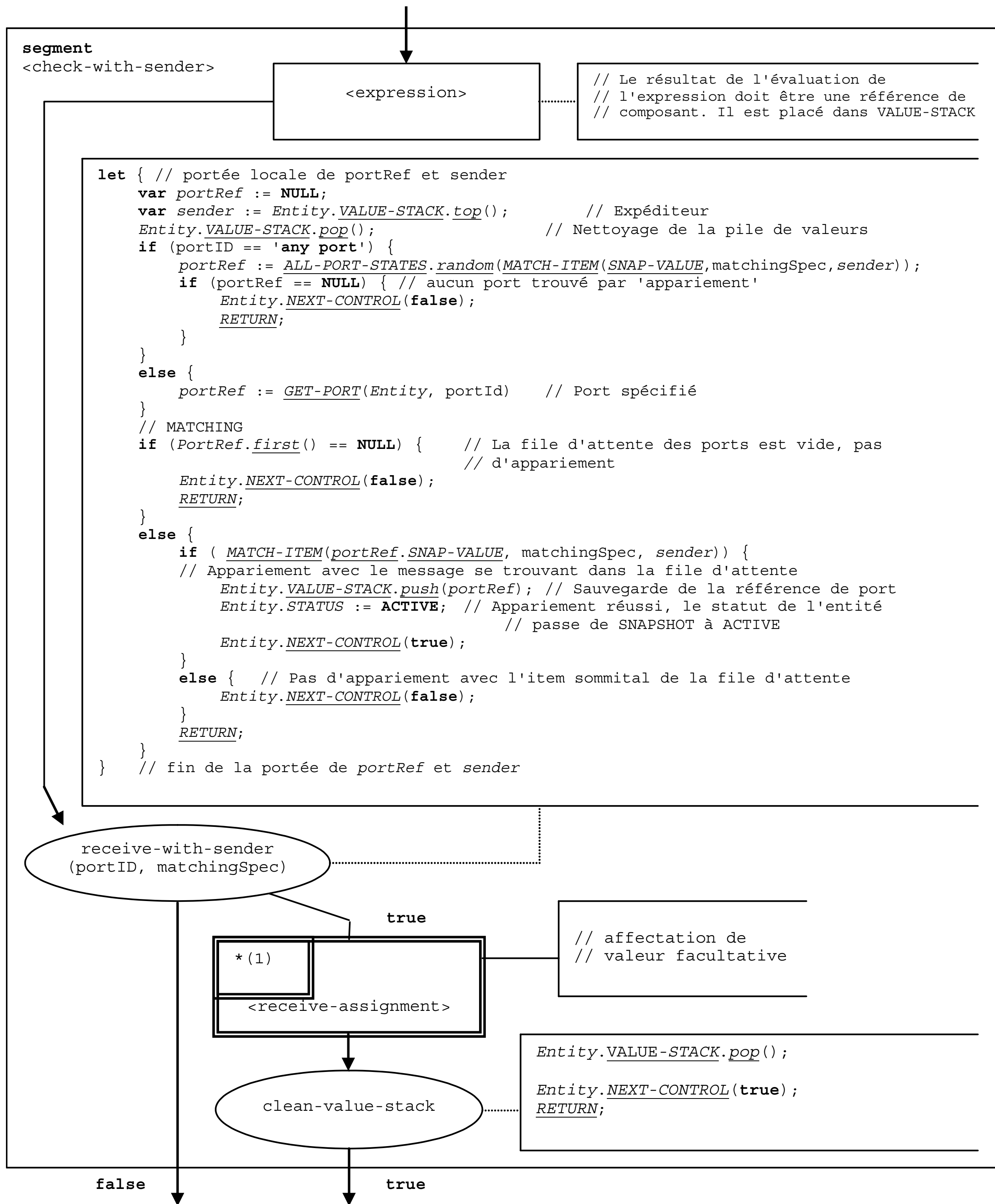


Figure 57/Z.143 – Segment de graphe de flux <check-with-sender>

9.8.2 Segment de graphe de flux <check-without-sender>

Le segment de graphe de flux `<check-without-sender>` sur la Figure 58 définit l'exécution d'une opération **`check`** sans clause **`from`**.

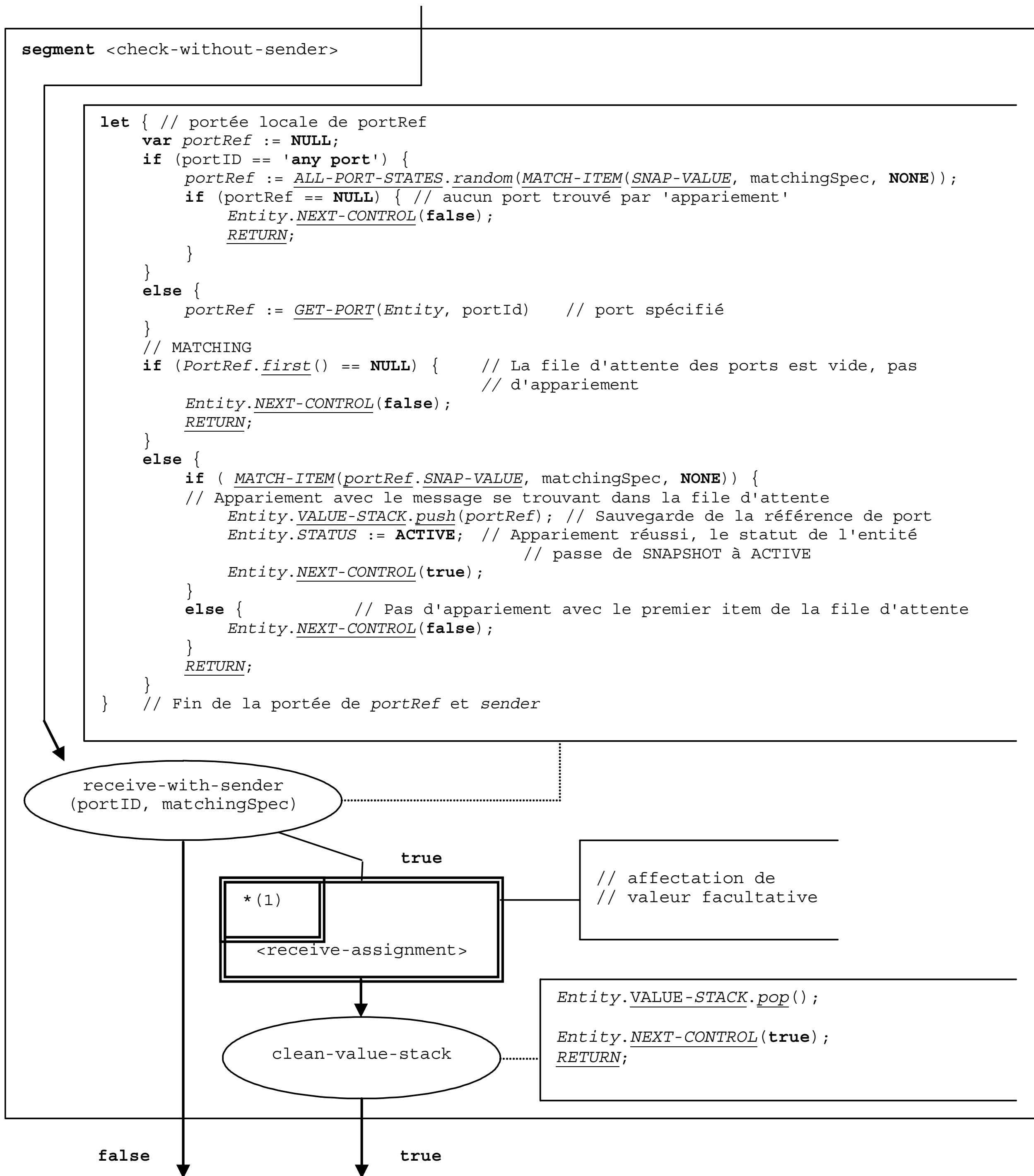


Figure 58/Z.143 – Segment de graphe de flux <check-without-sender>

9.9 Opération clear applicable aux ports

La structure syntaxique de l'opération **clear** applicable aux ports est la suivante:

```
<portId>.clear
```

Le segment de graphe de flux <clear-port-op> sur la Figure 59 définit l'exécution de l'opération **clear** applicable aux ports.

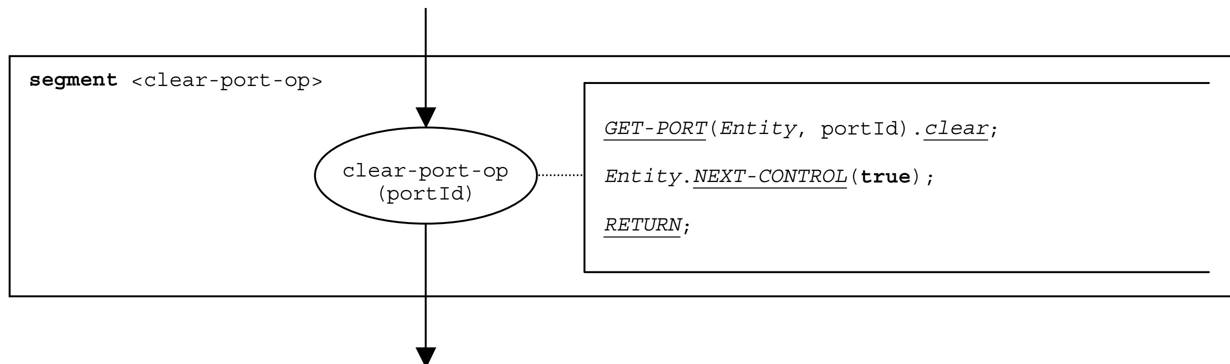


Figure 59/Z.143 – Segment de graphe de flux <clear-port-op>

9.10 Opération connect

La structure syntaxique de l'opération **connect** est la suivante:

connect(<component-expression$_1$>:<portId1>, <component-expression$_2$>:<portId2>)

Les identificateurs <portId1> et <portId2> sont considérés comme étant des identificateurs de port des composants de test correspondants. Il est fait référence aux composants associés aux ports au moyen des références de composant <component-expression$_1$> et <component-expression$_2$>. Les références peuvent être stockées dans des variables ou retournées par une fonction, autrement dit il s'agit d'expressions, dont l'évaluation donne des références de composant. La pile de valeurs est utilisée pour le stockage des références de composant.

L'exécution de l'opération **connect** est définie par le segment de graphe de flux <connect-op> illustré sur la Figure 60. Dans la description du graphe de flux, la première expression à évaluer est <component-expression$_1$> et la deuxième est <component-expression$_2$>, autrement dit l'expression <component-expression$_2$> se trouve au sommet de la pile de valeurs lorsque le nœud connect-op est exécuté.

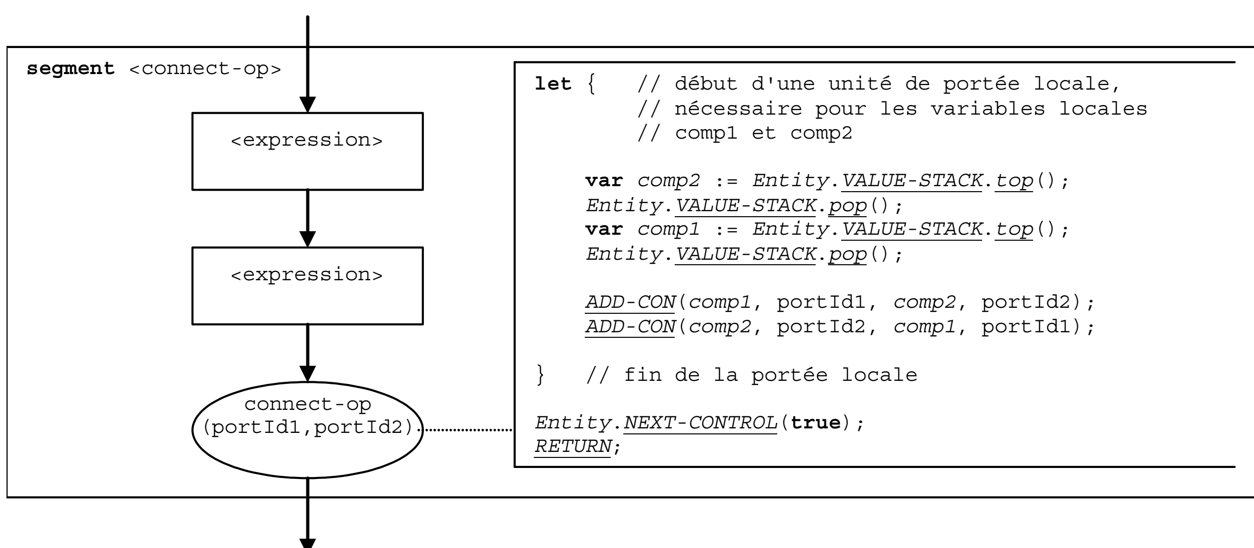


Figure 60/Z.143 – Segment de graphe de flux <connect-op>

9.11 Définition d'une constante

La structure syntaxique de la définition de **constant** est la suivante:

```
const <constType> <constId> := <constType-expression>
```

La valeur d'une constante est considérée comme une expression dont l'évaluation donne une valeur du type de la constante.

NOTE – Les constantes globales sont remplacées par leurs valeurs dans une étape de prétraitement avant que cette sémantique soit appliquée (voir § 9.2). Les constantes locales sont traitées comme des déclarations de variable avec initialisation. Il faut vérifier que l'usage des constantes est correct pendant l'analyse de sémantique statique d'un module TTCN-3, sachant que des constantes ne doivent jamais être présentes dans la partie gauche d'une affectation.

Le segment de graphe de flux <constant-definition> sur la Figure 61 définit l'exécution d'une déclaration de constante pour laquelle la valeur de la constante est donnée sous la forme d'une expression.

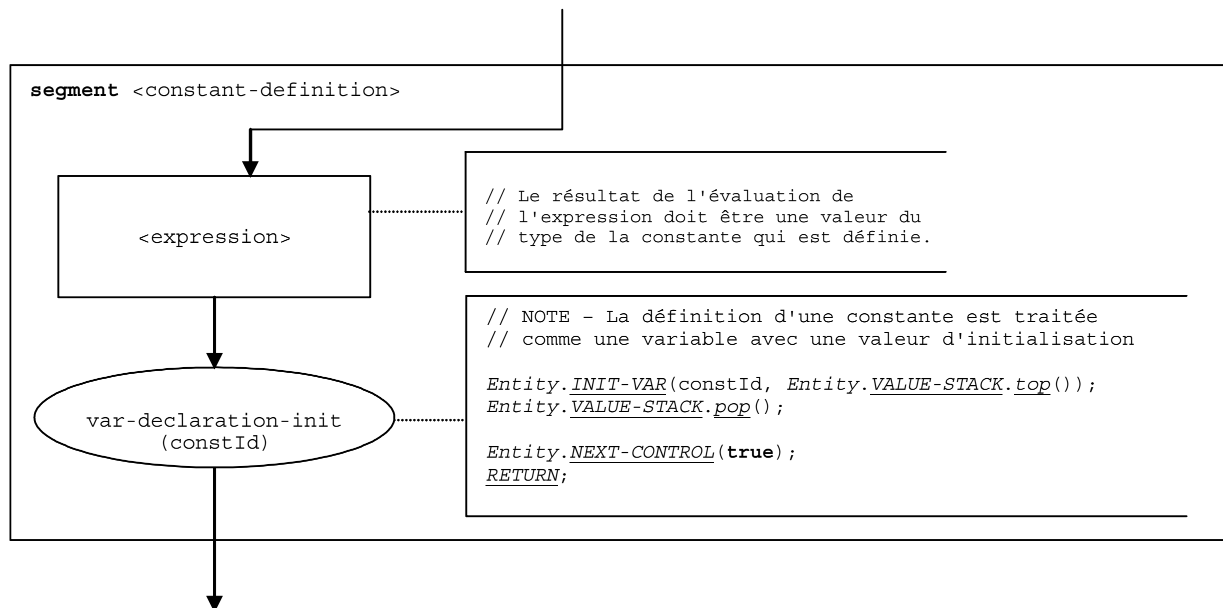


Figure 61/Z.143 – Segment de graphe de flux <constant-definition>

9.12 Opération create

La structure syntaxique de l'opération **create** est la suivante:

```
<componentTypeId>.create
```

Le segment de graphe de flux <create-op> sur la Figure 62 définit l'exécution de l'opération **create**.

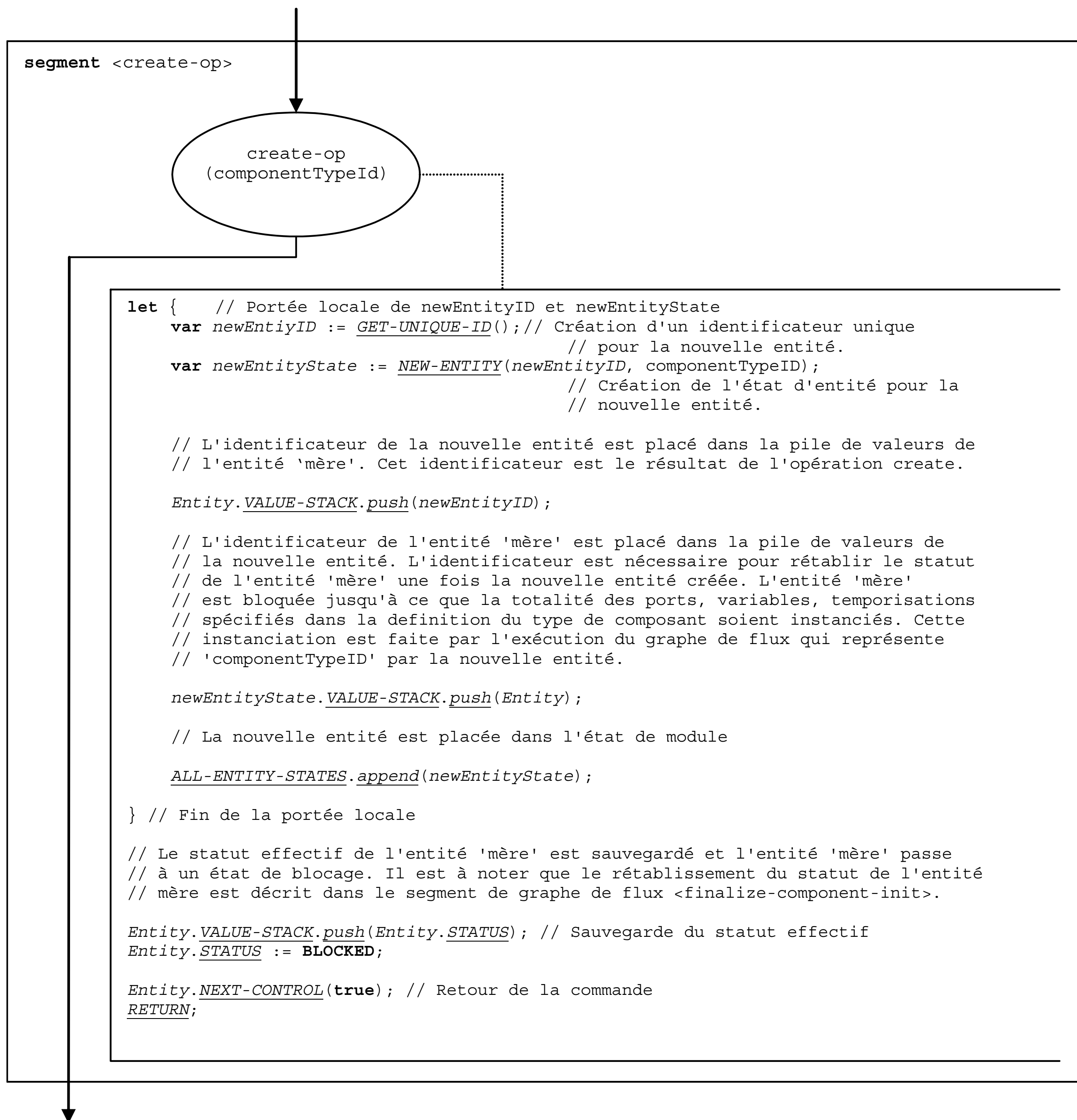


Figure 62/Z.143 – Segment de graphe de flux <create-op>

9.13 Instruction deactivate

La structure syntaxique de l'instruction **deactivate** est la suivante:

```
deactivate [(<default-expression>)]
```

L'instruction **deactivate** spécifie la désactivation de l'un ou de la totalité des comportements par défaut actifs de l'entité qui exécute l'instruction **deactivate**. Si un seul comportement par défaut doit être désactivé, l'expression facultative <default-expression> doit être telle que son évaluation donne une référence identifiant le comportement par défaut à désactiver. L'appel d'une instruction **deactivate** sans expression **<default-expression>** désactive tous les comportements par défaut actifs.

L'exécution d'une instruction **deactivate** est définie par le segment de graphe de flux <deactivate-stmt> sur la Figure 63-a.

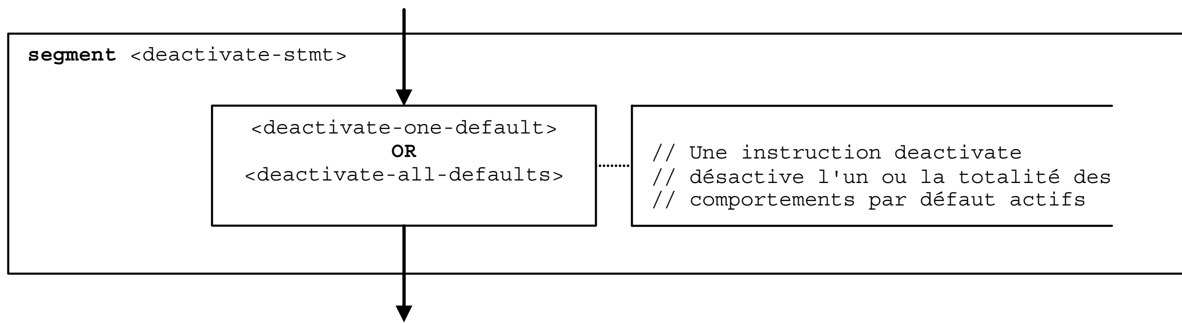


Figure 63-a/Z.143 – Segment de graphe de flux <deactivate-stmt>

9.13.1 Segment de graphe de flux <deactivate-one-default>

Le segment de graphe de flux `<deactivate-one-default>` sur la Figure 63-b spécifie la désactivation d'un seul comportement par défaut actif. L'expression `<default-expression>` doit être telle que son évaluation donne une référence de comportement par défaut. L'expression peut être donnée sous la forme d'une valeur de variable ou d'une fonction retournant une valeur. L'instruction **`deactivate`** supprime le comportement par défaut spécifié de la liste *DEFAULT-LIST* de l'entité qui exécute l'instruction **`deactivate`**.

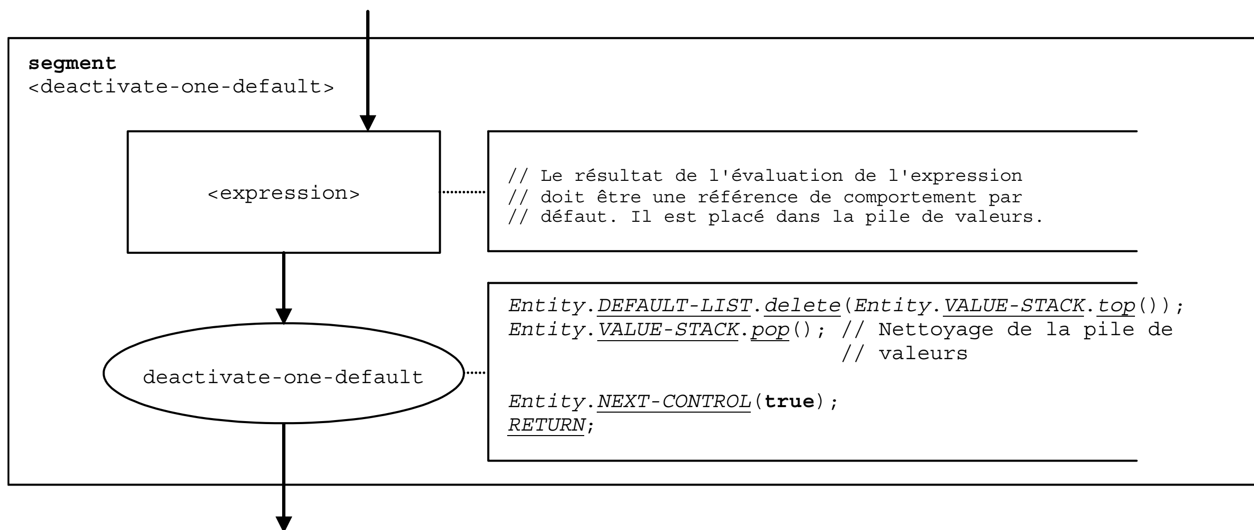


Figure 63-b/Z.143 – Segment de graphe de flux <deactivate-one-default>

9.13.2 Segment de graphe de flux <deactivate-all-defaults>

Le segment de graphe de flux <deactivate-all-defaults> sur la Figure 63-c spécifie la désactivation de tous les comportements par défaut actifs. L'instruction **`deactivate`** supprime la liste *DEFAULT-LIST* de l'entité qui exécute l'instruction **`deactivate`**.

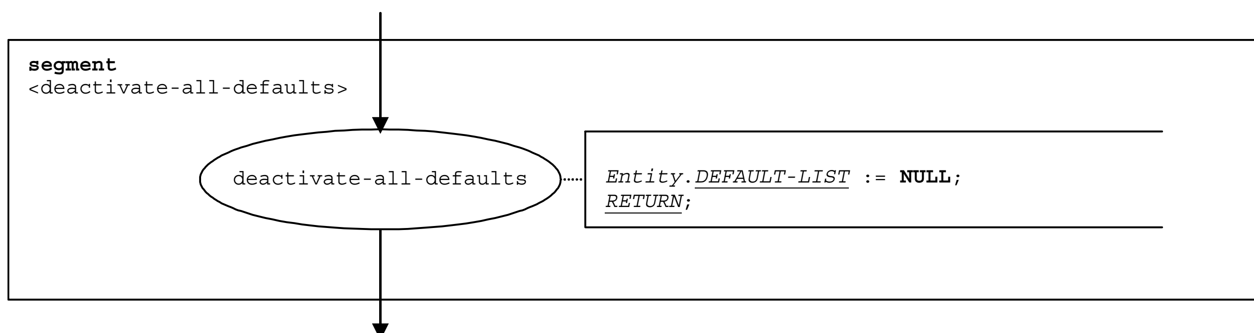


Figure 63-c/Z.143 – Segment de graphe de flux <deactivate-all-defaults>

9.14 Opération disconnect

La structure syntaxique de l'opération **disconnect** est la suivante:

```
disconnect(<component-expression1>:<portId1>,
    <component-expression2>:<portId2>)
```

Les identificateurs <portId1> et <portId2> sont considérés comme étant des identificateurs de port des composants de test correspondants. Il est fait référence aux composants associés aux ports au moyen des références de composant <component-expression$_1$> et <component-expression$_2$>. Les références peuvent être stockées dans des variables ou retournées par des fonctions, autrement dit il s'agit d'expressions, dont l'évaluation donne des références de composant. La pile de valeurs est utilisée pour le stockage des références de composant.

L'exécution de l'opération **disconnect** est définie par le segment de graphe de flux <disconnect-op> illustré sur la Figure 64. Dans le segment de graphe de flux, la première expression à évaluer est <component-expression$_1$> et la deuxième est <component-expression$_2$>, autrement dit l'expression <component-expression$_2$> se trouve au sommet de la pile de valeurs lorsque le nœud disconnect-op est exécuté.

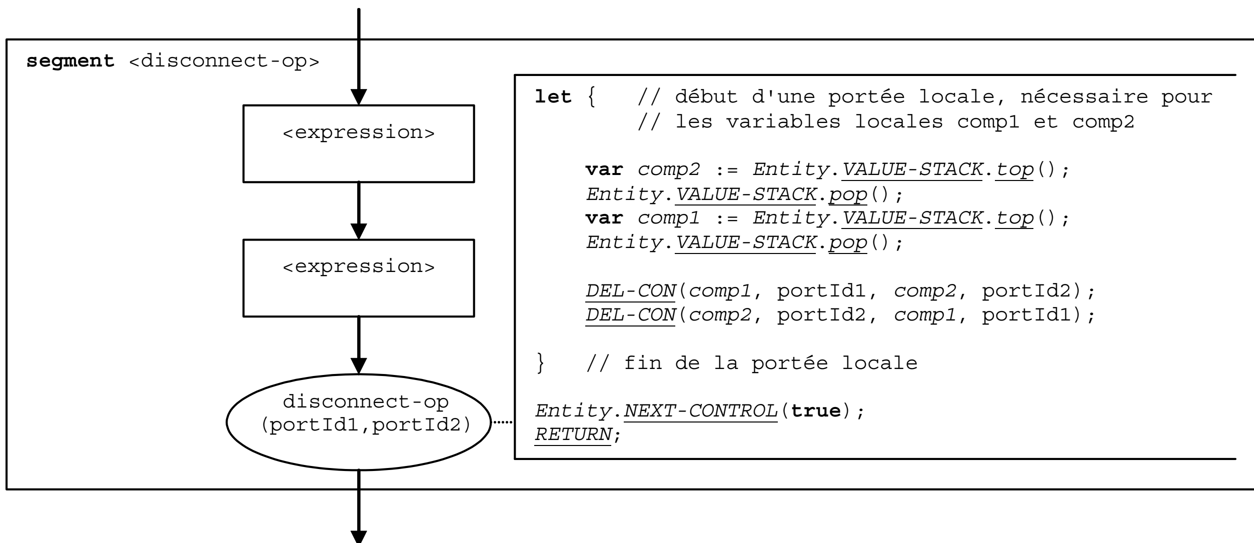


Figure 64/Z.143 – Segment de graphe de flux <disconnect-op>

9.15 Instruction do-while

La structure syntaxique de l'instruction **do-while** est la suivante:

```
do <statement-block>
while (<boolean-expression>)
```

L'exécution d'une instruction **do-while** est définie par le segment de graphe de flux <do-while-stmt> illustré sur la Figure 65.

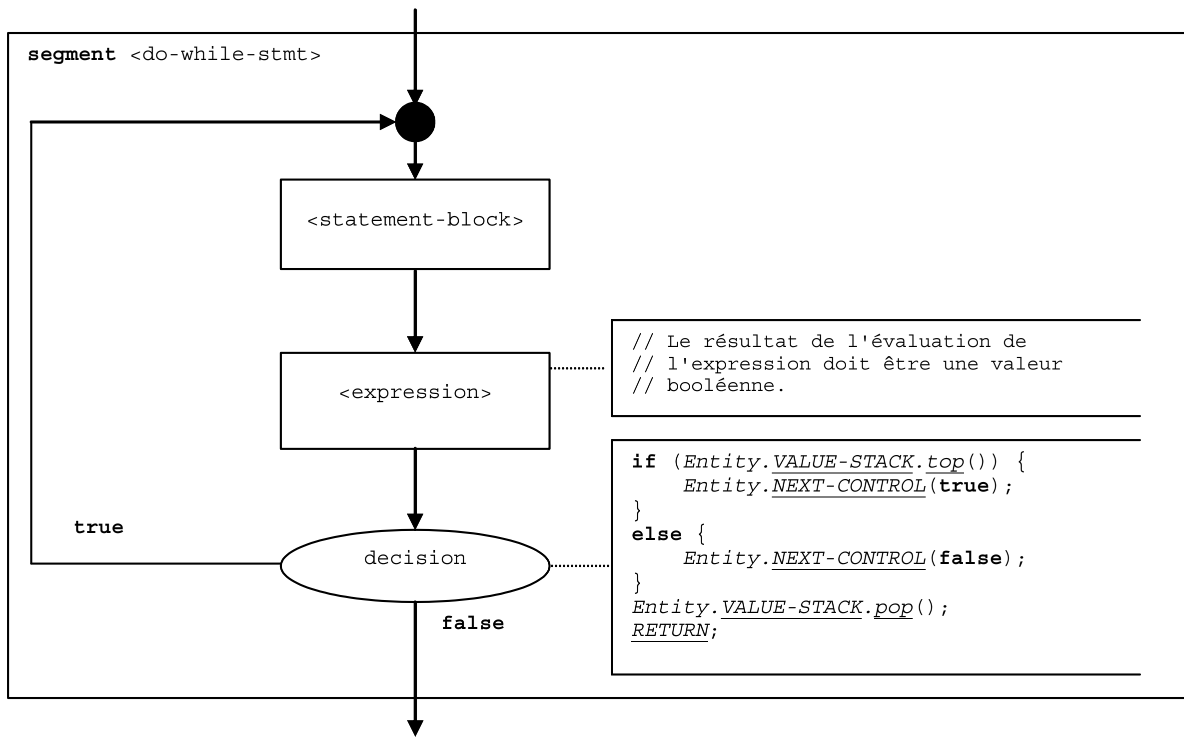


Figure 65/Z.143 – Segment de graphe de flux <do-while-stmt>

9.16 Opération done applicable aux composants

La structure syntaxique de l'opération **`done`** applicable aux composants est la suivante:

```
<component-expression>.done
```

L'opération **`done`** applicable aux composants permet de vérifier si un composant est en cours ou s'il a été arrêté. Suivant si le composant vérifié est en cours ou s'il a été arrêté, l'opération **`done`** décide des modalités de la poursuite du flux de commande. Une référence de composant identifie le composant à vérifier. Cette référence peut être stockée dans une variable ou être retournée par une fonction, autrement dit il s'agit d'une expression. Dans un souci de simplicité, les mots clés '**`all component`**' et '**`any component`**' sont considérés comme des expressions spéciales.

Le segment de graphe de flux `<done-op>` sur la Figure 66 définit l'exécution de l'opération **`done`** applicable aux composants.

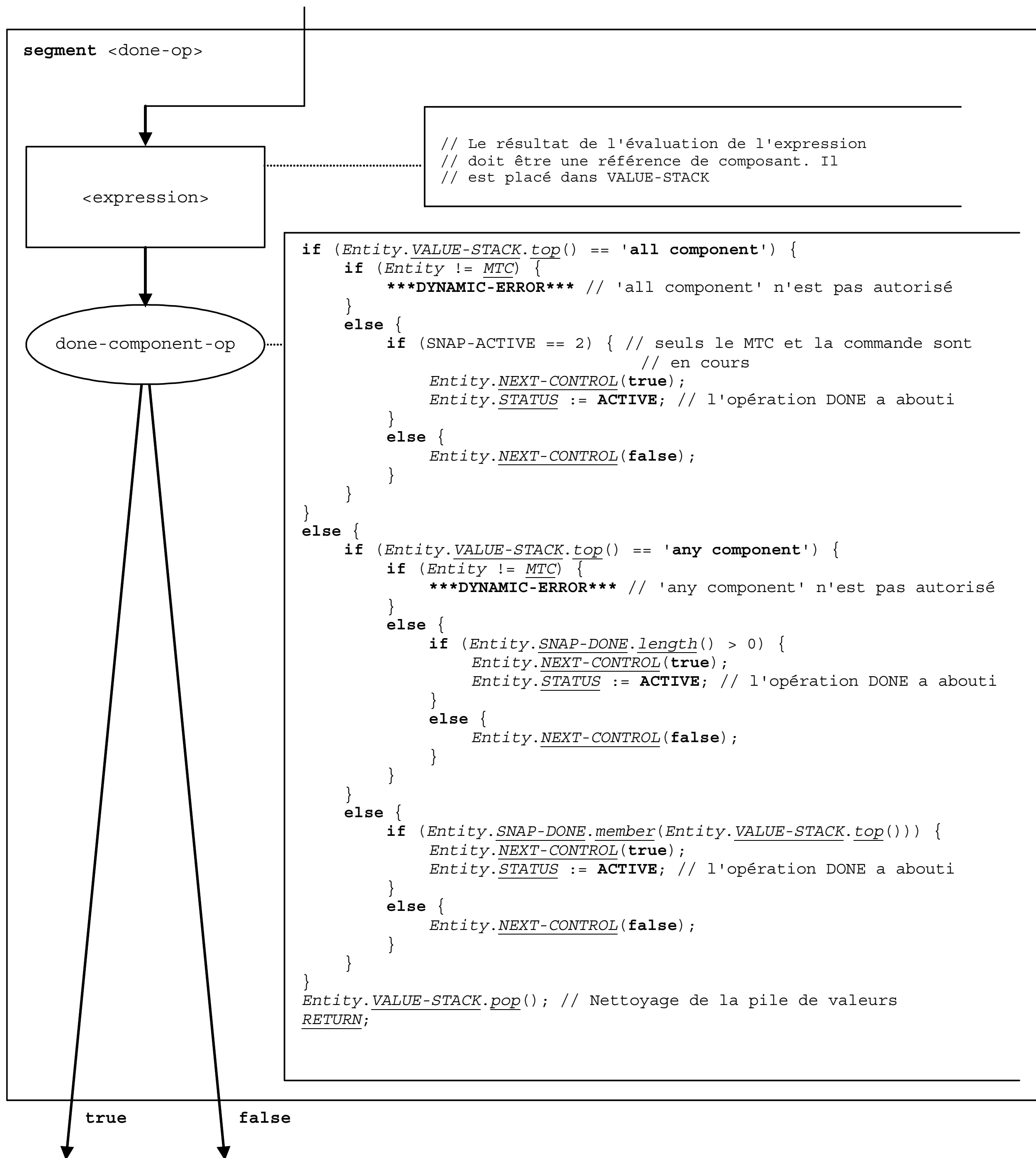


Figure 66/Z.143 – Segment de graphe de flux <done-component-op>

9.17 Instruction execute

La structure syntaxique de l'instruction **`execute`** est la suivante:

```
execute(<testCaseId>([<act-par1>, ... , <act-parn>)]) [, <float-expression>])
```

L'instruction **`execute`** décrit l'exécution d'un test élémentaire `<testCaseId>` avec les paramètres effectifs (facultatifs) `<act-par`$_1$`>, … , <act-par`$_n$`>`. Elle peut facultativement être protégée par une durée donnée sous la forme d'une expression dont l'évaluation donne une valeur de type **`float`**. Si, pendant la durée spécifiée, le test élémentaire ne retourne pas de verdict, une exception d'expiration de temporisation se produit, le test élémentaire est arrêté et un verdict **`error`** est retourné.

NOTE – La sémantique opérationnelle modélise l'arrêt du test élémentaire par un arrêt du composant MTC. En réalité, d'autres mécanismes pourront être plus appropriés.

Si aucune exception d'expiration 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 prenne fin et, pour la poursuite de l'exécution du test élémentaire, le flux de commande est donné au composant MTC. Le flux de commande est redonné à l'instance de commande lorsque le composant MTC prend fin.

Le segment de graphe de flux `<execute-stmt>` sur la Figure 67 définit l'exécution d'une instruction **`execute`**.

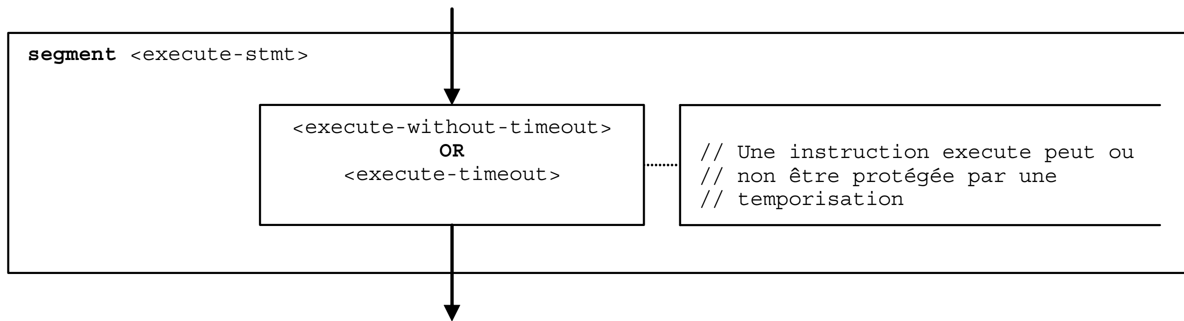


Figure 67/Z.143 – Segment de graphe de flux <execute-stmt>

9.17.1 Segment de graphe de flux <execute-without-timeout>

L'exécution d'un test élémentaire commence avec la création du composant **`mtc`**, qui est ensuite démarré avec le comportement défini dans la définition du test élémentaire. Ensuite, la commande de module attend jusqu'à ce que le test élémentaire prenne fin. La création et le démarrage du composant MTC peuvent être décrits au moyen des instructions **`create`** et **`start`**:

```
var mtcType MyMTC := mtcType.create;
MyMTC.start(TestCaseName(P1...Pn));
```

Le segment de graphe de flux `<execute-without-timeout>` sur la Figure 68 définit l'exécution d'une instruction **`execute`** sans l'occurrence d'une exception d'expiration de temporisation, au moyen des segments de graphe de flux des opérations **`create`** et **`start`**.

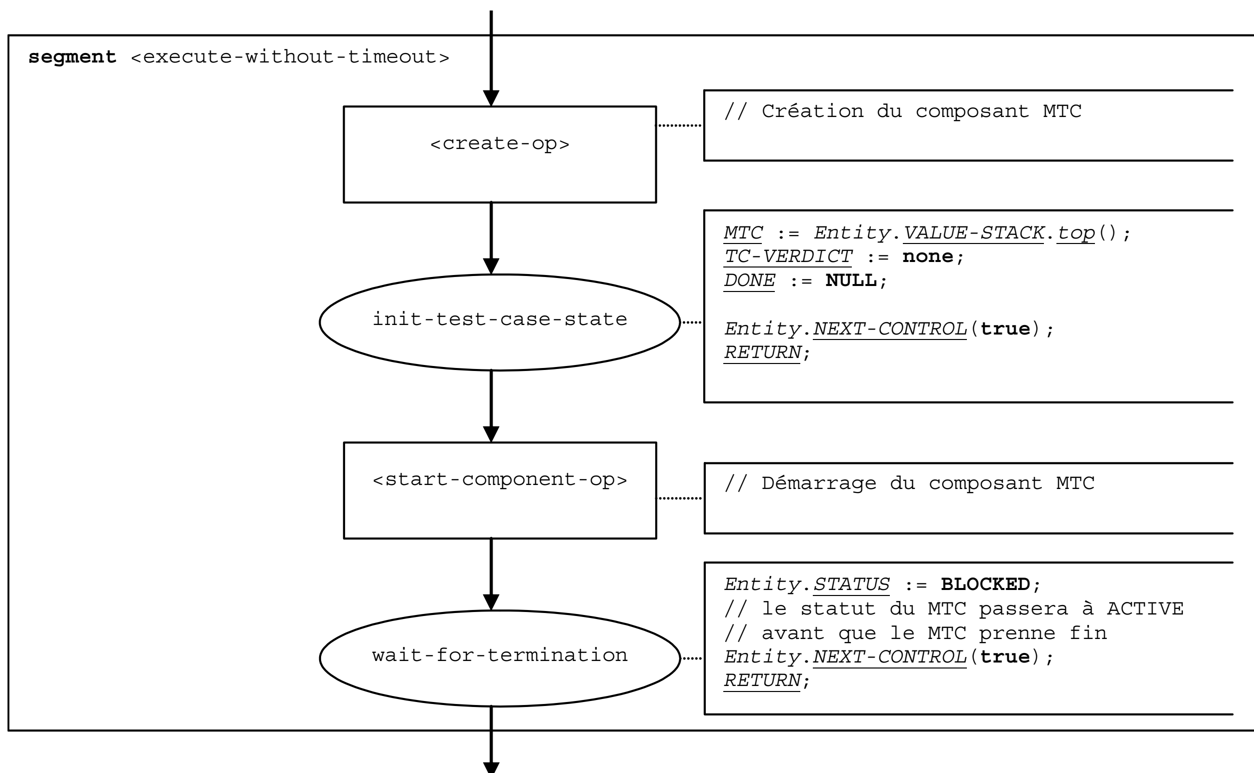


Figure 68/Z.143 – Segment de graphe de flux <execute-without-timeout>

9.17.2 Segment de graphe de flux <execute-timeout>

Le segment de graphe de flux <execute-timeout> sur la Figure 69 définit l'exécution d'une instruction **execute** qui est protégée par une temporisation. Le segment de graphe de flux modélise aussi la création et le démarrage du composant MTC par des opérations **create** et **start**. En outre, *TIMER-GUARD* protège la terminaison.

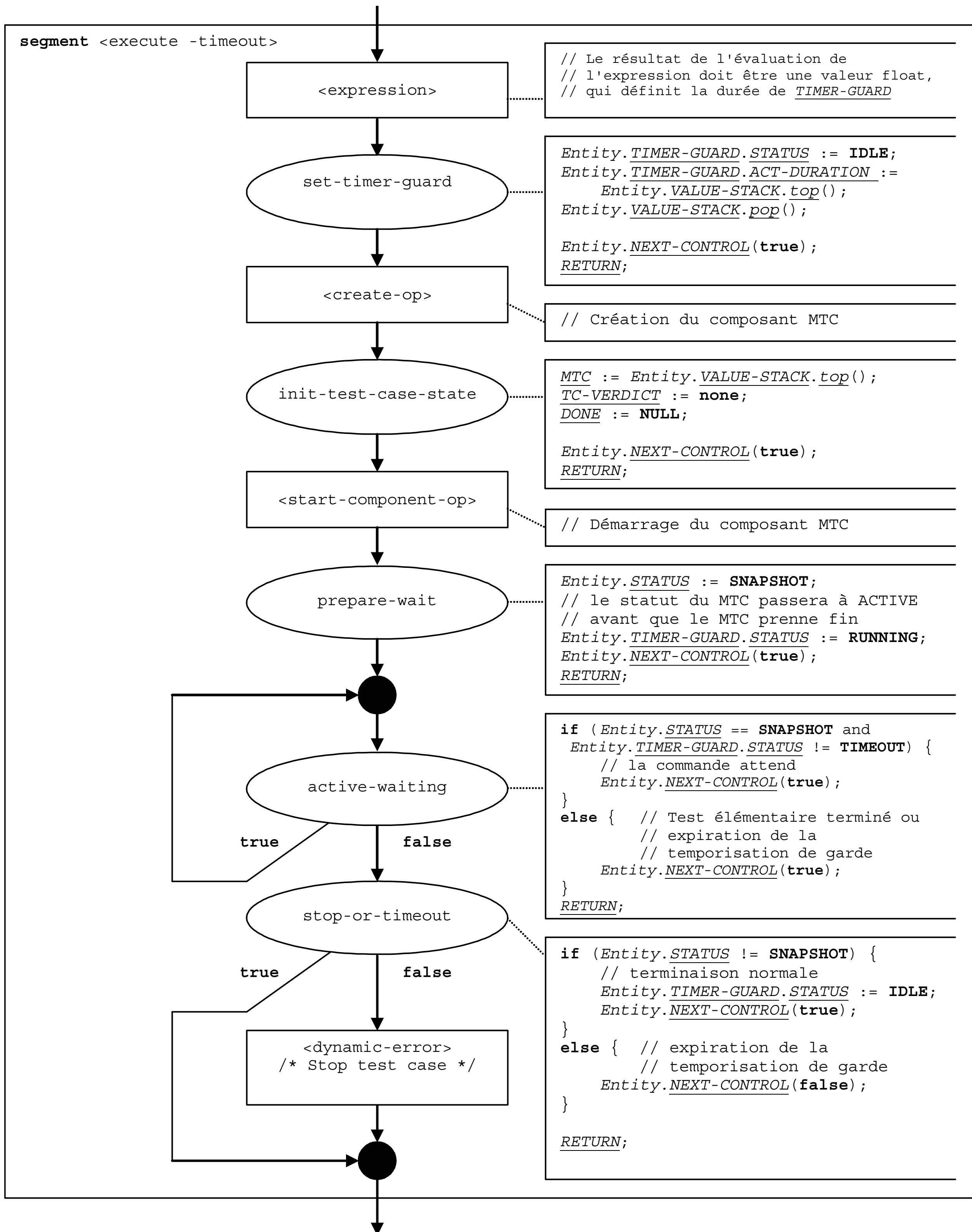


Figure 69/Z.143 – Segment de graphe de flux <execute-timeout>

9.18 Expression

Pour le traitement des expressions, il faut distinguer les quatre cas suivants:

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 la suivante:

```
<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 mathématiques comme +, –, **not**, etc.

L'exécution d'une expression est définie par le segment de graphe de flux <expression> illustré sur la Figure 70.

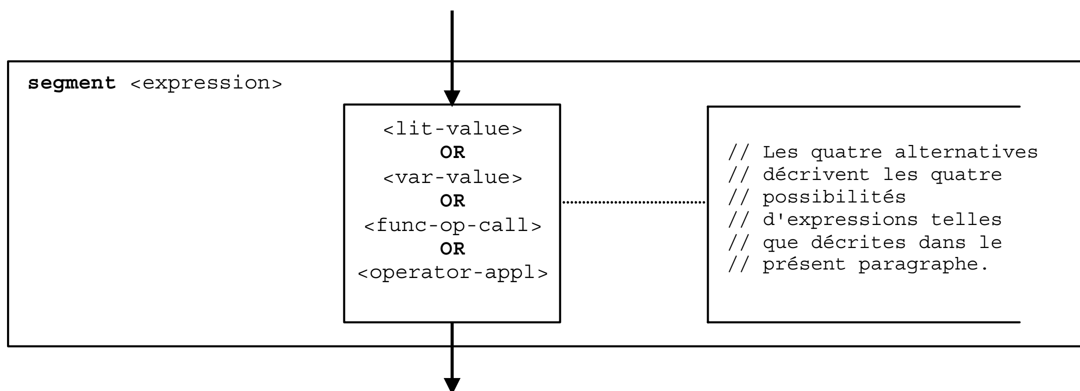


Figure 70/Z.143 – Segment de graphe de flux <expression>

9.18.1 Segment de graphe de flux <lit-value>

Le segment de graphe de flux <lit-value> sur la Figure 71 place une valeur de littéral dans la pile de valeurs d'une entité.

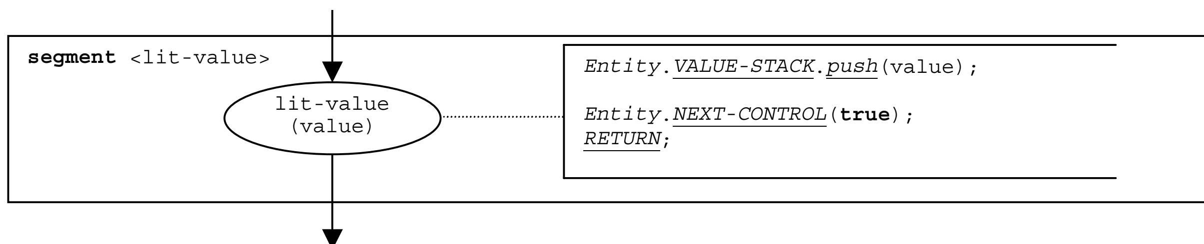


Figure 71/Z.143 – Segment de graphe de flux <lit-value>

9.18.2 Segment de graphe de flux <var-value>

Le segment de graphe de flux <var-value> sur la Figure 72 place la valeur d'une variable dans la pile de valeurs d'une entité.

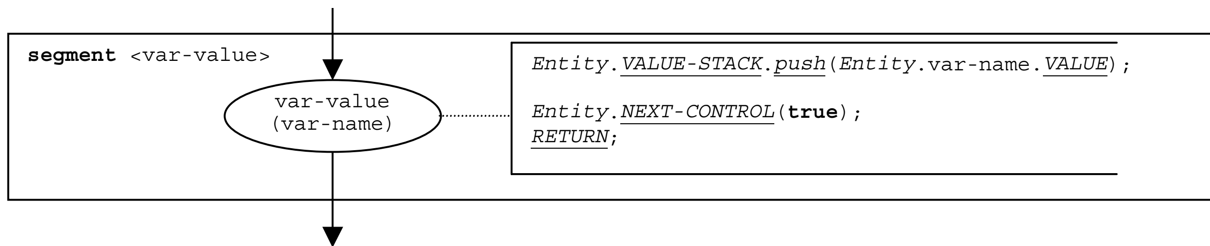


Figure 72/Z.143 – Segment de graphe de flux <var-value>

9.18.3 Segment de graphe de flux <func-op-call>

Le segment de graphe de flux `<func-op-call>` sur la Figure 73 fait référence à des appels de fonctions et d'opérations, qui retournent une valeur qui est placée dans la pile de valeurs d'une entité. Tous ces appels sont considérés comme des expressions.

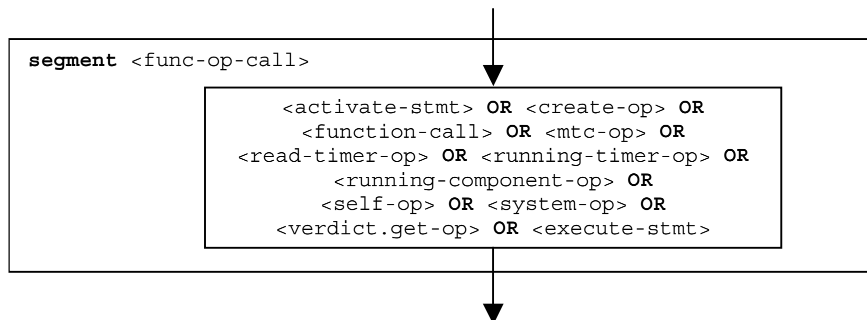


Figure 73/Z.143 – Segment de graphe de flux <func-op-call>

9.18.4 Segment de graphe de flux <operator-appl>

La représentation de graphe de flux sur la Figure 74 repose directement sur l'hypothèse de l'utilisation de la notation polonaise inversée pour l'évaluation des expressions contenant un opérateur. Les opérandes sont calculés et placés dans la pile d'évaluation. Pour l'application de l'opérateur, les opérandes sont supprimés de pile d'évaluation et l'opérateur est appliqué. Le résultat de l'application de l'opérateur est ensuite placé dans la pile d'évaluation. La suppression des opérandes et le placement du résultat dans la pile sont considérés comme faisant partie de l'application de l'opérateur (instruction `Entity.APPLY-OPERATOR(operator)` sur la Figure 74), autrement dit ils ne sont pas modélisés par la sémantique opérationnelle.

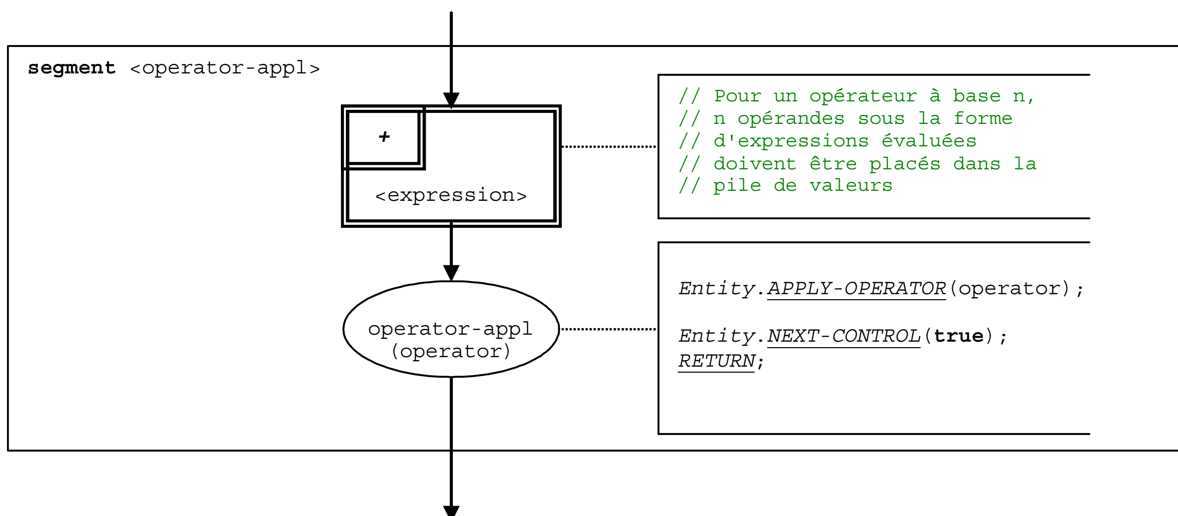


Figure 74/Z.143 – Segment de graphe de flux <operator-appl>

9.18b Segment de graphe de flux <dynamic-error>

En cas d'erreur dynamique, le segment de graphe de flux `<dynamic-error>` (voir Figure 74-b) est invoqué par le système de test. Toutes les ressources attribuées au test élémentaire sont supprimées et le verdict **error** est affecté au test élémentaire. La commande est donnée à l'instruction de la partie commande qui suit l'instruction **`execute`** dans laquelle l'erreur s'est produite.

Le segment de graphe de flux `<dynamic-error>` est invoqué par la commande de module si un test élémentaire ne se termine pas dans le délai spécifié (voir § 9.17.2).

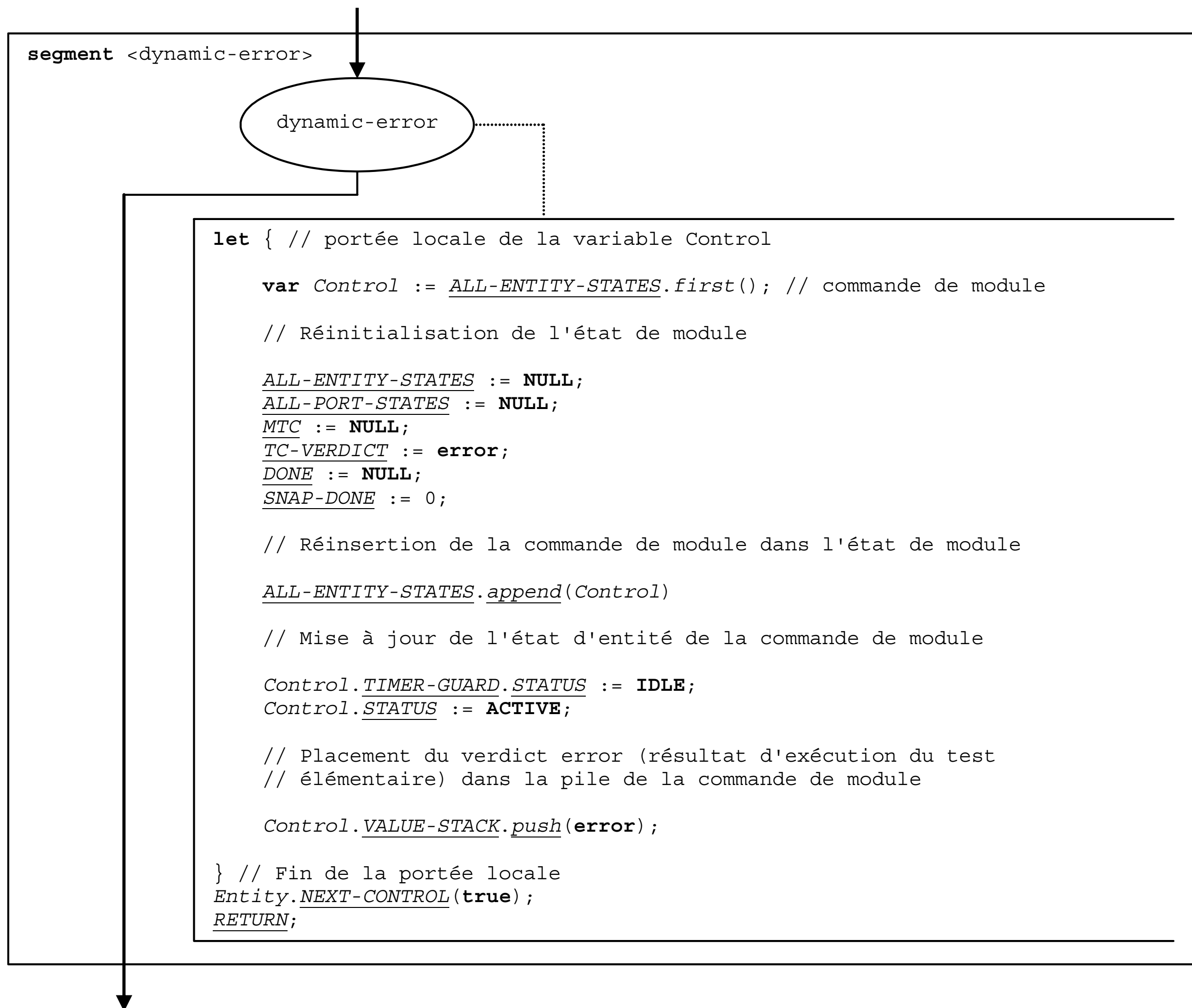


Figure 74-b/Z.143 – Segment de graphe de flux <dynamic-error>

9.19 Segment de graphe de flux <finalize-component-init>

Le segment de graphe de flux `<finalize-component-init>` fait partie du graphe de flux représentant le comportement d'une définition de type de composant. Son exécution est définie sur la Figure 75.

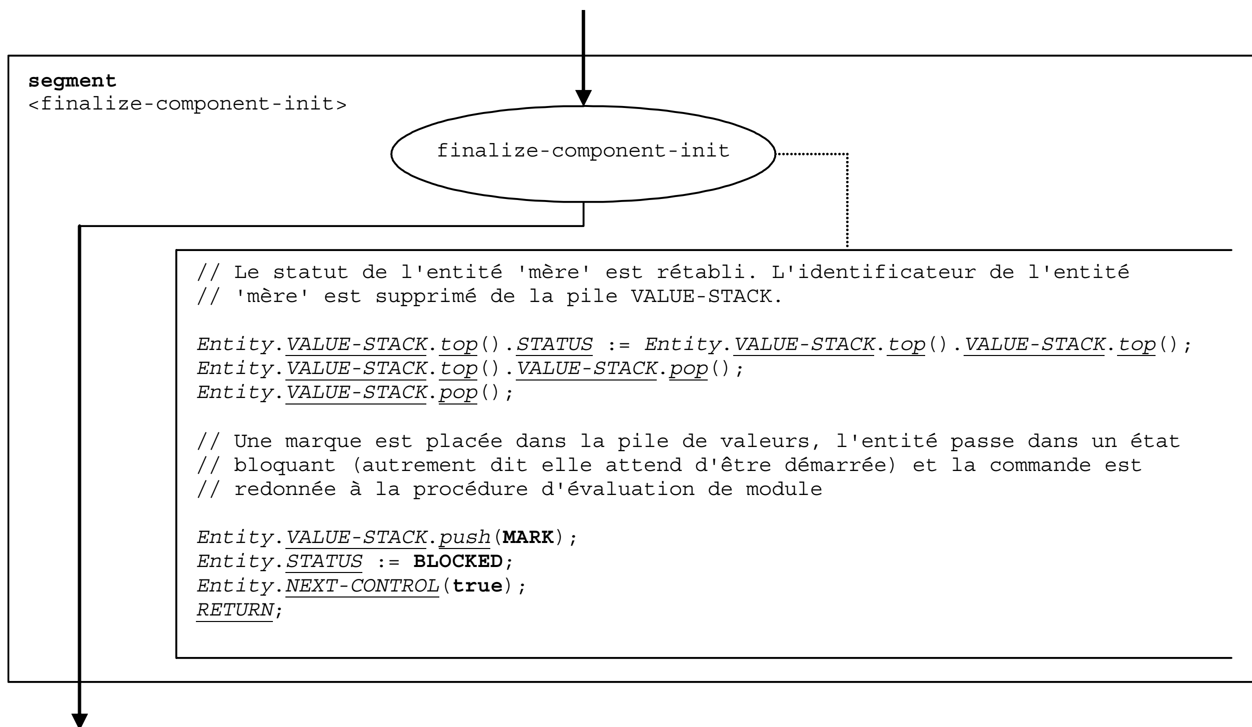


Figure 75/Z.143 – Segment de graphe de flux <finalize-component-init>

9.20 Segment de graphe de flux <init-component-scope>

Le segment de graphe de flux `<init-component-scope>` fait partie du graphe de flux représentant le comportement d'une définition de type de composant. Son exécution est définie sur la Figure 76.

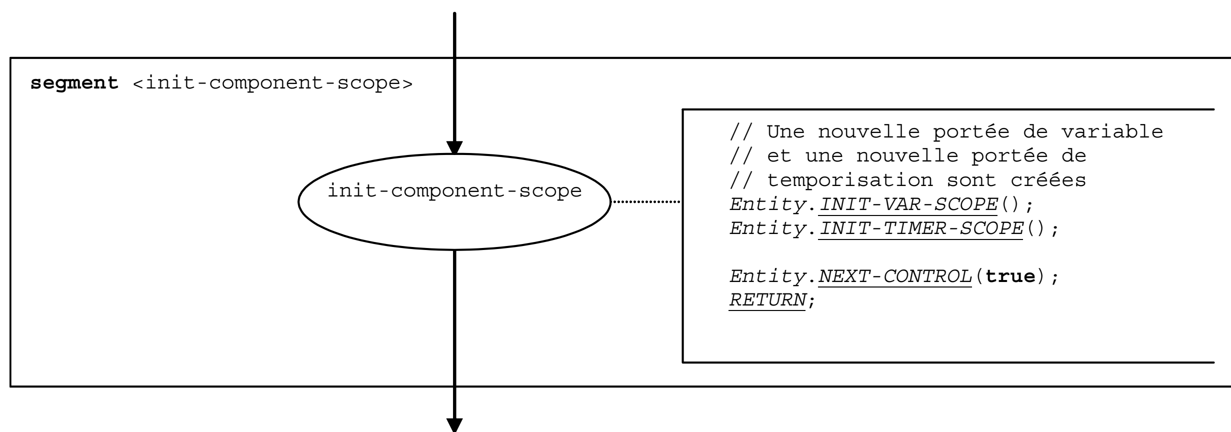


Figure 76/Z.143 – Segment de graphe de flux <init-component-scope>

9.21 Segment de graphe de flux <parameter-handling>

Le segment de graphe de flux `<parameter-handling>` est utilisé au début des graphes de flux représentant des tests élémentaires, des variantes et des fonctions. Il initialise une nouvelle portée et crée des variables et des temporisations pour le traitement des paramètres. Dans le segment de graphe de flux `<parameter-handling>`, on suppose que l'enregistrement d'appel du test élémentaire, de la variante ou de la fonction qui est appelé se trouve au sommet de la pile de valeurs.

L'exécution du segment de graphe de flux `<parameter-handling>` est illustrée sur la Figure 77.

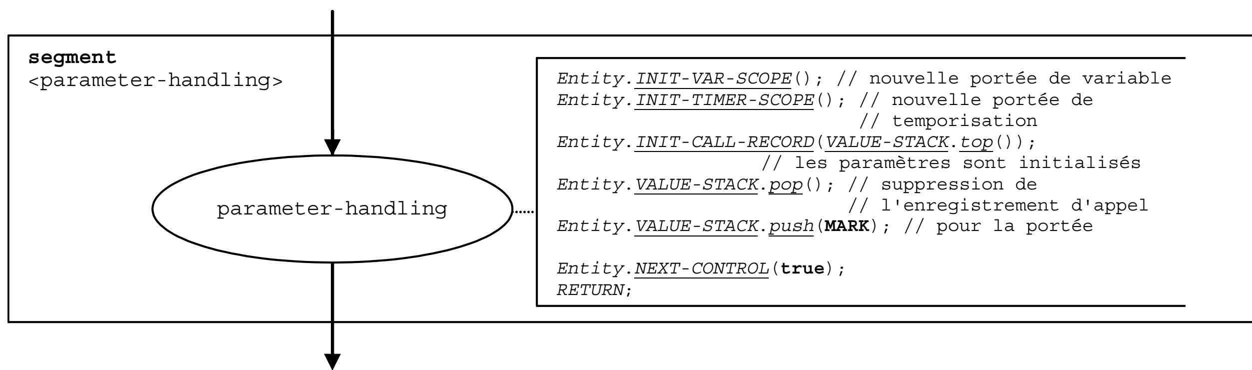


Figure 77/Z.143 – Segment de graphe de flux <parameter-handling>

9.22 Segment de graphe de flux <statement-block>

La structure syntaxique d'un bloc d'instructions est la suivante:

```
{ <statement1>; ... ; <statementn> }
```

Un bloc d'instructions est une unité de portée. Au moment d'entrer dans une unité de portée, de nouvelles portées pour les variables, les temporisations et la pile de valeurs doivent être initialisées. Au moment de quitter une unité de portée, la totalité des variables, des temporisations et des valeurs de pile de cette portée doivent être détruites.

NOTE 1 – Le bloc d'instructions n'est pas un concept 'officiel' de la notation TTCN-3. Les blocs d'instructions ne peuvent figurer que dans le corps de fonctions, de variantes, de tests élémentaires et de la commande de module ainsi que dans des instructions composites, par exemple **alt**, **if-else** ou **do-while**.

NOTE 2 – Les opérations de réception et les appels de variante ne peuvent pas figurer dans des blocs d'instructions, ils sont imbriqués dans des instructions **alt** ou des opérations **call**.

NOTE 3 – La sémantique opérationnelle traite aussi les opérations et les déclarations comme des instructions, autrement dit les opérations et les instructions sont autorisées dans les blocs d'instructions.

NOTE 4 – Certaines fonctions TTCN-3, comme par exemple **system** ou **self**, sont considérées comme des expressions, qui ne sont pas utiles en tant qu'instructions autonomes dans des blocs d'instructions. Leurs représentations sous forme de graphe de flux ne sont pas énumérées sur la Figure 78.

Le segment de graphe de flux `<statement-block>` sur la Figure 78 définit l'exécution d'un bloc d'instructions.

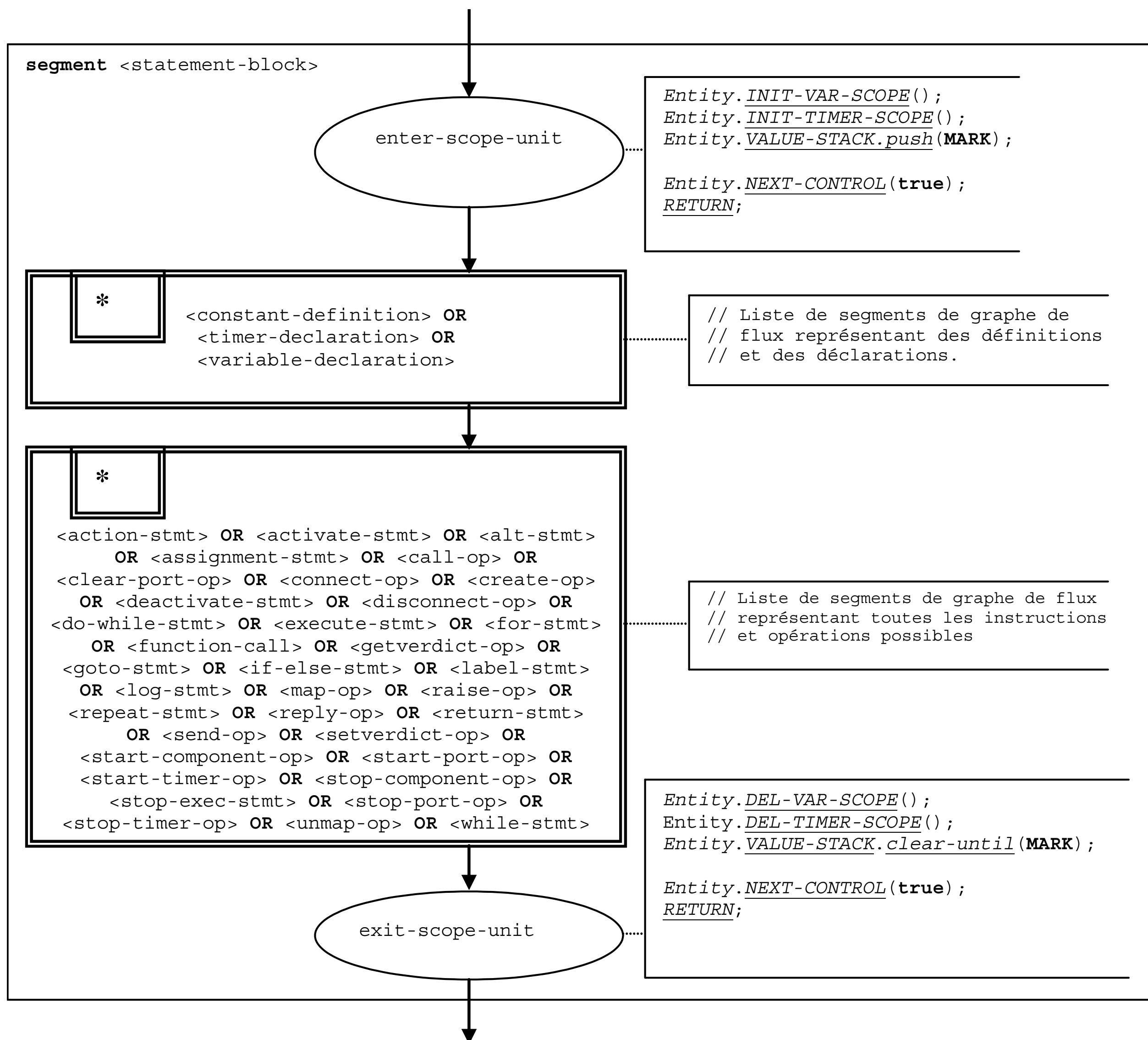


Figure 78/Z.143 – Segment de graphe de flux <statement-block>

9.23 Instruction for

La structure syntaxique de l'instruction **`for`** est la suivante:

```
for (<assignment>|<variable-declaration>, <boolean expression>, <assignment>) <statement-block>
```

L'initialisation de la variable d'indice et sa manipulation correspondante sont considérées comme des affectations faites à la variable d'indice. Il est également autorisé de déclarer et d'initialiser la variable d'indice directement dans l'instruction **`for`**. L'expression `<boolean-expression>` décrit le critère de terminaison de la boucle spécifiée par l'instruction **`for`** et le bloc `<statement-block>` décrit le corps de la boucle.

L'exécution de l'instruction **`for`** est définie par le segment de graphe de flux `<for-stmt>` illustré sur la Figure 79. L'affectation `<assignment>` initiale ou la déclaration de variable alternative avec l'affectation `<var-declaration-init>` (voir § 9.57.1) décrit l'initialisation de la variable d'indice. L'affectation `<assignment>` figurant dans la branche **`true`** du nœud `decision` décrit la manipulation de la variable d'indice. L'instruction **`for`** est une unité de portée pour une variable d'indice nouvellement déclarée, ce qui est modélisé au moyen des nœuds `enter-var-scope` et `exit-var-scope`.

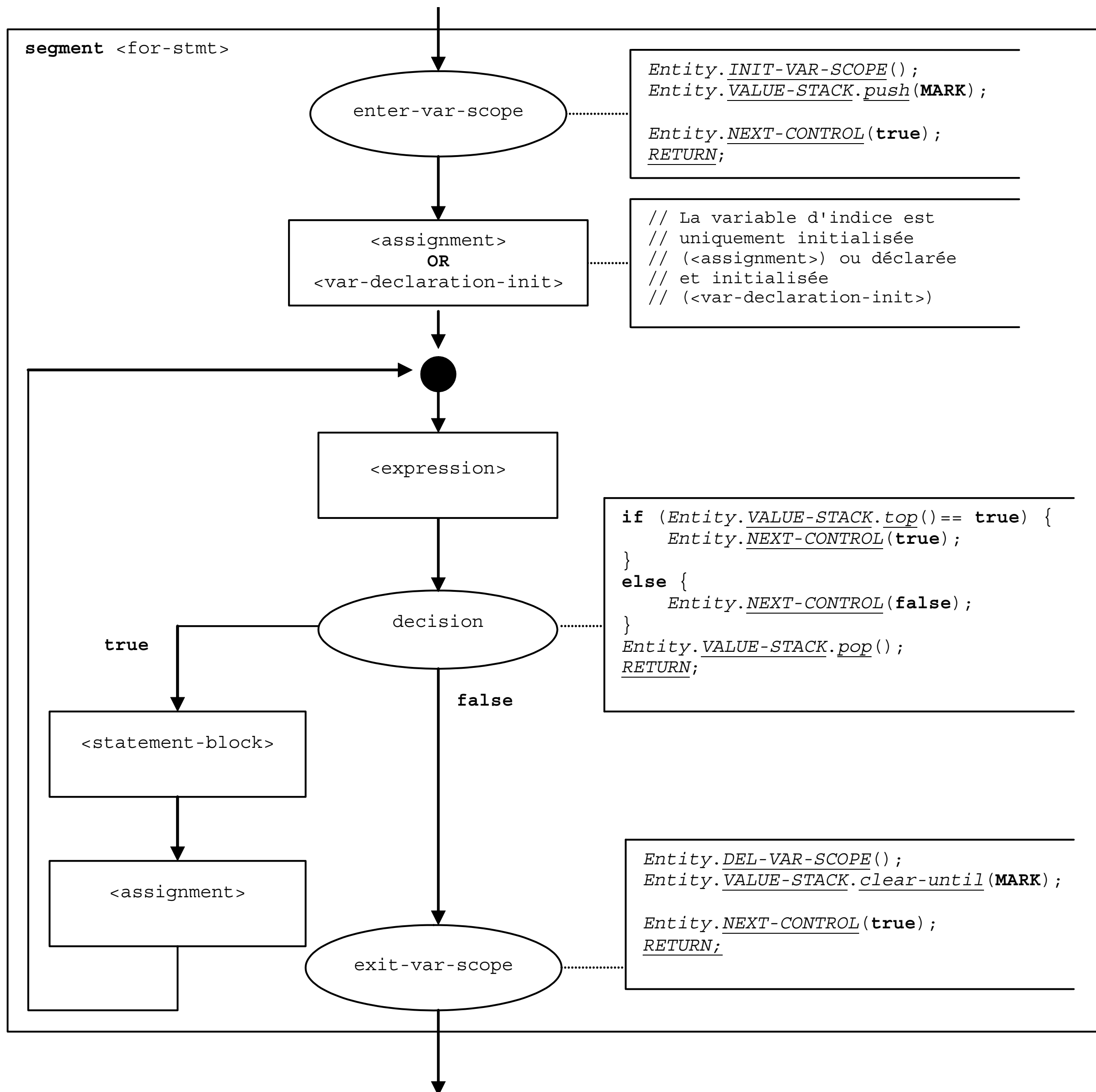


Figure 79/Z.143 – Segment de graphe de flux <for-stmt>

9.24 Appel de fonction

La structure syntaxique d'un appel de fonction est la suivante:

<function-name>([<act-par-desc$_1$>, ... , <act-par-desc$_n$>])

Le nom <function-name> est le nom d'une fonction et les descriptions <act-par-descr$_1$>, ... , <act-par-descr$_n$> décrivent les valeurs des paramètres effectifs de l'appel de fonction.

NOTE 1 – Un appel de fonction et un appel de variante sont traités de la même manière. Par conséquent, l'appel de variante (voir § 9.4) fait référence au présent paragraphe.

On suppose que pour chaque description <act-par-desc$_1$>, l'identificateur de paramètre formel correspondant <f-par-Id$_1$> est connu, autrement dit il est possible d'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 de flux <function-call> sur la Figure 80 définit l'exécution d'un appel de fonction. L'exécution est structurée en trois étapes. Dans la première étape, un enregistrement d'appel pour la fonction <function-name> est créé. Dans la deuxième étape, les valeurs des paramètres effectifs sont calculées et affectées au champ correspondant dans l'enregistrement d'appel. Dans la troisième étape, deux situations doivent être distinguées: la fonction appelée est une fonction définie par l'utilisateur (<user-def-func-call>), autrement dit il existe une représentation de la

fonction sous forme de graphe de flux, ou la fonction appelée est une fonction prédéfinie ou externe (`<predef-ext-func-call>`). Dans le cas de l'appel d'une fonction définie par l'utilisateur, la commande est donnée à la fonction appelée. Dans le cas d'une fonction prédéfinie ou externe, on suppose que l'enregistrement d'appel peut être utilisé pour exécuter la fonction en une seule étape. Le traitement correct des paramètres par référence et de la valeur de retour (qui doit être placée dans la pile de valeurs) relève de la fonction appelée, autrement dit il sort du cadre de cette sémantique opérationnelle.

NOTE 2 – Si l'appel de fonction modélise un appel de variante, seule la branche `<user-def-func-call>` sera choisie, car il existe une représentation de la variante appelée sous forme de graphe de flux.

NOTE 3 – Le segment `<function call>` est également utilisé pour décrire le démarrage du composant MTC dans une instruction **`execute`**. Dans ce cas, un enregistrement d'appel pour le test élémentaire est construit et seule la branche `<user-def-func-call>` sera choisie.

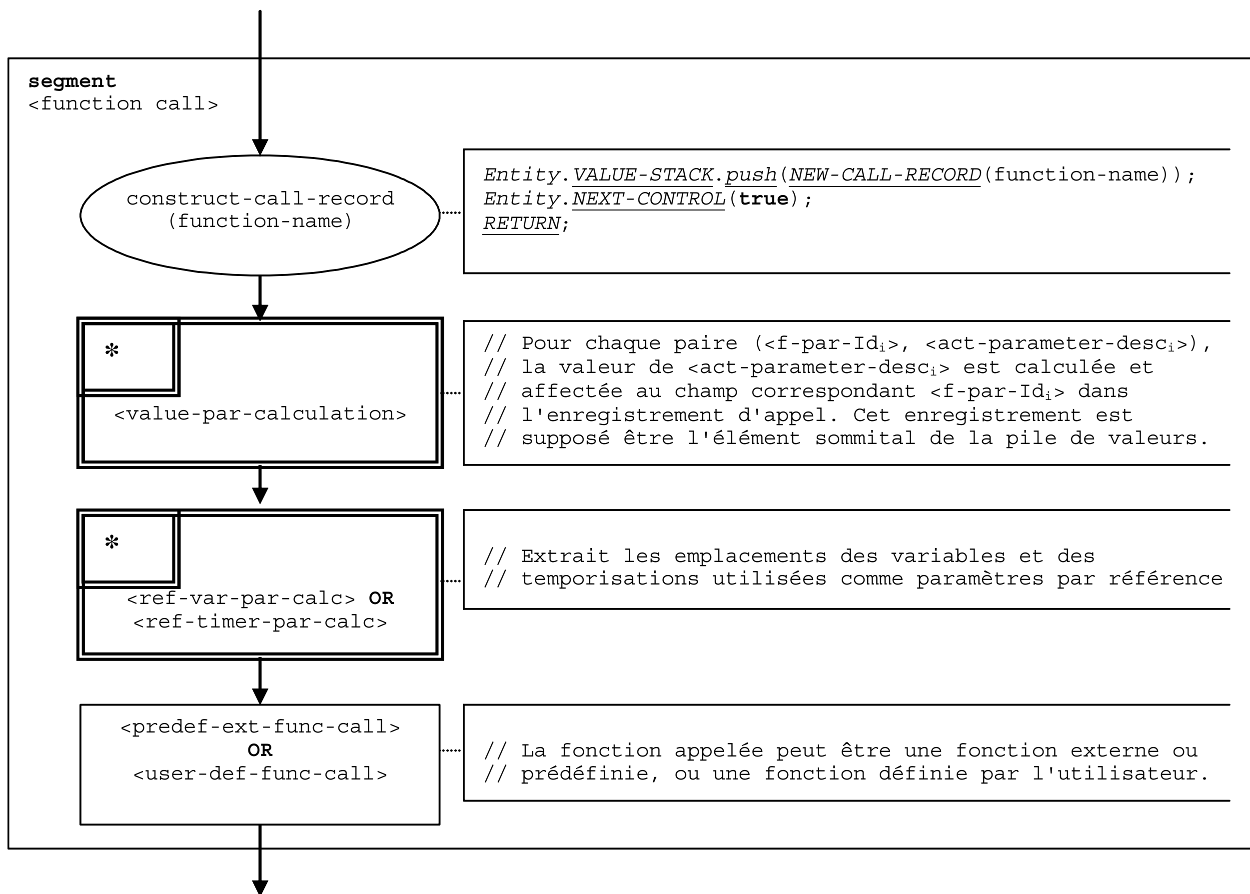


Figure 80/Z.143 – Segment de graphe de flux <function-call>

9.24.1 Segment de graphe de flux <value-par-calculation>

Le segment de graphe de flux `<value-par-calculation>` sert à calculer les valeurs des paramètres effectifs et à les affecter aux champs correspondants dans les enregistrements d'appel pour les fonctions, les variantes et les tests élémentaires.

On suppose qu'un enregistrement d'appel est l'élément sommital de la pile de valeurs et qu'une paire:

```
(<f-par-Idi>, <act-parameter-desci>)
```

doit être traitée. La description `<act-parameter-desc`$_i$`>` doit être évaluée et `<f-par-Id`$_i$`>` est l'identificateur d'un paramètre formel qui a un champ correspondant dans l'enregistrement d'appel figurant dans la pile de valeurs.

L'exécution du segment de graphe de flux `<value-par-calculation>` est illustrée sur la Figure 81.

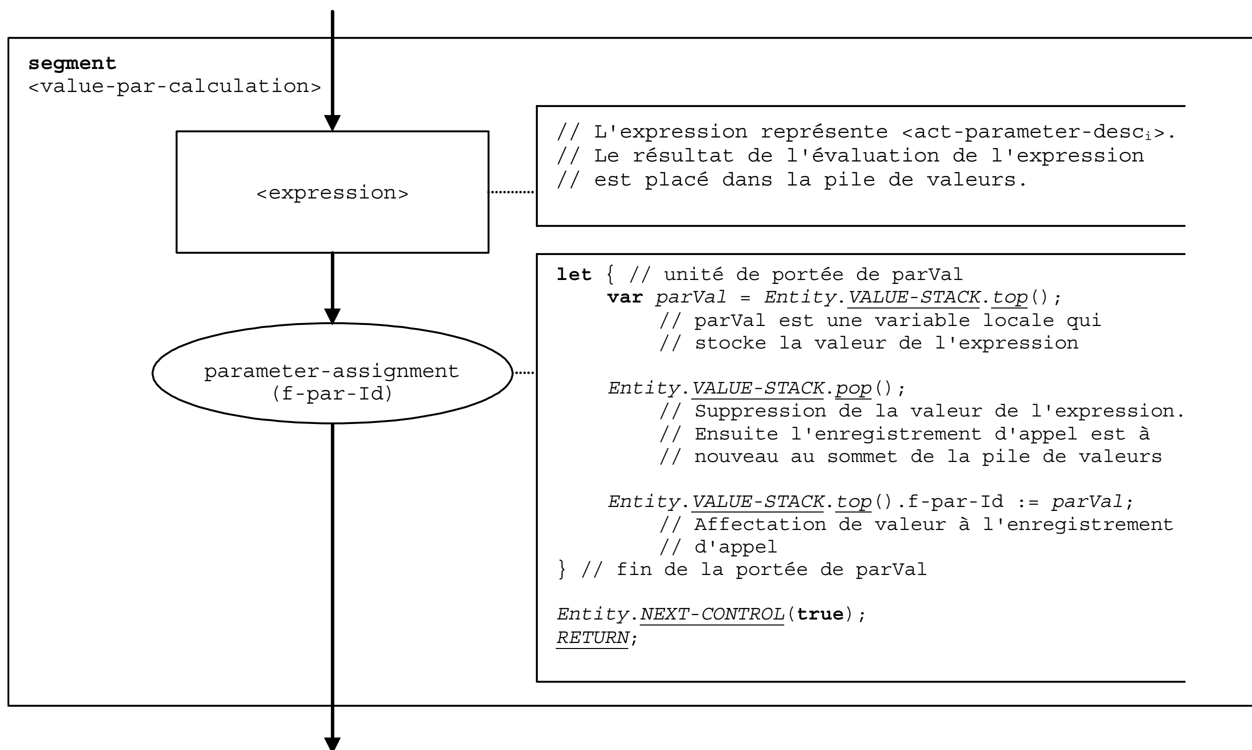


Figure 81/Z.143 – Segment de graphe de flux <value-par-calculation>

9.24.2 Segment de graphe de flux <ref-par-var-calc>

Le segment de graphe de flux `<ref-par-var-calc>` sert à extraire les emplacements des variables utilisées comme paramètres par référence effectifs et à les affecter aux champs correspondants dans les enregistrements d'appel pour les fonctions, les variantes et les tests élémentaires.

On suppose qu'un enregistrement d'appel est l'élément sommital de la pile de valeurs et qu'une paire:

```
(<f-par-Idi>, <act-pari>)
```

doit être traitée. Le paramètre `<act-par`$_i$`>` est le paramètre effectif pour lequel l'emplacement doit être extrait et l'identificateur `<f-par-Id`$_i$`>` est l'identificateur d'un paramètre formel qui a un champ correspondant dans l'enregistrement d'appel figurant dans la pile de valeurs.

L'exécution du segment de graphe de flux `<ref-par-var-calc>` est illustrée sur la Figure 82.

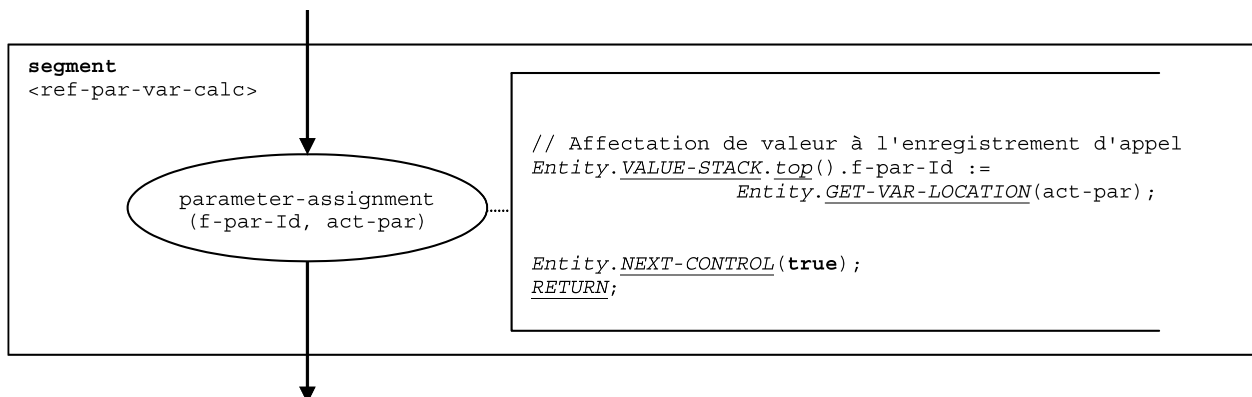


Figure 82/Z.143 – Segment de graphe de flux <ref-par-var-calc>

9.24.3 Segment de graphe de flux <ref-par-timer-calc>

Le segment de graphe de flux <ref-par-timer-calc> sert à extraire les emplacements des temporisations utilisées comme paramètres par référence effectifs et à les affecter aux champs correspondants dans les enregistrements d'appel pour les fonctions, les variantes et les tests élémentaires.

On suppose qu'un enregistrement d'appel est l'élément sommital de la pile de valeurs et qu'une paire:

```
(<f-par-Idi>, <act-pari>)
```

doit être traitée. Le paramètre <act-par$_i$> est le paramètre effectif pour lequel l'emplacement doit être extrait et l'identificateur <f-par-Id$_i$> est l'identificateur d'un paramètre formel qui a un champ correspondant dans l'enregistrement d'appel figurant dans la pile de valeurs.

L'exécution du segment de graphe de flux <ref-par-timer-calc> est illustrée sur la Figure 83.

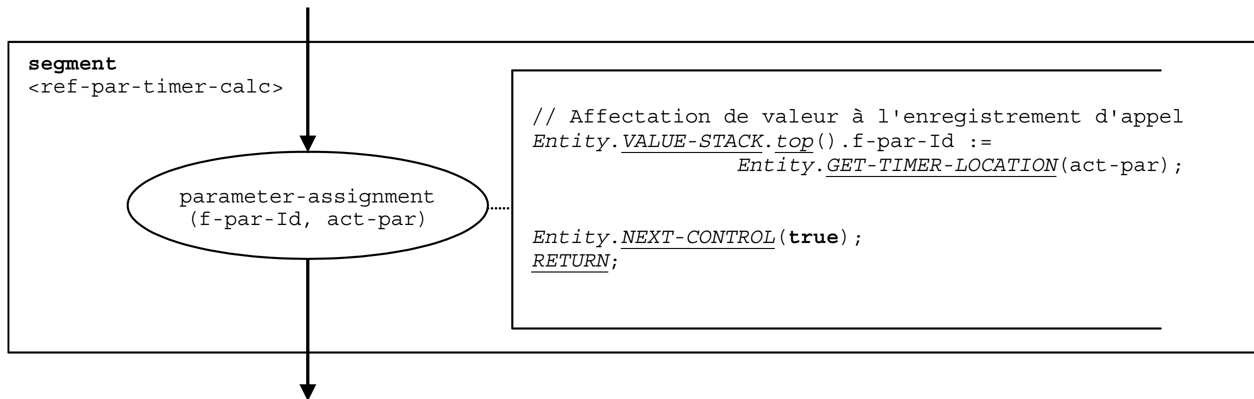


Figure 83/Z.143 – Segment de graphe de flux <ref-par-timer-calc>

9.24.4 Segment de graphe de flux <user-def-func-call>

Le segment de graphe de flux <user-def-func-call> (Figure 84) décrit le transfert de la commande à une fonction définie par l'utilisateur qui est appelée.

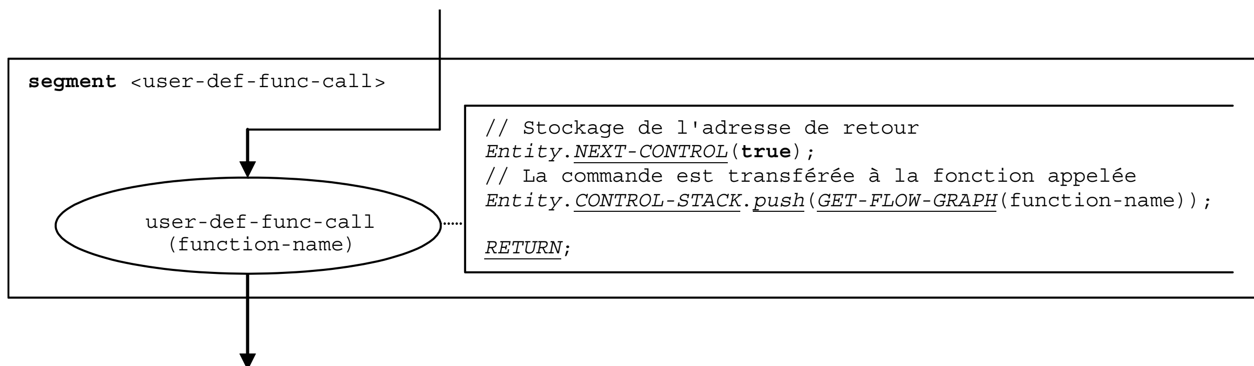


Figure 84/Z.143 – Segment de graphe de flux <user-def-func-call>

9.24.5 Segment de graphe de flux <predef-ext-func-call>

Le segment de graphe de flux <predef-ext-func-call> (Figure 85) décrit l'appel d'une fonction prédéfinie ou externe.

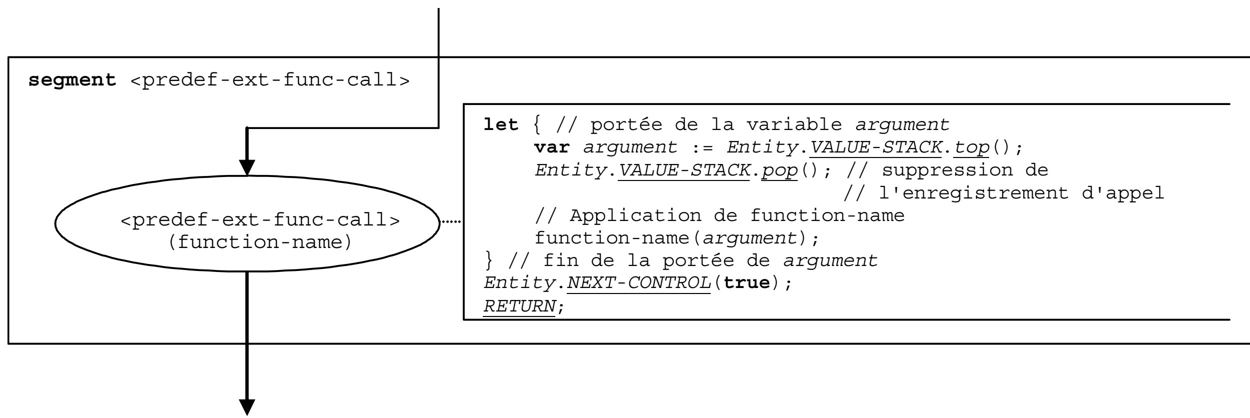


Figure 85/Z.143 – Segment de graphe de flux <predef-ext-func-call>

9.25 Opération getcall

La structure syntaxique de l'opération **getcall** est la suivante:

```
<portId>.getcall (<matchingSpec>) [from <component_expression>] -> [<assignmentPart>]
```

Mis à part le mot clé **`getcall`**, cette structure syntaxique est identique à la structure syntaxique de l'opération **`receive`**. Par conséquent, la sémantique opérationnelle traite l'opération **`getcall`** de la même manière que l'opération **`receive`**, ce qui est illustré dans le segment de graphe de flux `<getcall-op>` (voir Figure 86), qui définit l'exécution d'une opération **`getcall`**. La figure fait référence aux segments de graphe de flux liés à l'opération **`receive`** (voir § 9.37).

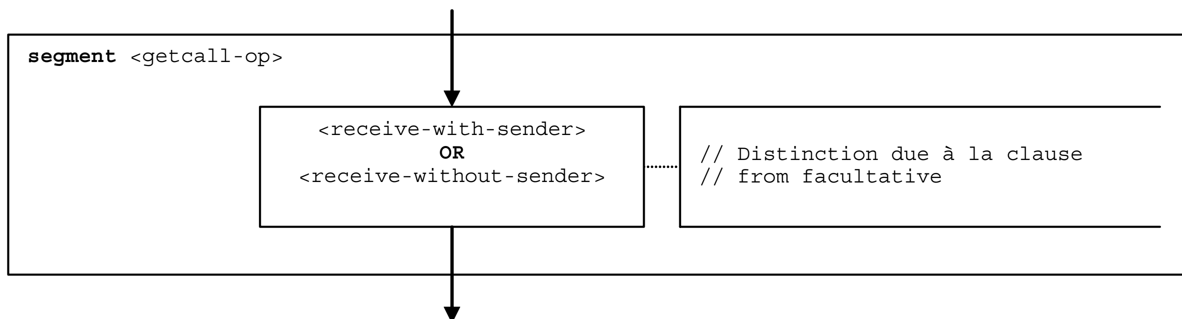


Figure 86/Z.143 – Segment de graphe de flux <getcall-op>

9.26 Opération getreply

La structure syntaxique de l'opération **`getreply`** est la suivante:

```
<portId>.getreply (<matchingSpec>) [from <component-expression>] [-> <assignmentPart>]
```

Mis à part le mot clé **`getreply`**, cette structure syntaxique est identique à la structure syntaxique de l'opération **`receive`**. Par conséquent, la sémantique opérationnelle traite l'opération **`getreply`** de la même manière que l'opération **`receive`**, ce qui est illustré dans le segment de graphe de flux `<getreply-op>` (voir Figure 87), qui définit l'exécution d'une opération **`getreply`**. La figure fait référence aux segments de graphe de flux liés à l'opération **`receive`** (voir § 9.37).

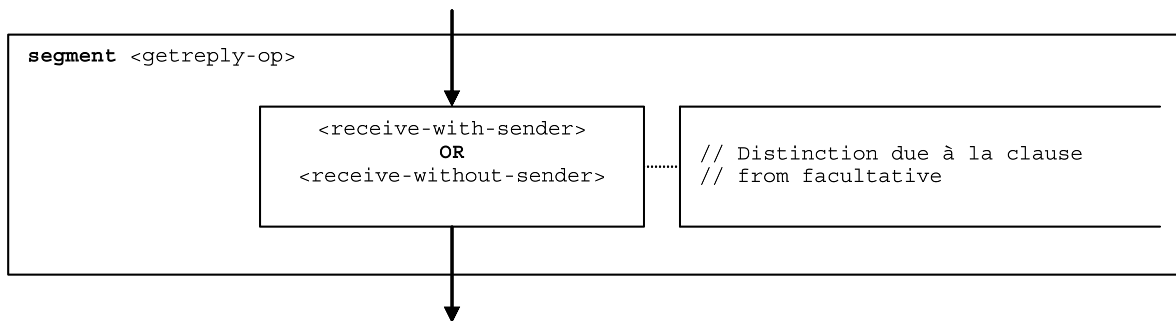


Figure 87/Z.143 – Segment de graphe de flux <getreply-op>

9.27 Opération getverdict

La structure syntaxique de l'opération **getverdict** est la suivante:

```
getverdict
```

Le segment de graphe de flux <getverdict-op> sur la Figure 88 définit l'exécution de l'opération **getverdict**.

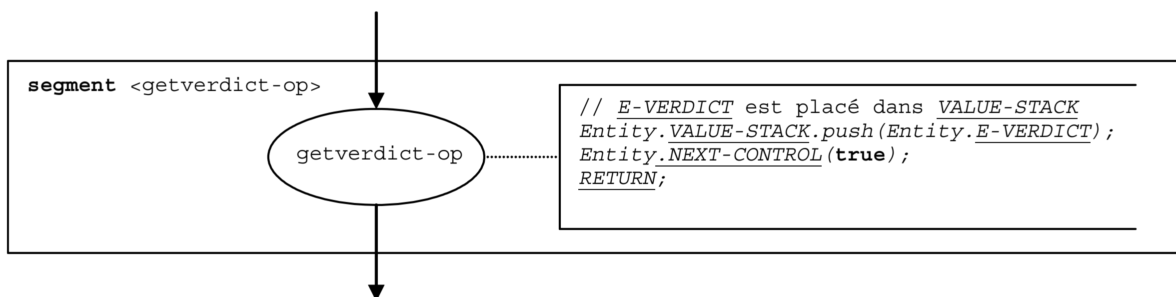


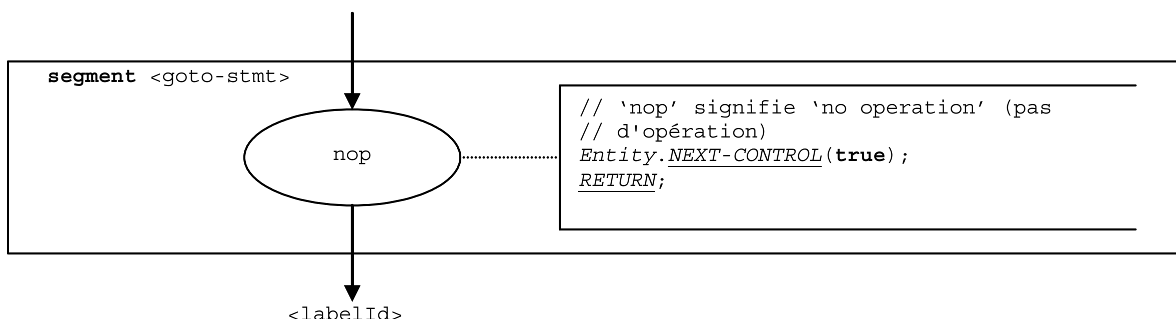
Figure 88/Z.143 – Segment de graphe de flux <getverdict-op>

9.28 Instruction goto

La structure syntaxique de l'instruction **goto** est la suivante:

```
goto <labelId>
```

Le segment de graphe de flux <goto-stmt> sur la Figure 89 définit l'exécution de l'instruction **goto**.



NOTE – Le paramètre <labelId> de l'instruction **goto** indique le transfert de la commande à l'endroit où l'étiquette *<labelId>* est définie (voir aussi § 9.30).

Figure 89/Z.143 – Segment de graphe de flux <goto-stmt>

9.29 Instruction if-else

La structure syntaxique de l'instruction **`if-else`** est la suivante:

```
if (<boolean-expression>) <statement-block1>
    [else <statement-block2>]
```

La partie else de l'instruction **`if-else`** est facultative.

Le segment de graphe de flux `<if-else-stmt>` sur la Figure 90 définit l'exécution de l'instruction **`if-else`**.

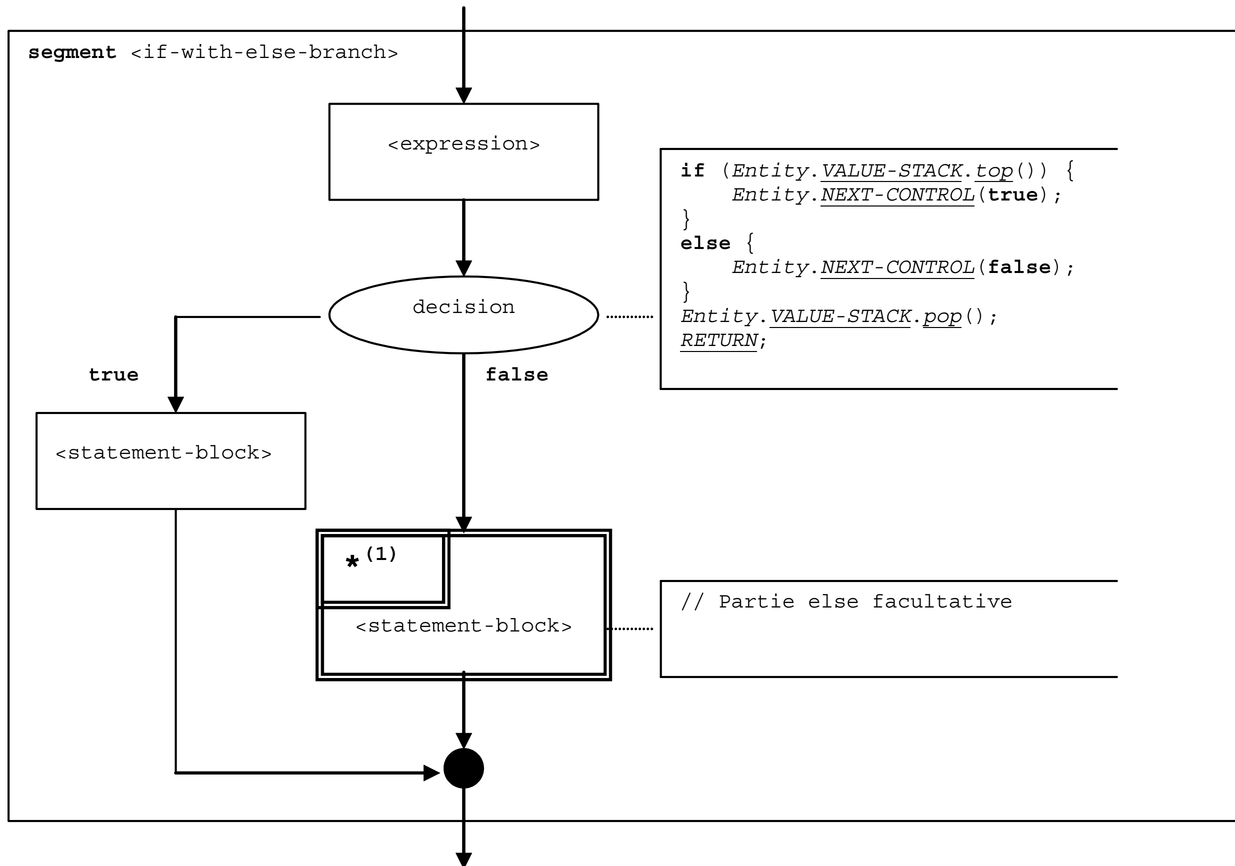


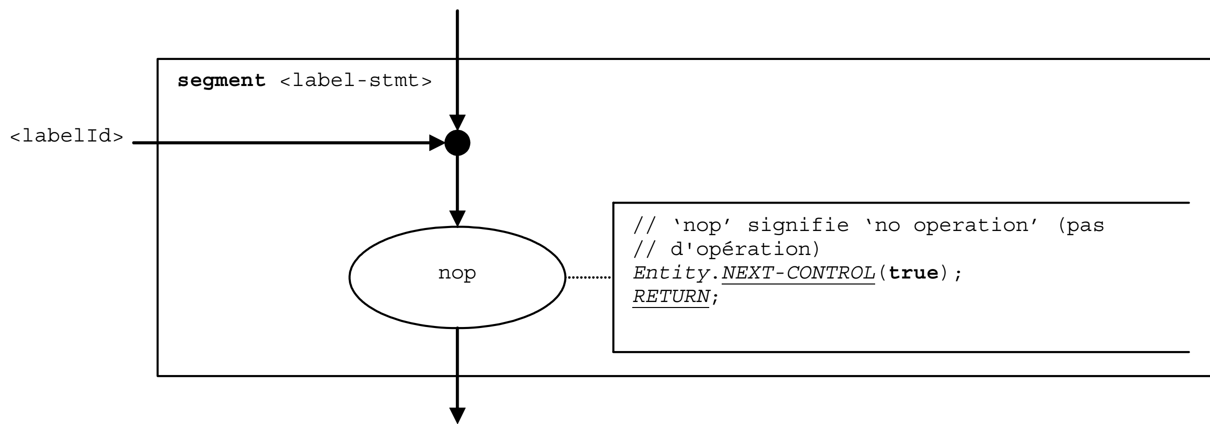
Figure 90/Z.143 – Segment de graphe de flux <if-else-stmt>

9.30 Instruction label

La structure syntaxique de l'instruction **`label`** est la suivante:

```
label <labelId>
```

Le segment de graphe de flux `<label-stmt>` sur la Figure 91 définit l'exécution de l'instruction **`label`**.



NOTE – Le paramètre <labelId> de l'instruction **label** indique qu'une étiquette peut être le point d'aboutissement d'un saut défini par une instruction **goto** (voir aussi § 9.28).

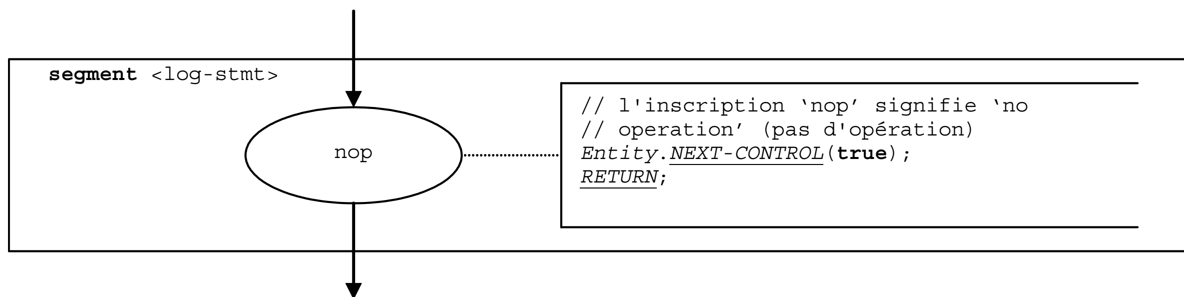
Figure 91/Z.143 – Segment de graphe de flux <label-stmt>

9.31 Instruction log

La structure syntaxique de l'instruction **log** est la suivante:

```
log (<informal-description>)
```

Le segment de graphe de flux <log-stmt> sur la Figure 92 définit l'exécution de l'instruction **log**.



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 de flux.

Figure 92/Z.143 – Segment de graphe de flux <log-stmt>

9.32 Opération map

La structure syntaxique de l'opération **map** est la suivante:

```
map(<component-expression>:<portId1>, system:<portId2>)
```

Les identificateurs <portId1> et <portId2> sont considérés comme étant des identificateurs de port du composant de test et de l'interface du système de test correspondants. Il est fait référence au composant associé à l'identificateur <portId1> au moyen de la référence de composant <component-expression>. La référence peut être stockée dans une variable ou retournée par une fonction, autrement dit il s'agit d'une expression dont l'évaluation donne une référence de composant. La pile de valeurs est utilisée pour le stockage de la référence de composant.

NOTE – Pour l'opération **map**, peu importe si l'instruction **system**:<portId> apparaît comme le premier ou le deuxième paramètre. Dans un souci de simplicité, on suppose qu'il s'agit toujours du deuxième paramètre.

L'exécution de l'opération **map** est définie par le segment de graphe de flux <map-op> illustré sur la Figure 93.

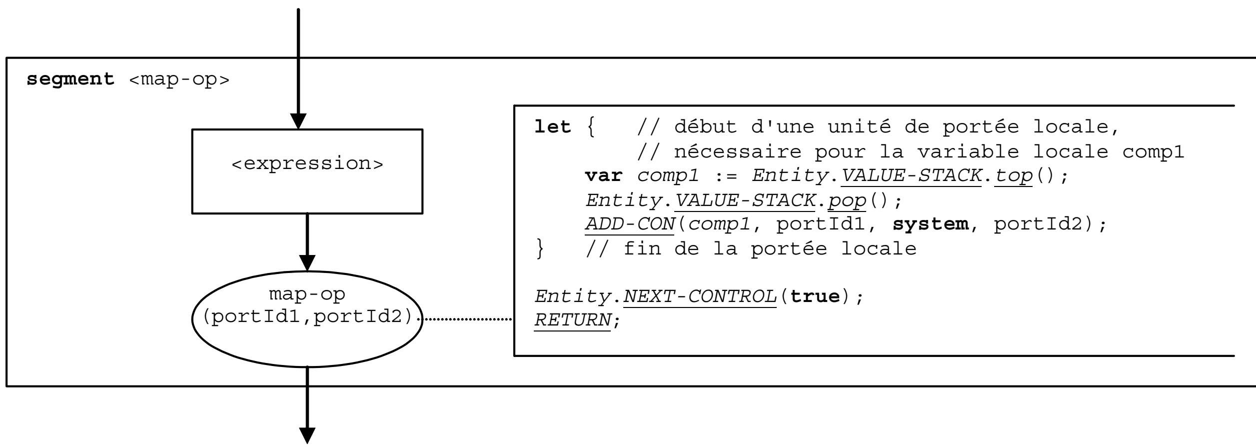


Figure 93/Z.143 – Segment de graphe de flux <map-op>

9.33 Opération mtc

La structure syntaxique de l'opération **`mtc`** est la suivante:

```
mtc
```

Le segment de graphe de flux `<mtc-op>` sur la Figure 94 définit l'exécution de l'opération **`mtc`**.

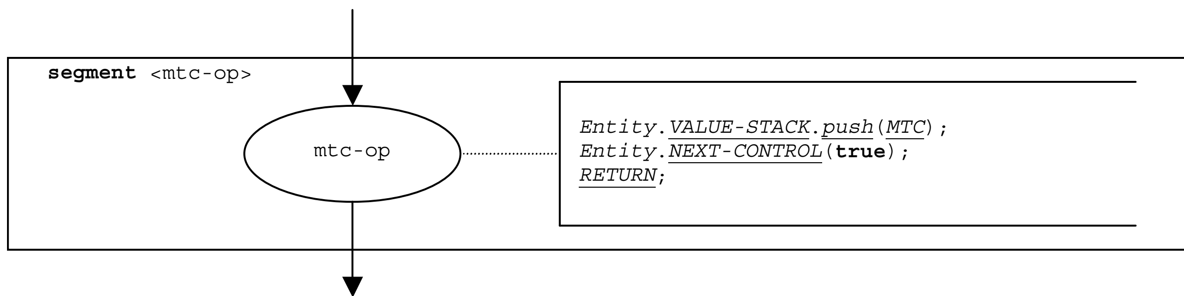


Figure 94/Z.143 – Segment de graphe de flux <mtc-op>

9.34 Déclaration de port

La structure syntaxique d'une déclaration de **`port`** est la suivante:

```
<portType> <portName>
```

Les déclarations de port peuvent figurer dans des définitions de type de composant. Une déclaration de port a pour effet de créer un nouveau port lorsqu'un nouveau composant du type correspondant est créé. Le segment de graphe de flux `<port-declaration>` sur la Figure 95 définit l'exécution d'une déclaration de port.

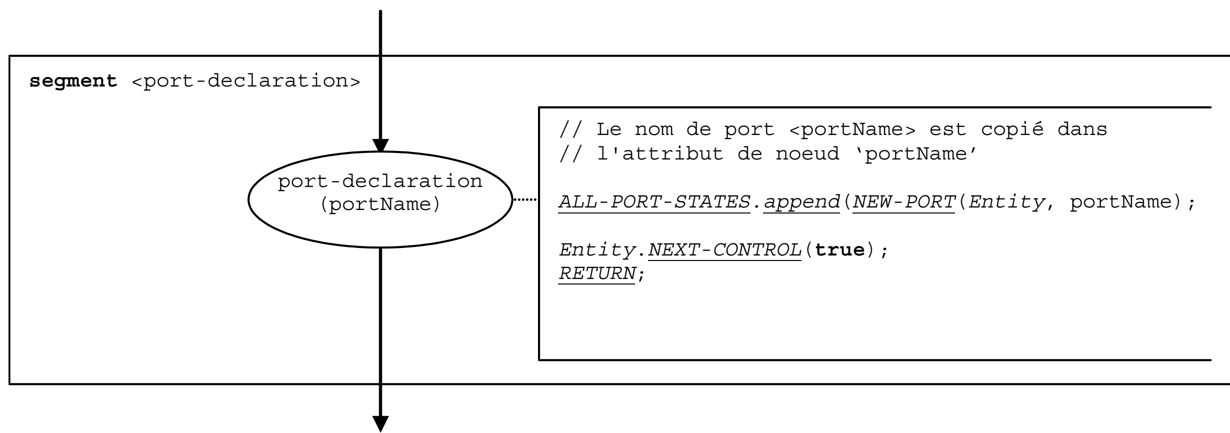


Figure 95/Z.143 – Segment de graphe de flux <port-declaration>

9.35 Opération raise

La structure syntaxique de l'opération **`raise`** est la suivante:

```
<portId>.raise (<exceptSpec>) [to <component-expression>]
```

La partie facultative `<component-expression>` dans la clause **`to`** indique l'entité réceptrice. Il peut s'agir d'une valeur de variable ou de la valeur de retour d'une fonction.

Le segment de graphe de flux `<raise-op>` sur la Figure 96 définit l'exécution d'une opération **`raise`**.

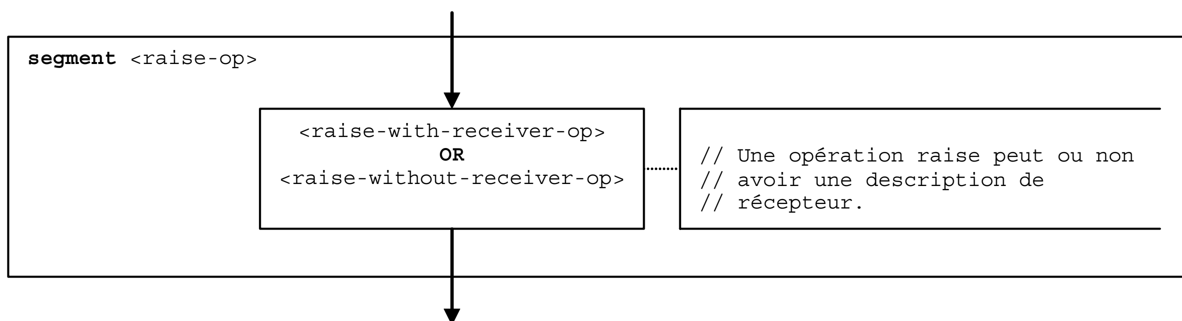


Figure 96/Z.143 – Segment de graphe de flux <raise-op>

9.35.1 Segment de graphe de flux <raise-with-receiver-op>

Le segment de graphe de flux `<raise-with-receiver-op>` sur la Figure 97 définit l'exécution d'une opération **`raise`** pour laquelle le récepteur est spécifié sous la forme d'une expression.

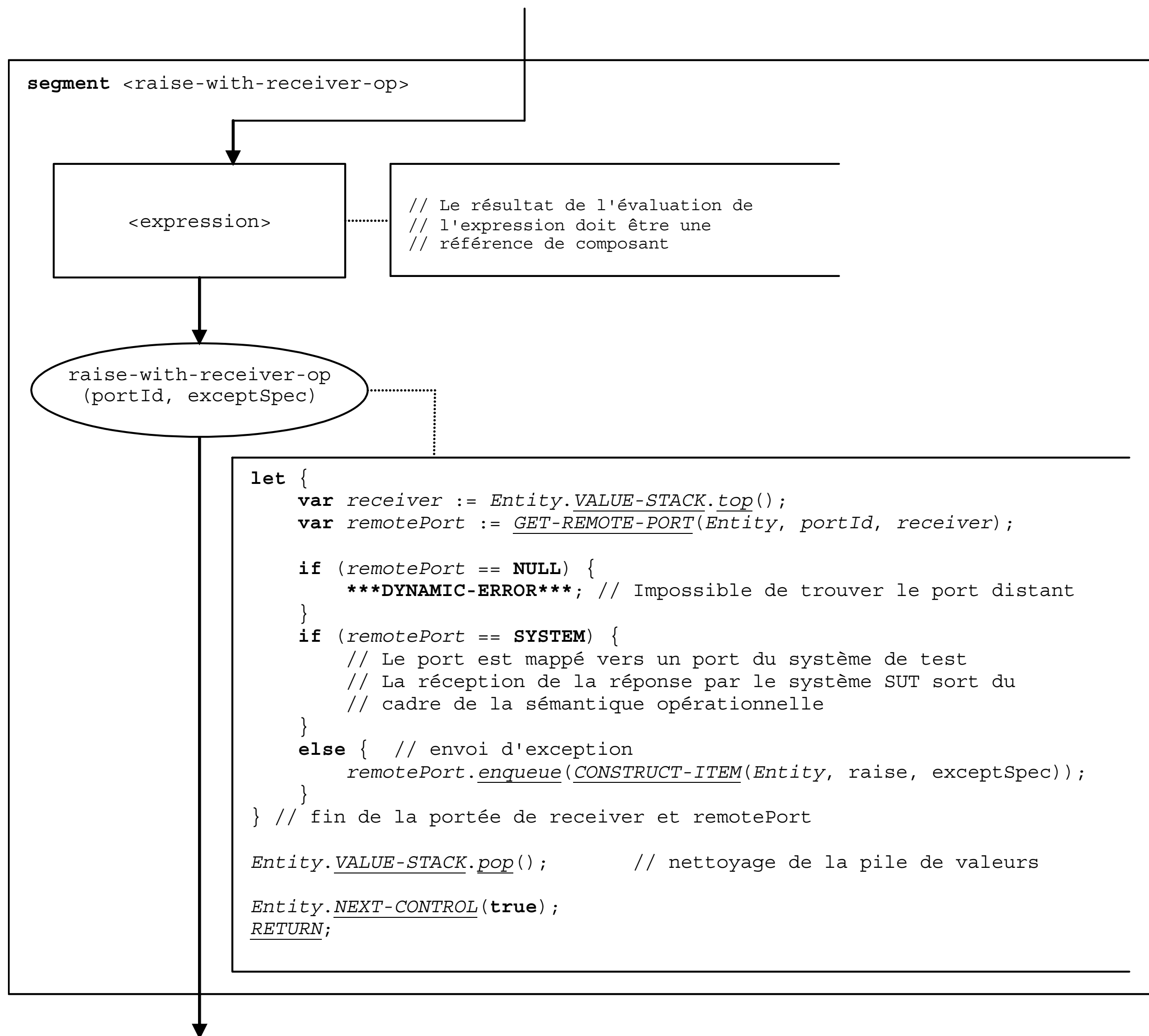


Figure 97/Z.143 – Segment de graphe de flux <raise-with-receiver-op>

9.35.2 Segment de graphe de flux <raise-without-receiver-op>

Le segment de graphe de flux `<raise-without-receiver-op>` sur la Figure 98 définit l'exécution d'une opération **`raise`** sans clause **`to`**.

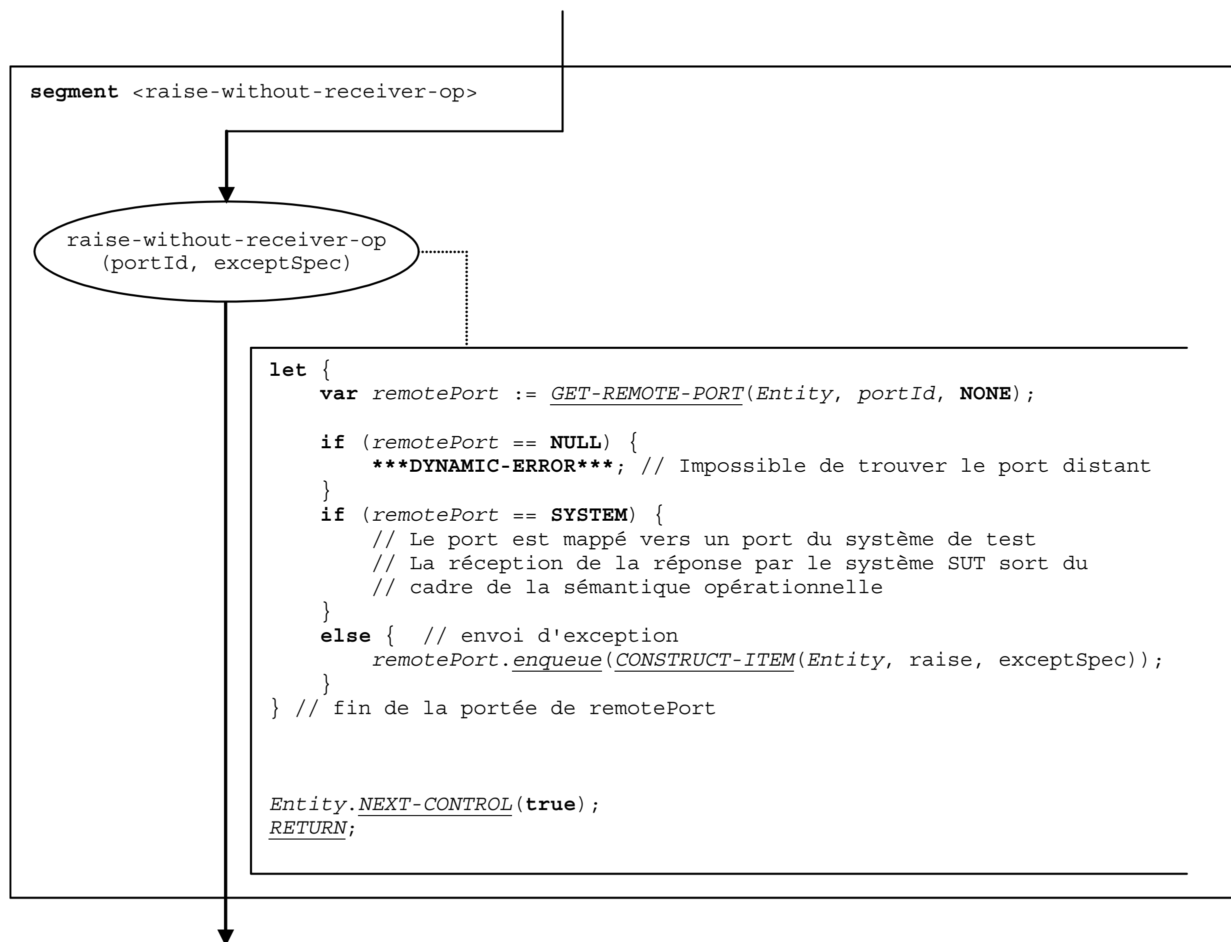


Figure 98/Z.143 – Segment de graphe de flux <raise-without-receiver-op>

9.36 Opération read applicable aux temporisations

La structure syntaxique de l'opération **read** applicable aux temporisations est la suivante:

```
<timerId>.read
```

Le segment de graphe de flux <read-timer-op> sur la Figure 99 définit l'exécution de l'opération **read** applicable aux temporisations.

En ce qui concerne l'opération **read** applicable aux temporisations, on fait la distinction entre, d'une part, son utilisation dans une protection booléenne d'une instruction **alt** ou d'une opération **call** bloquante et, d'autre part, tous les autres cas. Si cette opération est utilisée dans une protection booléenne, son résultat est fondé sur l'instantané effectif, autrement dit les entrées *SNAP-STATUS* et *SNAP-VALUE* du lien de temporisation; dans tous les autres cas, les entrées *STATUS*, *ACT-DURATION* et *TIME-LEFT* du lien de temporisation déterminent le résultat de l'opération.

```
segment <read-timer-op>

read-timer-op
(timerId)

let { // portée locale de la variable myValue

    var float myValue;

    if (Entity.STATUS == SNAPSHOT) {
        if (Entity.timerId.SNAP-STATUS == RUNNING) {
            myValue := Entity.timerId.SNAP-VALUE;
        }
        else {
            myValue := 0.0;
        }
    }
    else {
        if (Entity.timerId.STATUS == RUNNING) {
            myValue := Entity.timerId.ACT-DURATION - Entity.timerId.TIME-LEFT;

        }
        else {
            myValue := 0.0;
        }
    }

    Entity.VALUE-STACK.push(myValue);

} // fin de la portée locale

Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 99/Z.143 – Segment de graphe de flux <read-timer-op>

9.37 Opération receive

La structure syntaxique de l'opération **receive** est la suivante:

```
<portId>.receive (<matchingSpec>) [from <component-expression>] [-> <assignmentPart>]
```

La partie facultative `<component-expression>` dans la clause **from** indique l'entité expéditrice. Il peut s'agir d'une valeur de variable ou de la valeur de retour d'une fonction, autrement dit on suppose qu'il s'agit d'une expression. La partie facultative `<assignmentPart>` indique l'affectation des informations reçues si le message reçu correspond à la spécification d'appariement `<matchingSpec>` et à la clause (facultative) **from**.

Le segment de graphe de flux `<receive-op>` sur la Figure 100 définit l'exécution d'une opération **receive**.

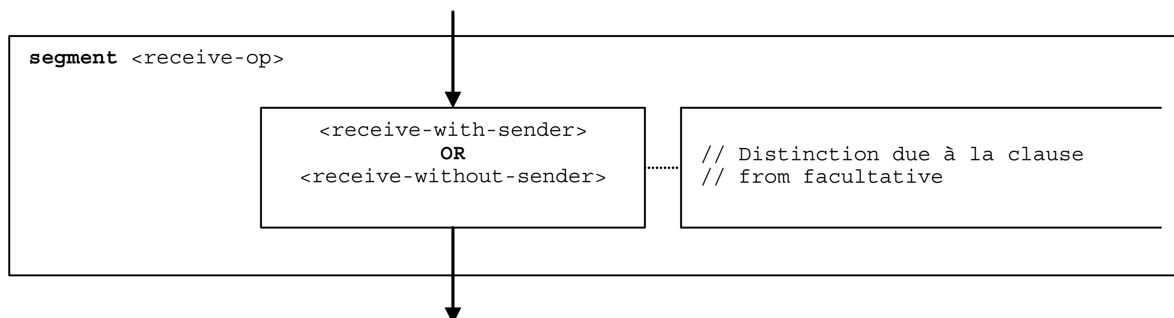


Figure 100/Z.143 – Segment de graphe de flux <receive-op>

9.37.1 Segment de graphe de flux <receive-with-sender>

Le segment de graphe de flux `<receive-with-sender>` sur la Figure 101 définit l'exécution d'une opération **`receive`** pour laquelle l'expéditeur est spécifié sous la forme d'une expression.

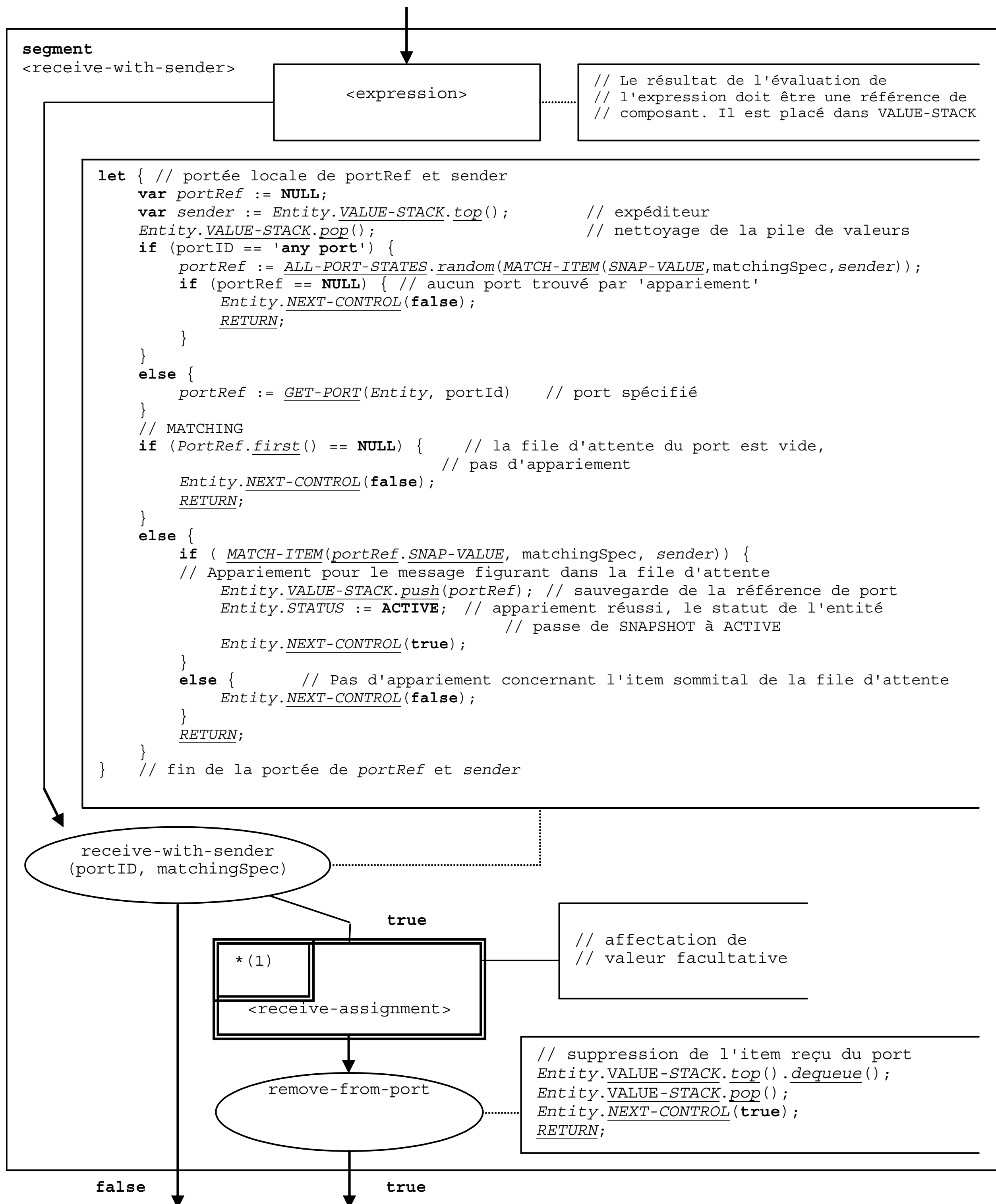


Figure 101/Z.143 – Segment de graphe de flux <receive-with-sender>

9.37.2 Segment de graphe de flux <receive-without-sender>

Le segment de graphe de flux `<receive-without-sender>` sur la Figure 102 définit l'exécution d'une opération **`receive`** sans clause **`from`**.

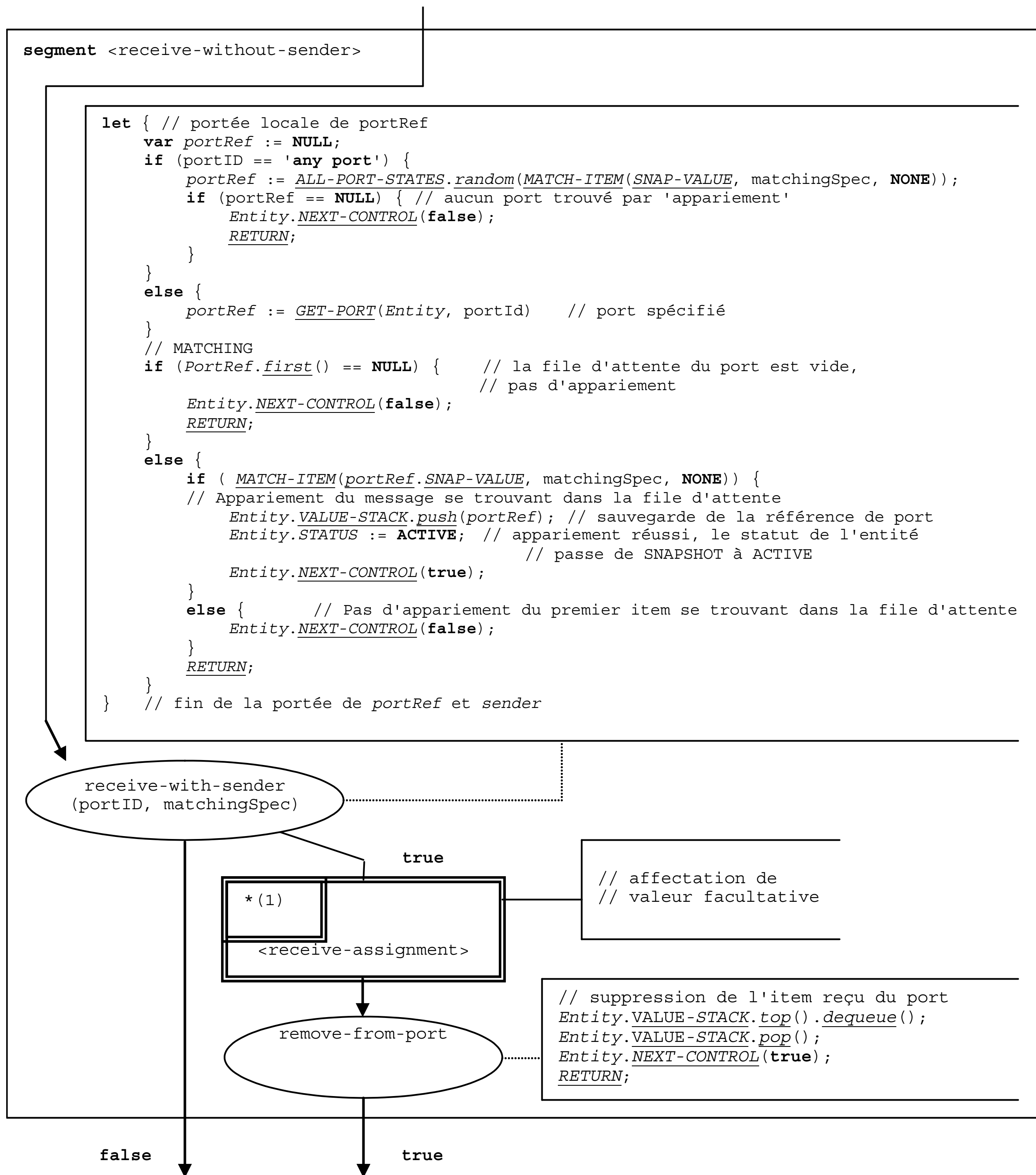


Figure 102/Z.143 – Segment de graphe de flux <receive-without-sender>

9.37.3 Segment de graphe de flux <receive-assignment>

Le segment de graphe de flux `<receive-assignment>` sur la Figure 103 définit l'extraction d'informations des messages reçus et leur affectation à des variables.

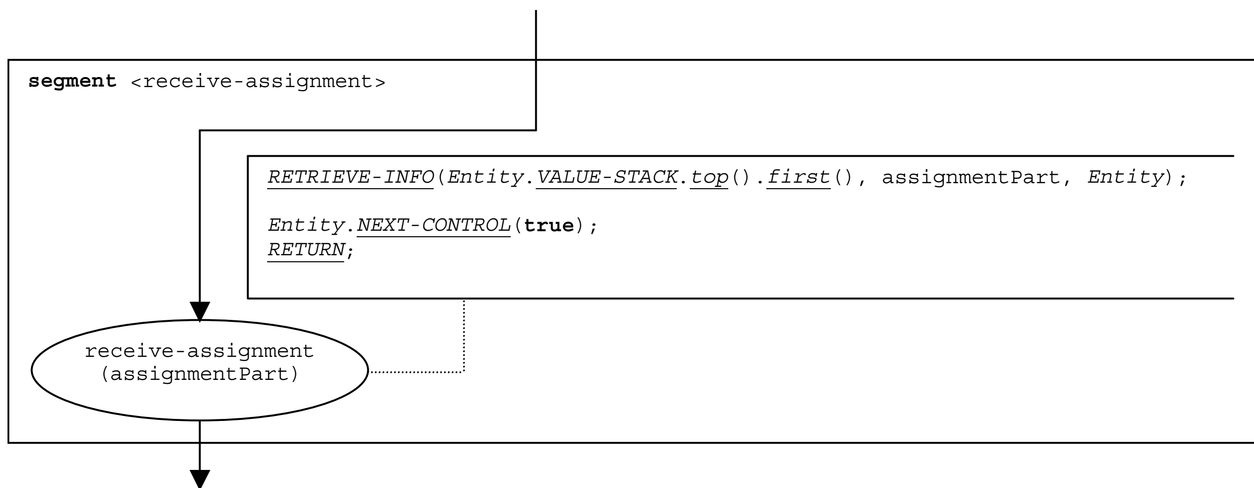


Figure 103/Z.143 – Segment de graphe de flux <receive-assignment>

9.38 Instruction repeat

La structure syntaxique de l'instruction **repeat** est la suivante:

```
repeat
```

Fondamentalement, l'instruction **repeat** est une instruction **return** sans valeur de retour et qui fait passer le statut de l'entité à **REPEAT**. Le statut **REPEAT** imposera la réévaluation de l'instruction **alt** dans laquelle l'instruction **repeat** a été exécutée. Le segment de graphe de flux <repeat-stmt> sur la Figure 104 définit l'exécution de l'instruction **repeat**.

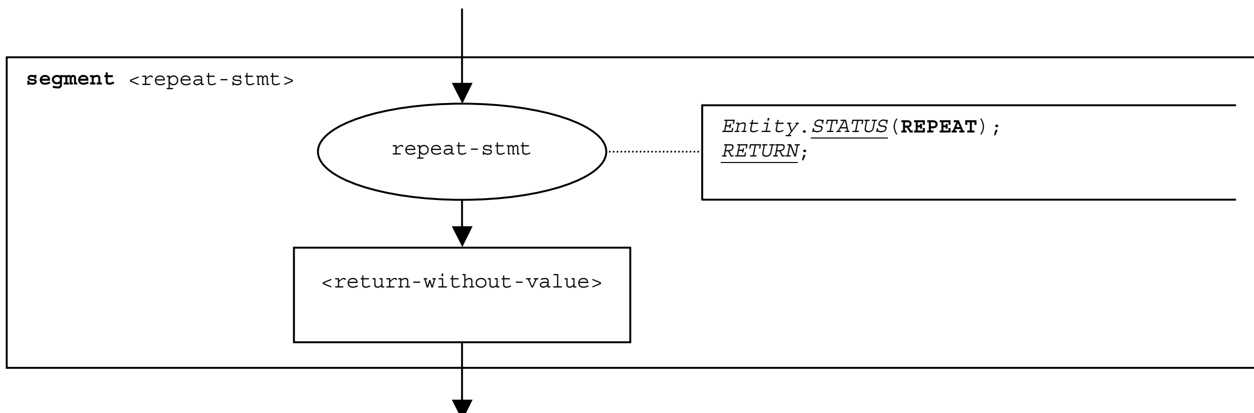


Figure 104/Z.143 – Segment de graphe de flux <repeat-stmt>

9.39 Opération reply

La structure syntaxique de l'opération **reply** est la suivante:

```
<portId>.reply (<replySpec>) [to <component-expression>]
```

La partie facultative <component_expression> dans la clause **to** indique l'entité réceptrice. Il peut s'agir d'une valeur de variable ou de la valeur de retour d'une fonction.

Le segment de graphe de flux <reply-op> sur la Figure 105 définit l'exécution d'une opération **reply**.

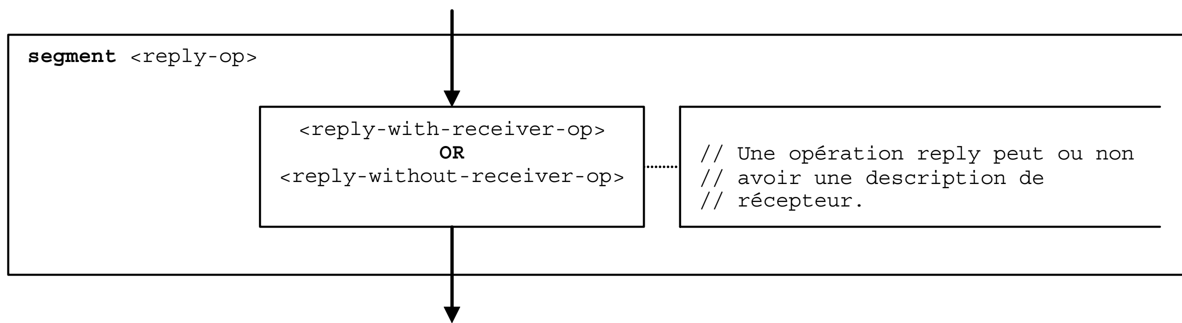


Figure 105/Z.143 – Segment de graphe de flux <reply-op>

9.39.1 Segment de graphe de flux <reply-with-receiver-op>

Le segment de graphe de flux `<reply-with-receiver-op>` sur la Figure 106 définit l'exécution d'une opération **`reply`** pour laquelle le récepteur est spécifié sous la forme d'une expression.

```
segment <reply-with-receiver-op>

<expression>
// Le résultat de l'évaluation de
// l'expression doit être une
// référence de composant

reply-with-receiver-op
(portId, replySpec)

let {
    var receiver := Entity.VALUE-STACK.top();
    var remotePort := GET-REMOTE-PORT(Entity, portId, receiver);

    if (remotePort == NULL) {
        ***DYNAMIC-ERROR***; // Impossible de trouver le port distant
    }
    if (remotePort == SYSTEM) {
        // Le port est mappé vers un port du système de test
        // La réception de la réponse par le système SUT sort du
        // cadre de la sémantique opérationnelle
    }
    else {  // envoi de la réponse
        remotePort.enqueue(CONSTRUCT-ITEM(Entity, reply, replySpec));
    }
} // fin de la portée de receiver et remotePort

Entity.VALUE-STACK.pop();       // nettoyage de la pile de valeurs

Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 106/Z.143 – Segment de graphe de flux <reply-with-receiver-op>

9.39.2 Segment de graphe de flux <reply-without-receiver-op>

Le segment de graphe de flux `<reply-without-receiver-op>` sur la Figure 107 définit l'exécution d'une opération **`reply`** sans clause **`to`**.

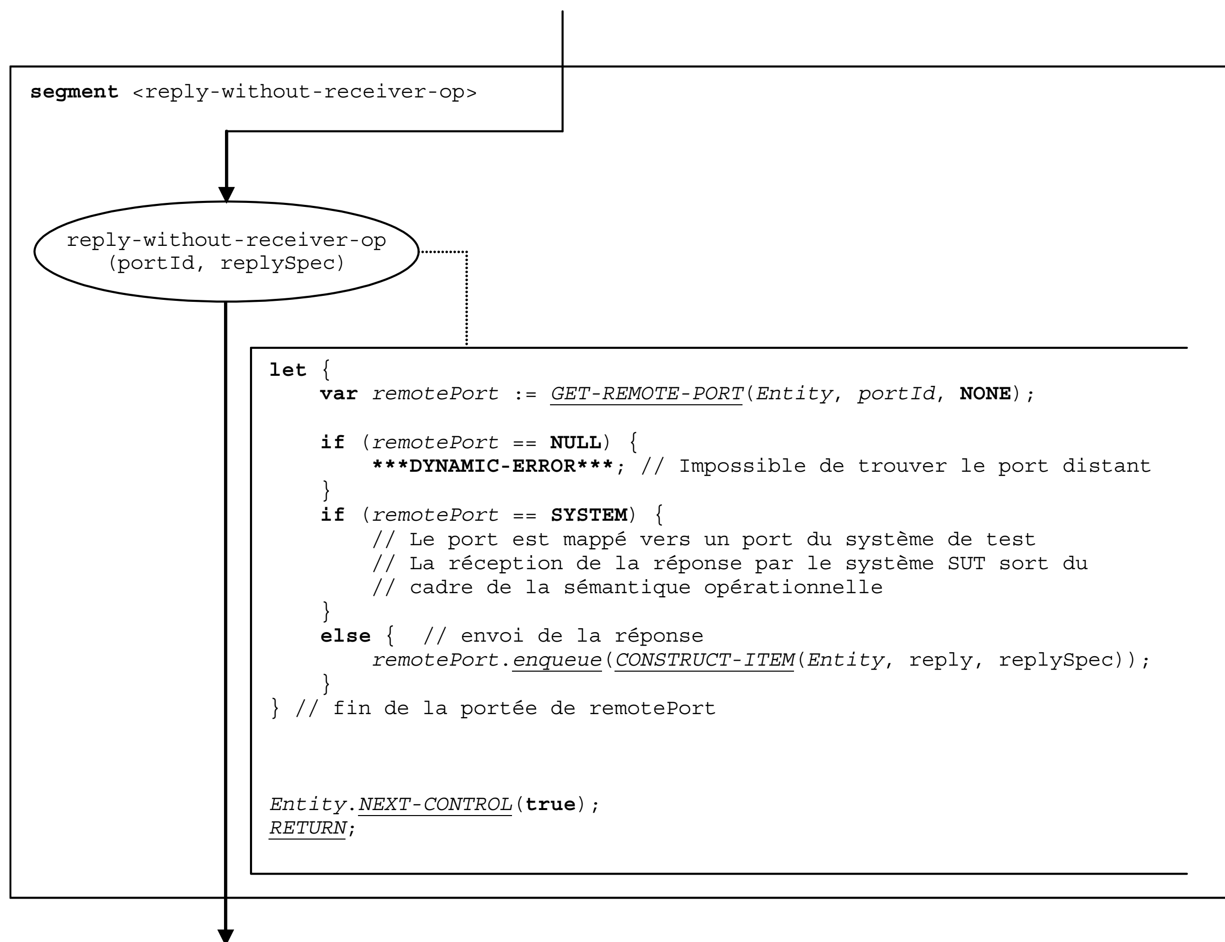


Figure 107/Z.143 – Segment de graphe de flux <reply-without-receiver-op>

9.40 Instruction return

La structure syntaxique de l'instruction **`return`** est la suivante:

```
return [<expression>]
```

La partie facultative `<expression>` décrit une valeur de retour possible. L'exécution d'une instruction **`return`** signifie que la commande quitte l'unité de portée effective, autrement dit les variables et les temporisations qui ne sont connues que dans cette portée doivent être supprimées et la pile de valeurs doit être mise à jour. Une instruction **`return`** a l'effet d'une opération **`stop`** applicable aux composants s'il s'agit de la dernière instruction figurant dans une description de comportement.

NOTE – Les tests élémentaires et la commande de module se termineront toujours au moyen d'une opération **`stop`** applicable aux composants, du fait de leur représentation sous forme de graphe de flux (voir § 8.2). Seuls les autres composants de test peuvent se terminer au moyen d'une instruction **`return`**.

Le segment de graphe de flux `<return-stmt>` sur la Figure 108 définit l'exécution d'une instruction **`return`**.

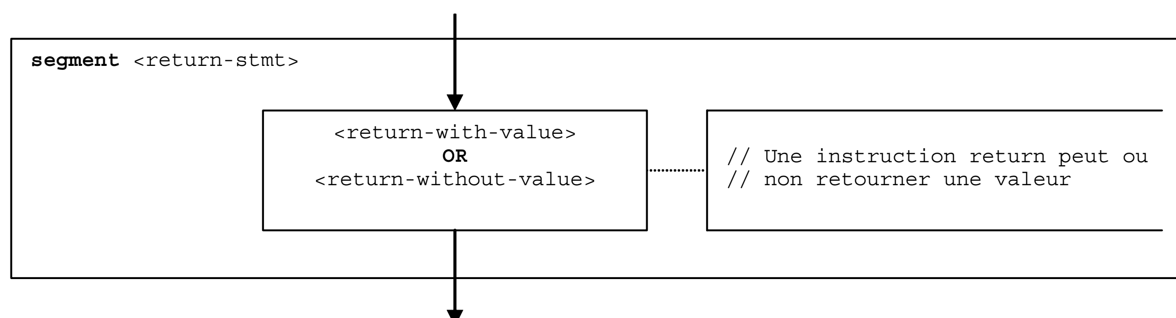


Figure 108/Z.143 – Segment de graphe de flux <return-stmt>

9.40.1 Segment de graphe de flux <return-with-value>

Le segment de graphe de flux `<return-with-value>` sur la Figure 109 définit l'exécution d'une instruction **`return`** qui retourne une valeur spécifiée sous la forme d'une expression.

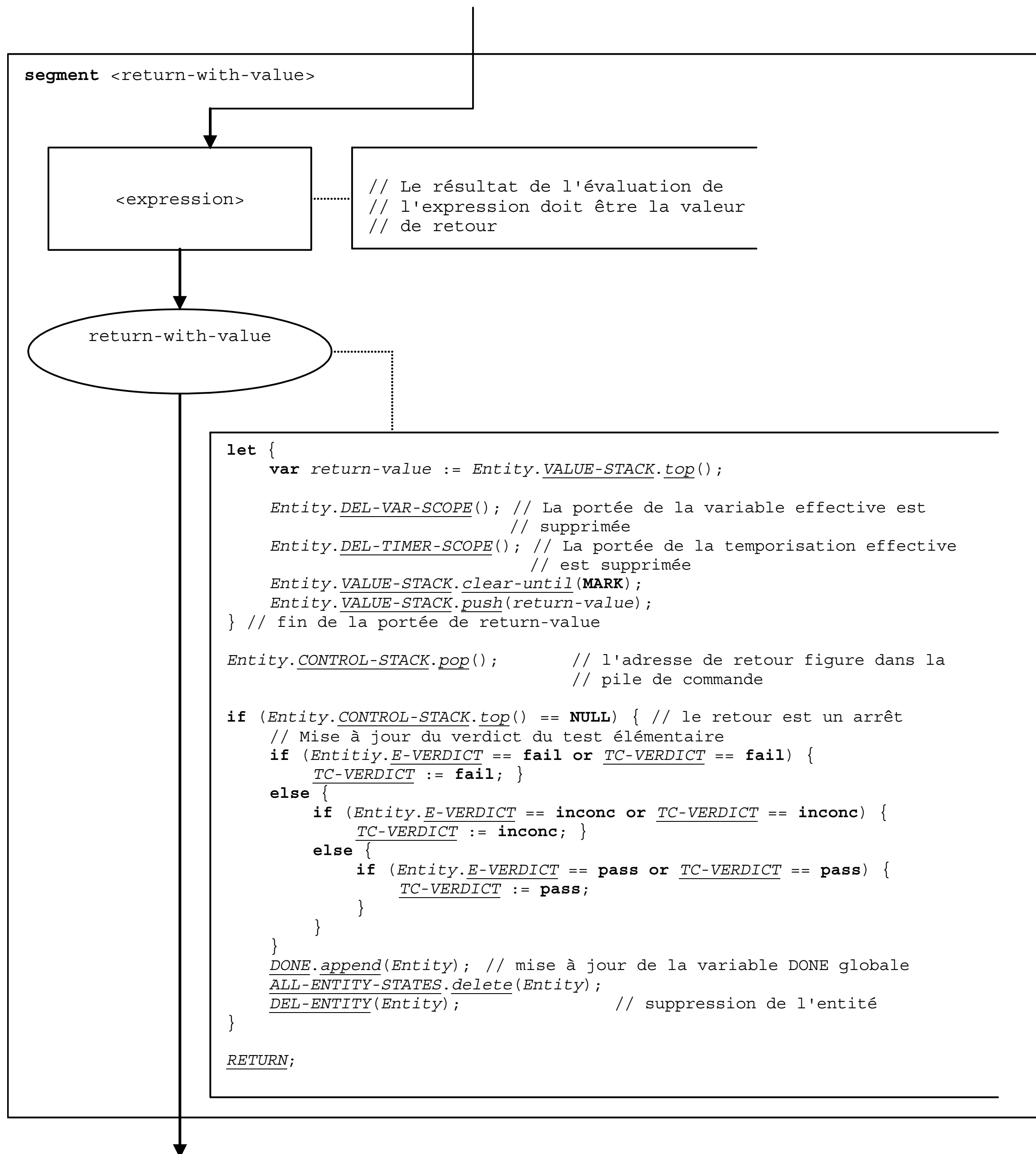


Figure 109/Z.143 – Segment de graphe de flux <return-with-value>

9.40.2 Segment de graphe de flux <return-without-value>

Le segment de graphe de flux `<return-without-value>` sur la Figure 110 définit l'exécution d'une instruction **`return`** qui ne retourne pas de valeur.

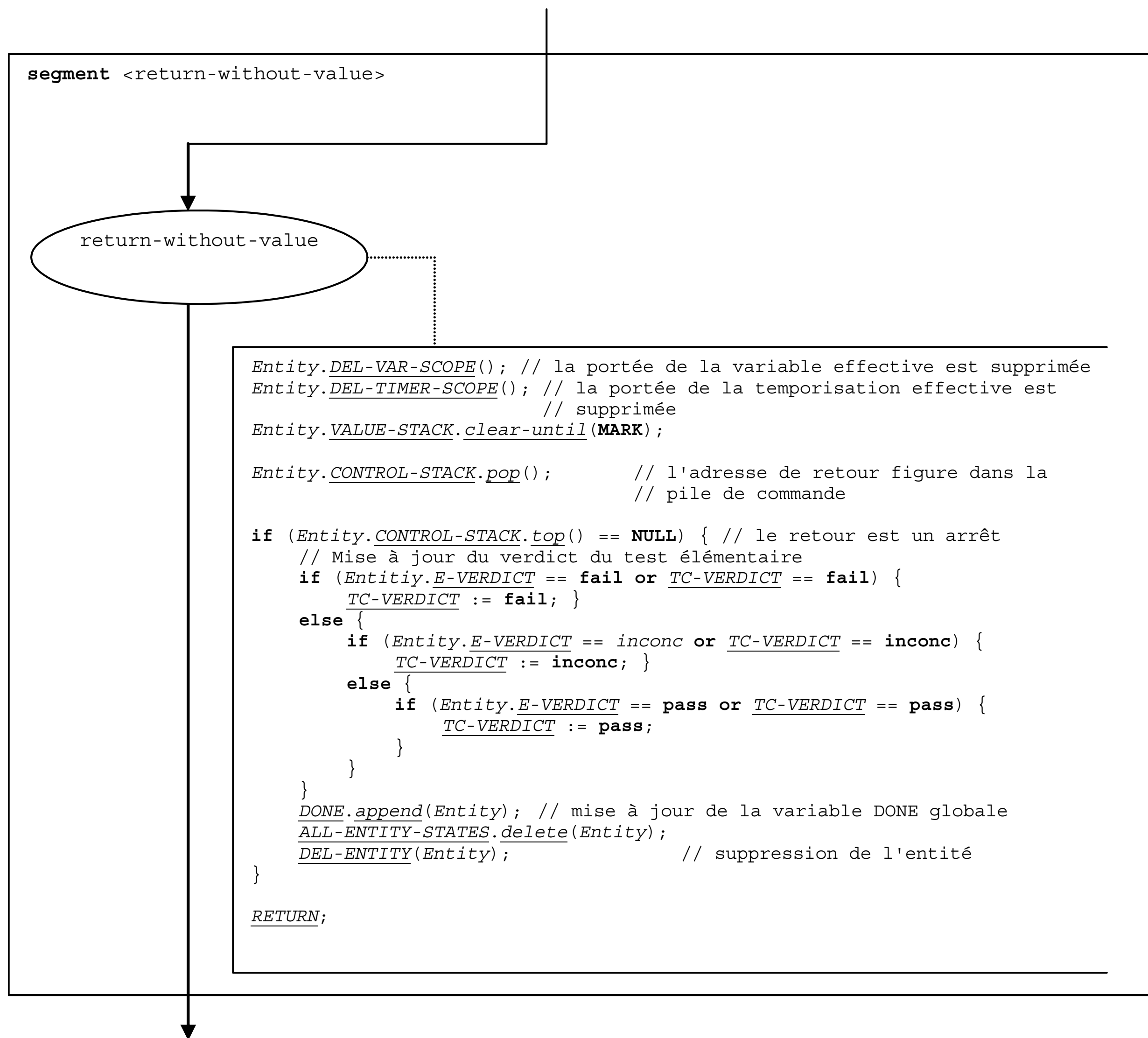


Figure 110/Z.143 – Segment de graphe de flux <return-without-value>

9.41 Opération running applicable aux composants

La structure syntaxique de l'opération **running** applicable aux composants est la suivante:

```
<component-expression>.running
```

L'opération **running** applicable aux composants permet de vérifier si un composant est en cours ou a été arrêté. Le composant à vérifier est identifié par une référence de composant, qui peut être donnée sous la forme d'une variable ou d'une fonction retournant une valeur, autrement dit il s'agit d'une expression. Dans un souci de simplicité, les mots clés '**all component**' et '**any component**' sont considérés comme des expressions spéciales.

En ce qui concerne l'opération **running** applicable aux composants, on fait la distinction entre, d'une part, son utilisation dans une protection booléenne d'une instruction **alt** ou d'une opération **call** bloquante et, d'autre part, tous les autres cas. Si cette opération est utilisée dans une protection booléenne, son résultat est fondé sur l'instantané effectif; dans tous les autres cas, son résultat correspond directement aux informations d'état.

Le résultat de l'opération **running** applicable aux composants est placé dans la pile de valeurs de l'entité qui a appelé l'opération.

Le segment de graphe de flux <running-component-op> sur la Figure 111 définit l'exécution de l'opération **running** applicable aux composants.

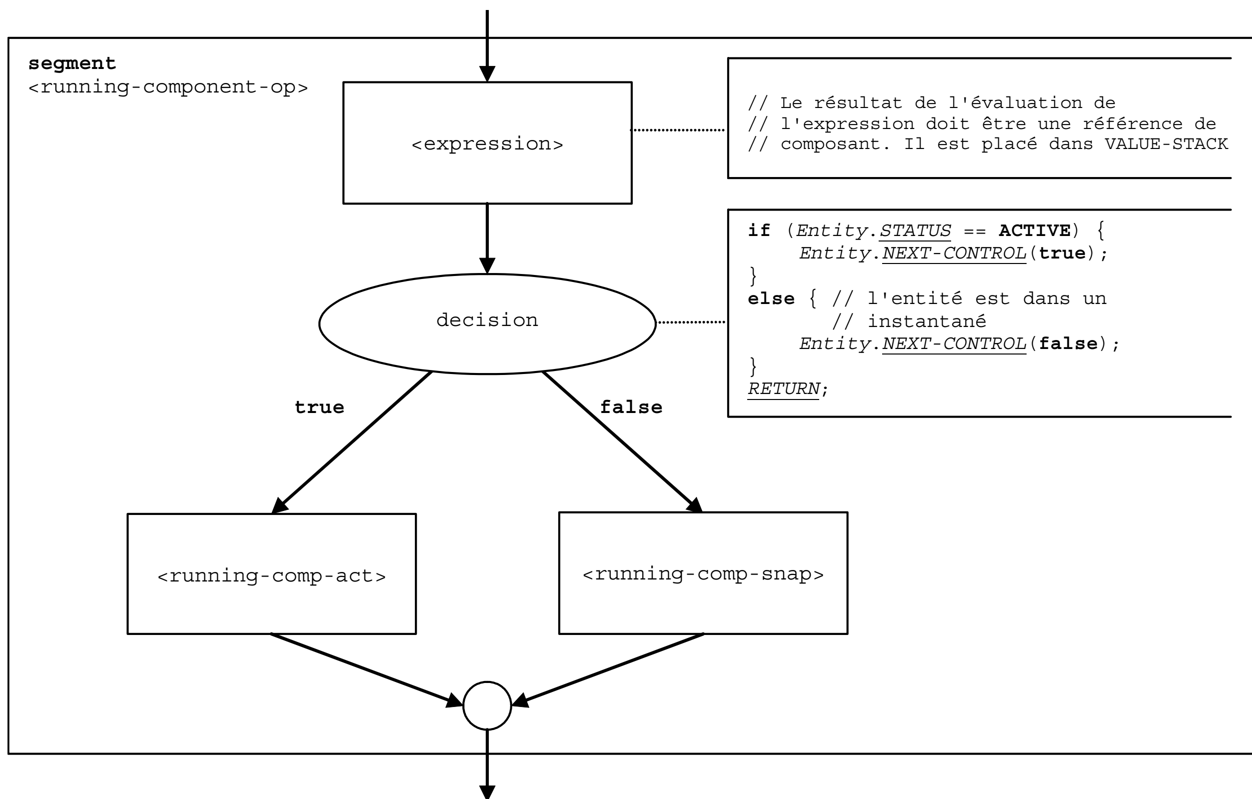


Figure 111/Z.143 – Segment de graphe de flux <running-component-op>

9.41.1 Segment de graphe de flux <running-comp-act>

Le segment de graphe de flux `<running-comp-act>` sur la Figure 112 décrit l'exécution de l'opération **`running`** applicable aux composants en dehors d'un instantané, autrement dit lorsque l'entité a le statut **`ACTIVE`**.

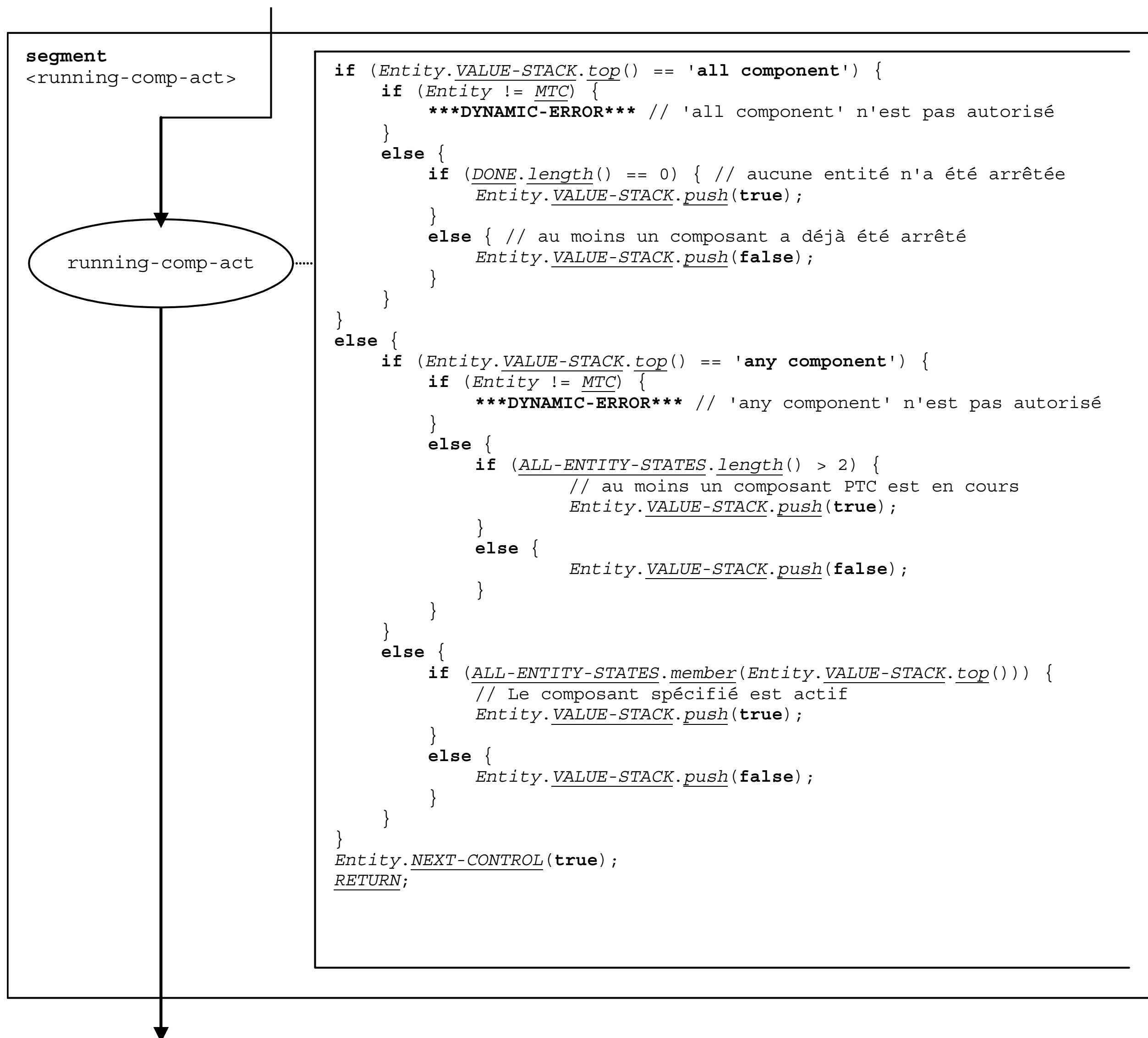


Figure 112/Z.143 – Segment de graphe de flux <running-comp-act>

9.41.2 Segment de graphe de flux <running-comp-snap>

Le segment de graphe de flux `<running-comp-snap>` sur la Figure 113 décrit l'exécution de l'opération **`running`** applicable aux composants pendant l'évaluation d'un instantané, autrement dit lorsque l'entité a le statut **`SNAPSHOT`**.

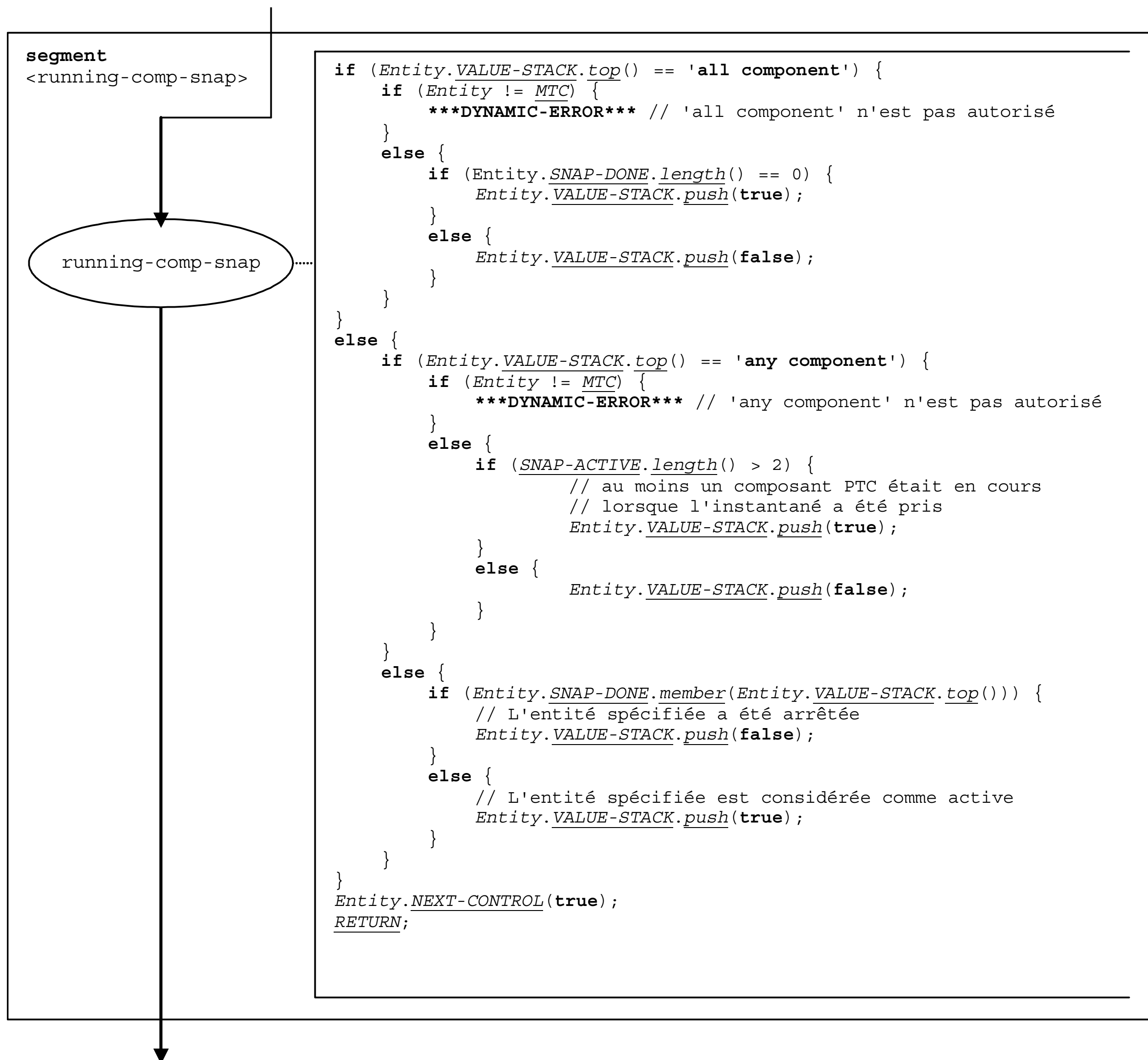


Figure 113/Z.143 – Segment de graphe de flux <running-comp-snap>

9.42 Opération running applicable aux temporisations

La structure syntaxique de l'opération **running** applicable aux temporisations est la suivante:

```
<timerId>.running
```

Le segment de graphe de flux <running-timer-op> sur la Figure 114 définit l'exécution de l'opération **running** applicable aux temporisations.

En ce qui concerne l'opération **running** applicable aux temporisations, on fait la distinction entre, d'une part, son utilisation dans une protection booléenne d'une instruction **alt** ou d'une opération **call** bloquante et, d'autre part, tous les autres cas. Si cette opération est utilisée dans une protection booléenne, son résultat est fondé sur l'instantané effectif, autrement dit l'entrée *SNAP-STATUS* du lien de temporisation; dans tous les autres cas, l'entrée *STATUS* du lien de temporisation détermine le résultat de l'opération.

Le mot clé **any** est traité comme une valeur spéciale de timerId.

```
segment <running-timer-op>

running-timer-op
(timerId)

let { // portée locale de la variable myStatus

    var statusType myStatus;    // statusType est un type symbolique pour les
                                // valeurs de statut des temporisations.

    if (timerId == 'any timer') {
        if (Entity.STATUS) == SNAPSHOT) {
            timerId := Entity.TIMER-STATE.first.random(SNAP-STATUS == RUNNING);
        }
        else {
            timerId := Entity.TIMER-STATE.first.random(STATUS == RUNNING);
        }
    }

    if (timerId != NULL) {
        myStatus := Entity.timerId.STATUS;
        if (Entity.STATUS == SNAPSHOT) {
            myStatus := Entity.timerId.SNAP-STATUS;
        }

        if (myStatus == RUNNING) {
            Entity.VALUE-STACK.push(true);
        }
        else {
            Entity.VALUE-STACK.push(false);
        }
    }
    else {
        Entity.VALUE-STACK.push(false);
    }
} // fin de la portée locale

Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 114/Z.143 – Segment de graphe de flux <running-timer-op>

9.43 Opération self

La structure syntaxique de l'opération **self** est la suivante:

```
self
```

Le segment de graphe de flux <self-op> sur la Figure 115 définit l'exécution de l'opération **self**.

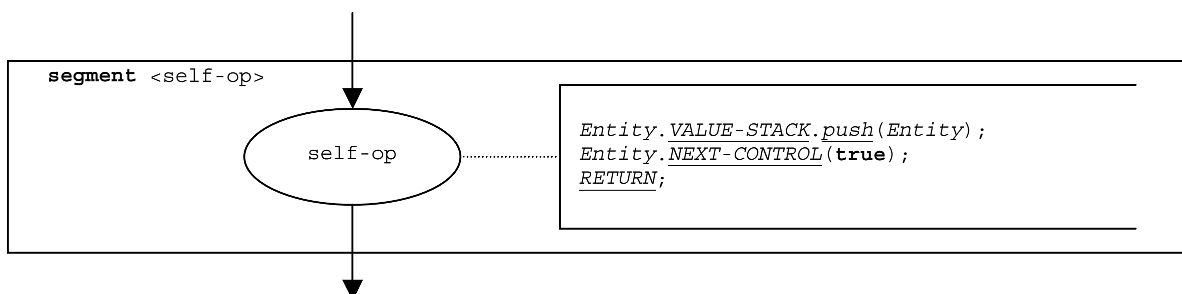


Figure 115/Z.143 – Segment de graphe de flux <self-op>

9.44 Opération send

La structure syntaxique de l'opération **send** est la suivante:

```
<portId>.send (<send-spec>) [to <component-expression>]
```

La partie facultative <component-expression> dans la clause **to** indique l'entité réceptrice. Il peut s'agir d'une valeur de variable ou de la valeur de retour d'une fonction.

Le segment de graphe de flux <send-op> sur la Figure 116 définit l'exécution d'une opération **send**.

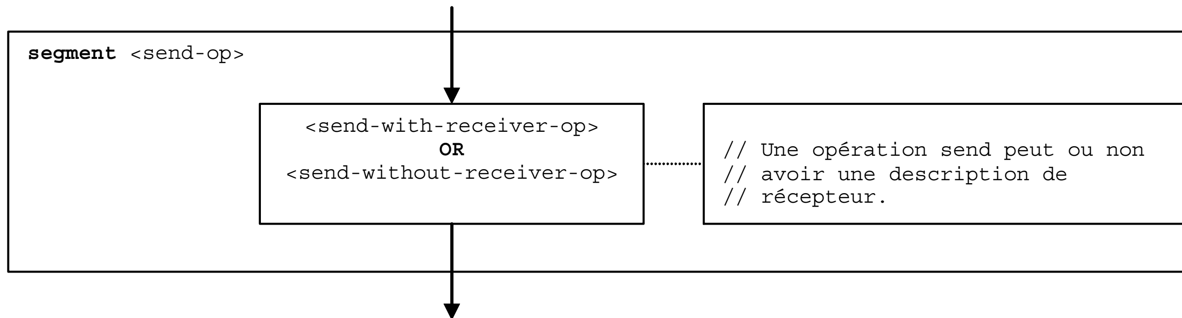


Figure 116/Z.143 – Segment de graphe de flux <send-op>

9.44.1 Segment de graphe de flux <send-with-receiver-op>

Le segment de graphe de flux <send-with-receiver-op> sur la Figure 117 définit l'exécution d'une opération **send** pour laquelle le récepteur est spécifié sous la forme d'une expression.

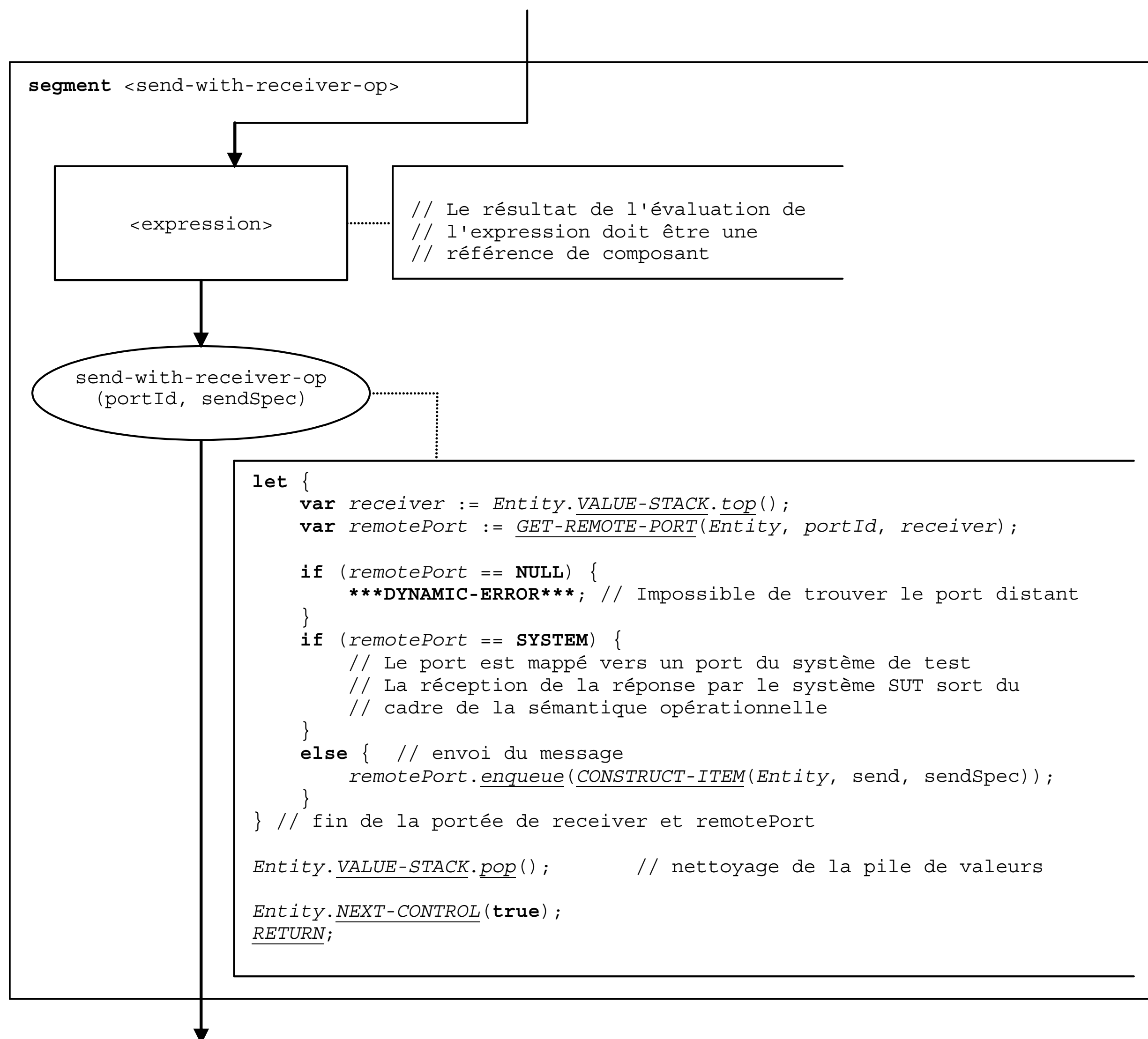


Figure 117/Z.143 – Segment de graphe de flux <send-with-receiver-op>

9.44.2 Segment de graphe de flux <send-without-receiver-op>

Le segment de graphe de flux `<send-without-receiver-op>` sur la Figure 118 définit l'exécution d'une opération **`send`** sans clause **`to`**.

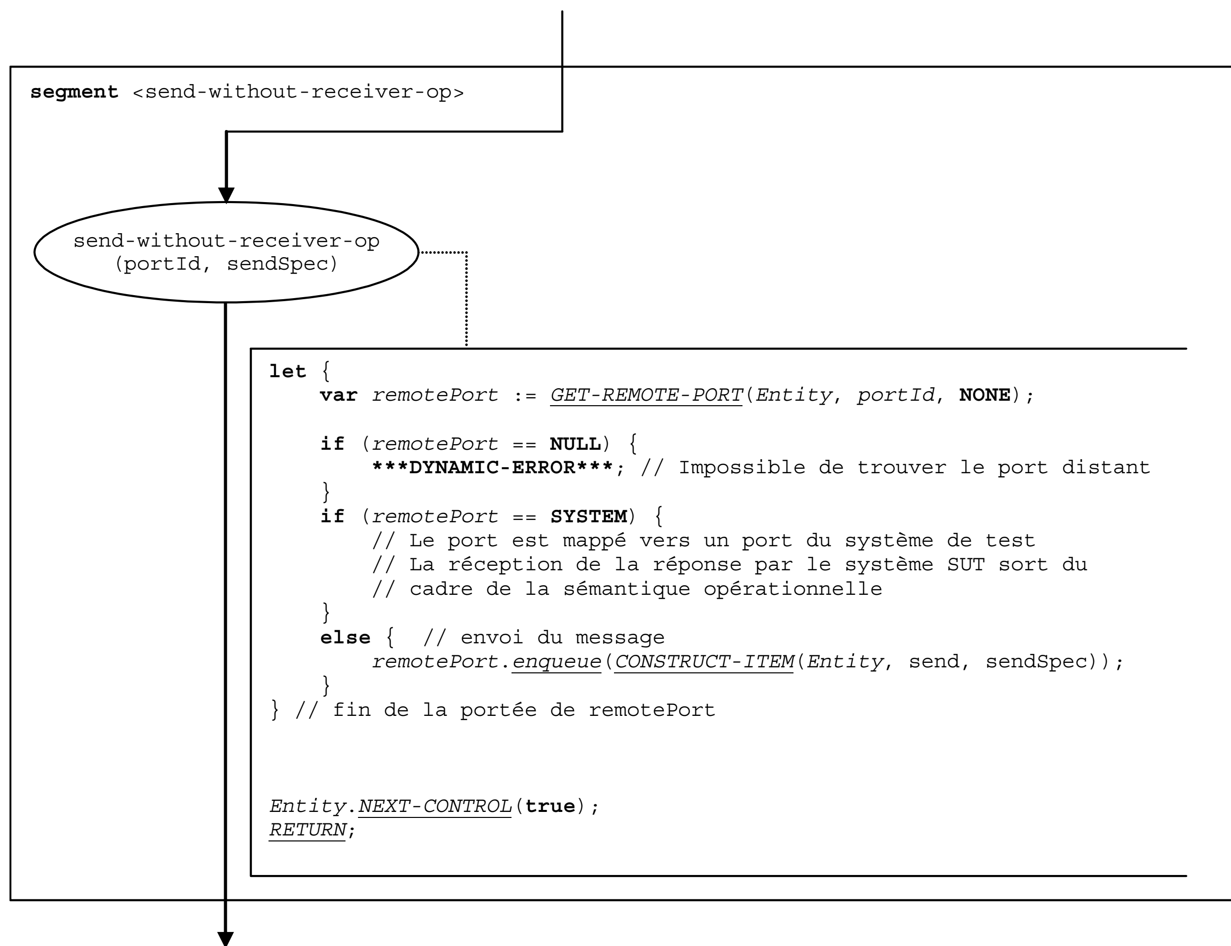


Figure 118/Z.143 – Segment de graphe de flux <send-without-receiver-op>

9.45 Opération setverdict

La structure syntaxique de l'opération **setverdict** est la suivante:

```
setverdict(<verdicttype-expression>)
```

Le paramètre <verdicttype-expression> de l'opération **setverdict** est une expression dont l'évaluation doit donner une valeur de type **verdicttype**, à savoir **none**, **pass**, **inconc** ou **fail**. L'expression est évaluée avant que l'opération **setverdict** soit appliquée.

Le segment de graphe de flux <setverdict-op> sur la Figure 119 définit l'exécution de l'opération **setverdict**.

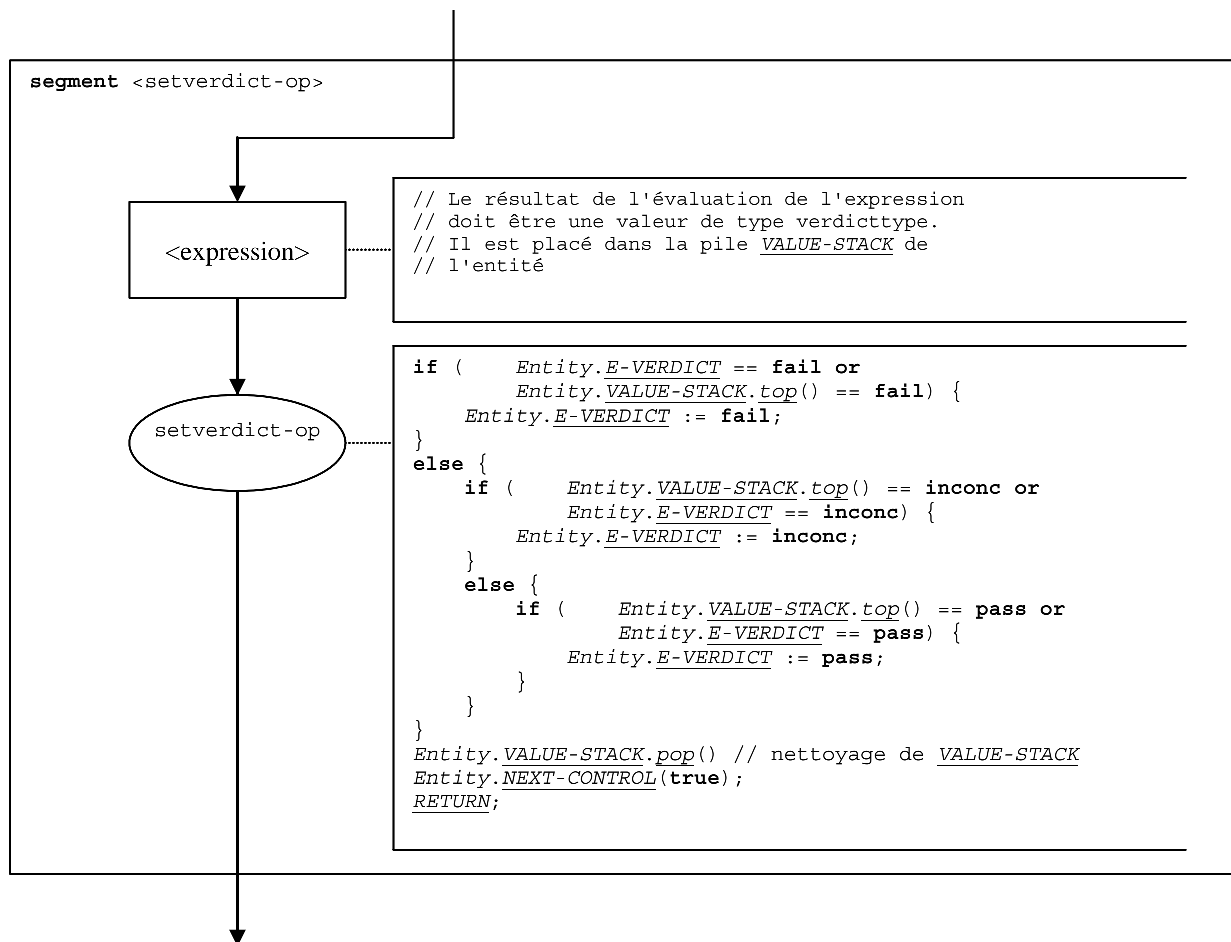


Figure 119/Z.143 – Segment de graphe de flux <setverdict-op>

9.46 Opération start applicable aux composants

La structure syntaxique de l'opération **start** applicable aux composants est la suivante:

```
<component-expression>.start(<function-name>(<act-par-desc_1>,..., <act-par-desc_n>))
```

L'opération **start** applicable aux composants démarre un composant nouvellement créé. Une référence de composant identifie le composant à démarrer. La référence peut être stockée dans une variable ou être retournée par une fonction, autrement dit il s'agit d'une expression dont l'évaluation donne une référence de composant.

Le nom <function-name> est le nom de la fonction qui définit le comportement du nouveau composant et les descriptions <act-par-descr$_1$>, ..., <act-par-descr$_n$> décrivent les valeurs des paramètres effectifs de la fonction <function-name>. Dans les fonctions référencées dans les opérations **start** applicables aux composants, seuls les paramètres par valeur sont autorisés. Les descriptions des paramètres effectifs sont données sous la forme d'expressions qui doivent être évaluées avant que l'appel puisse être exécuté. Le traitement des paramètres par valeur formels et effectifs est analogue à leur traitement dans les appels de fonction (voir § 9.24).

Le segment de graphe de flux <start-component-op> sur la Figure 120 définit l'exécution de l'opération **start** applicable aux composants. Cette opération est exécutée en quatre étapes. Dans la première étape, un enregistrement d'appel est créé. Dans la deuxième étape, les valeurs des paramètres effectifs sont calculées. Dans la troisième étape, la référence du composant à démarrer est extraite et, dans la quatrième étape, la commande et l'enregistrement d'appel sont donnés au nouveau composant.

NOTE – Le segment de graphe de flux sur la Figure 120 inclut le traitement des paramètres par référence (<ref-var-par-calc>). Des paramètres par référence sont nécessaires pour expliquer les paramètres par référence des tests élémentaires. Dans la sémantique opérationnelle, on suppose que ces paramètres sont traités par le composant MTC.

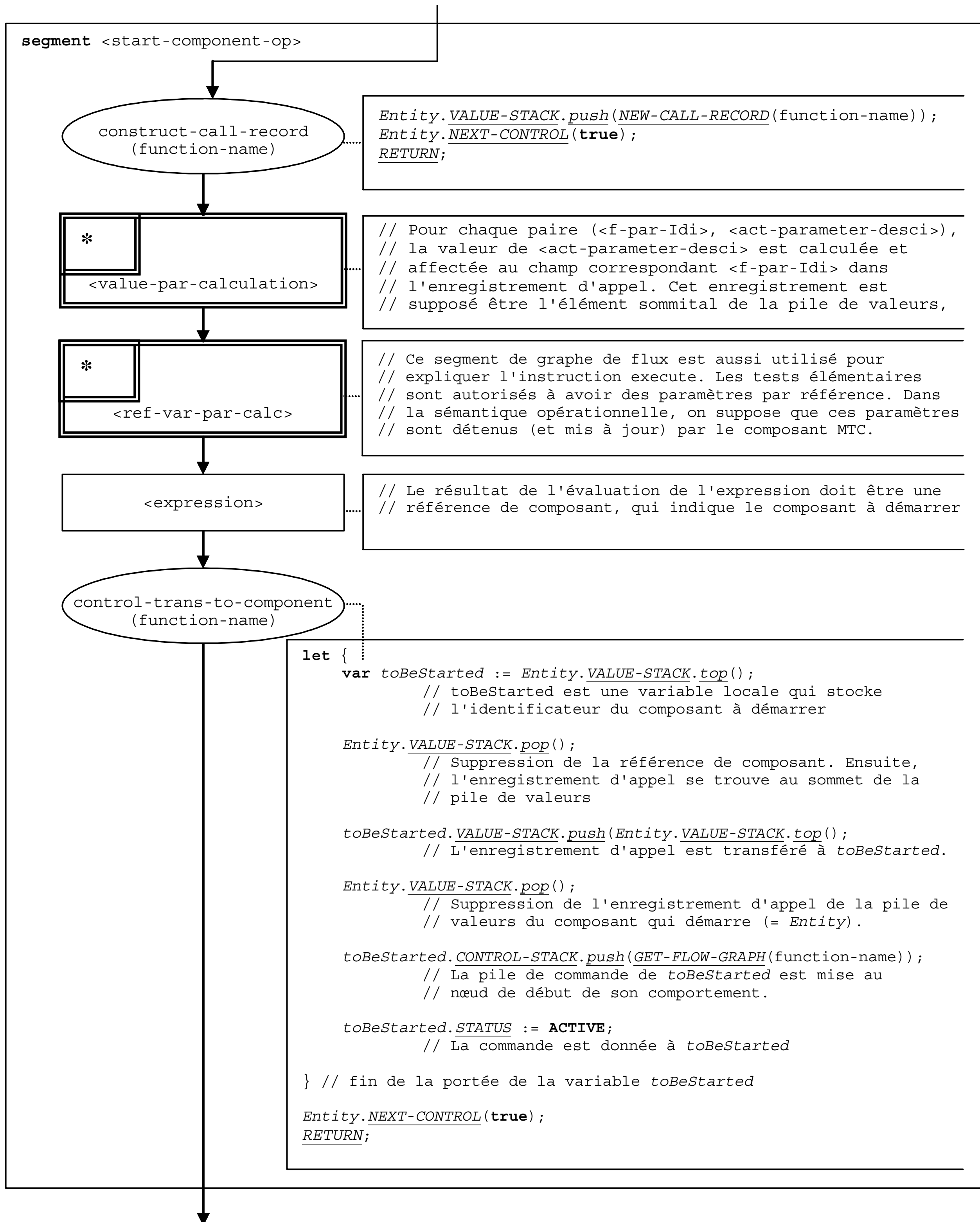


Figure 120/Z.143 – Segment de graphe de flux <start-component-op>

9.47 Opération start applicable aux ports

La structure syntaxique de l'opération **start** applicable aux ports est la suivante:

```
<portId>.start
```

Le segment de graphe de flux <start-port-op> sur la Figure 121 définit l'exécution de l'opération **start** applicable aux ports.

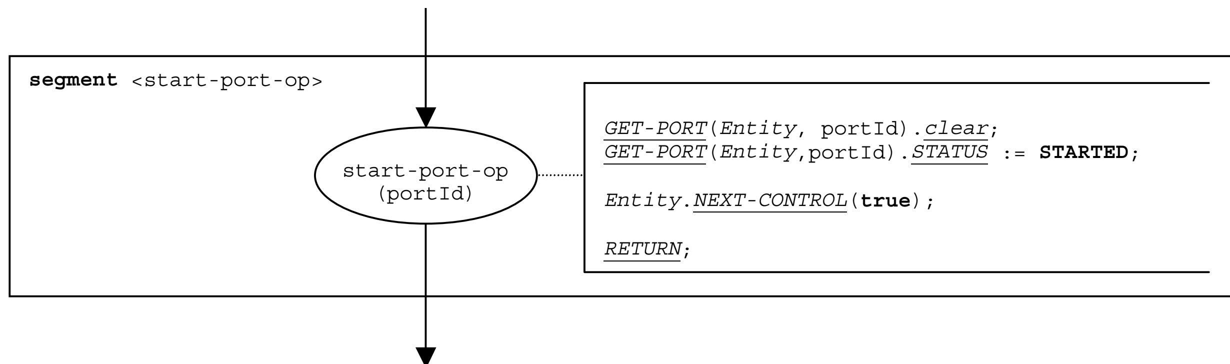


Figure 121/Z.143 – Segment de graphe de flux <start-port-op>

9.48 Opération start applicable aux temporisations

La structure syntaxique de l'opération **start** applicable aux temporisations est la suivante:

```
<timerId>.start [(<float-expression>)]
```

Le paramètre facultatif <float-expression> de l'opération **start** applicable aux temporisations indique la durée effective de la temporisation. Si ce paramètre n'est pas fourni, l'opération **start** utilisera la durée par défaut. L'évaluation de l'expression doit donner une valeur de type **float**. Si le paramètre est fourni, l'expression doit être évaluée avant que l'opération **start** soit appliquée. Le résultat de l'évaluation est placé dans la pile *VALUE-STACK* de *Entity*.

Le segment de graphe de flux <start-timer-op> sur la Figure 122 définit l'exécution de l'opération **start** applicable aux temporisations.

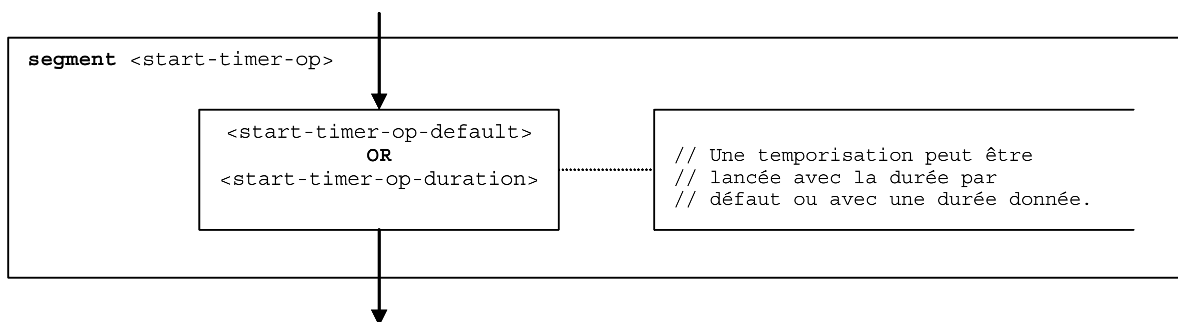


Figure 122/Z.143 – Segment de graphe de flux <start-timer-op>

9.48.1 Segment de graphe de flux <start-timer-op-default>

Le segment de graphe de flux <start-timer-op-default> sur la Figure 123 définit l'exécution de l'opération **start** applicable aux temporisations avec la valeur par défaut.

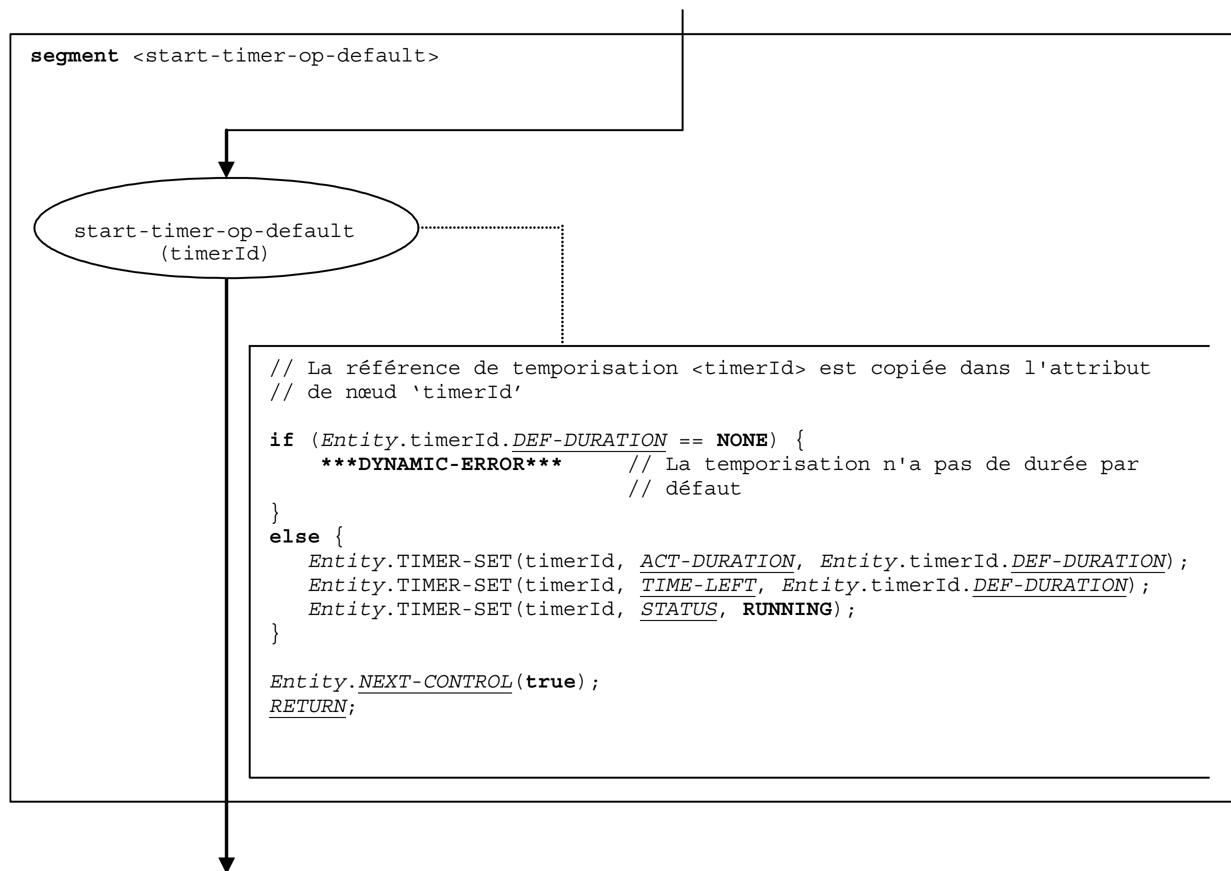


Figure 123/Z.143 – Segment de graphe de flux <start-timer-op-default>

9.48.2 Segment de graphe de flux <start-timer-op-duration>

Le segment de graphe de flux `<start-timer-op-duration>` sur la Figure 124 définit l'exécution de l'opération **`start`** applicable aux temporisations avec une durée donnée.

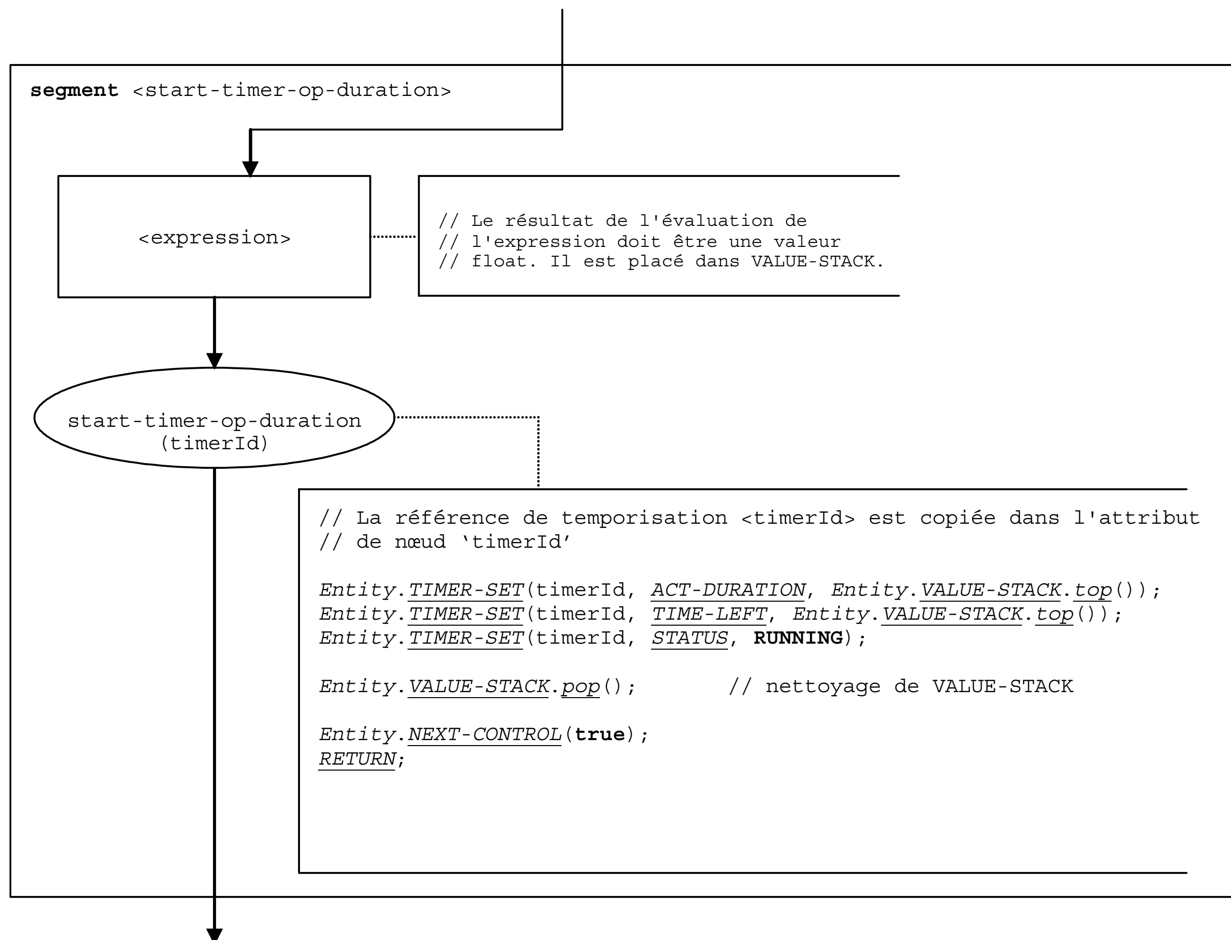


Figure 124/Z.143 – Segment de graphe de flux <start-timer-op-duration>

9.49 Opération stop applicable aux composants

La structure syntaxique de l'opération **stop** applicable aux composants est la suivante:

```
<component-expression>.stop
```

L'opération **stop** applicable aux composants arrête le composant spécifié. Tous les composants de test seront arrêtés, autrement dit le test élémentaire prend fin, si le composant MTC est arrêté (par exemple **mtc.stop**) ou s'arrête lui-même (par exemple **self.stop**). Le composant MTC peut arrêter tous les composants de test en parallèle en utilisant le mot clé **all**, à savoir **all component.stop**.

Un composant à arrêter est identifié par une référence de composant fournie sous la forme d'une expression, par exemple une valeur ou une fonction retournant une valeur. Dans un souci de simplicité, le mot clé '**all component**' est considéré comme une valeur spéciale de <component-expression>. Les opérations **mtc** et **self** sont évaluées conformément aux § 9.33 et 9.43.

Le segment de graphe de flux <stop-component-op> sur la Figure 125 définit l'exécution de l'opération **stop** applicable aux composants.

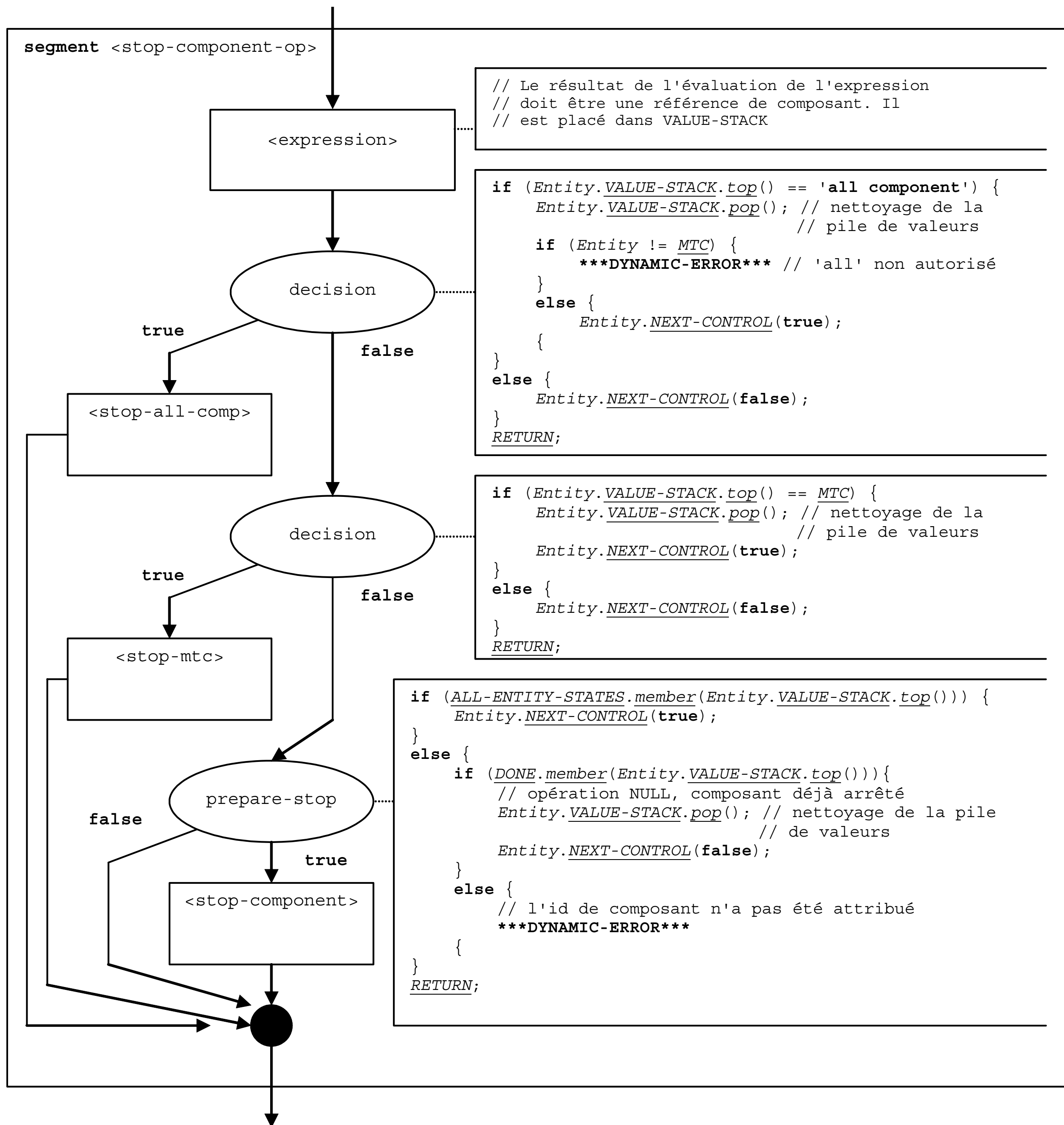


Figure 125/Z.143 – Segment de graphe de flux <stop-component-op>

9.49.1 Segment de graphe de flux <stop-mtc>

Le segment de graphe de flux `<stop-mtc>` sur la Figure 126 décrit l'arrêt d'un composant MTC d'un test élémentaire, ce qui a pour effet de terminer le test élémentaire, autrement dit le verdict final est calculé et placé dans la pile de valeurs de la commande de module, toutes les ressources sont libérées, la liste DONE de l'état de module est vidée et tous les composants de test y compris le composant MTC sont terminés.

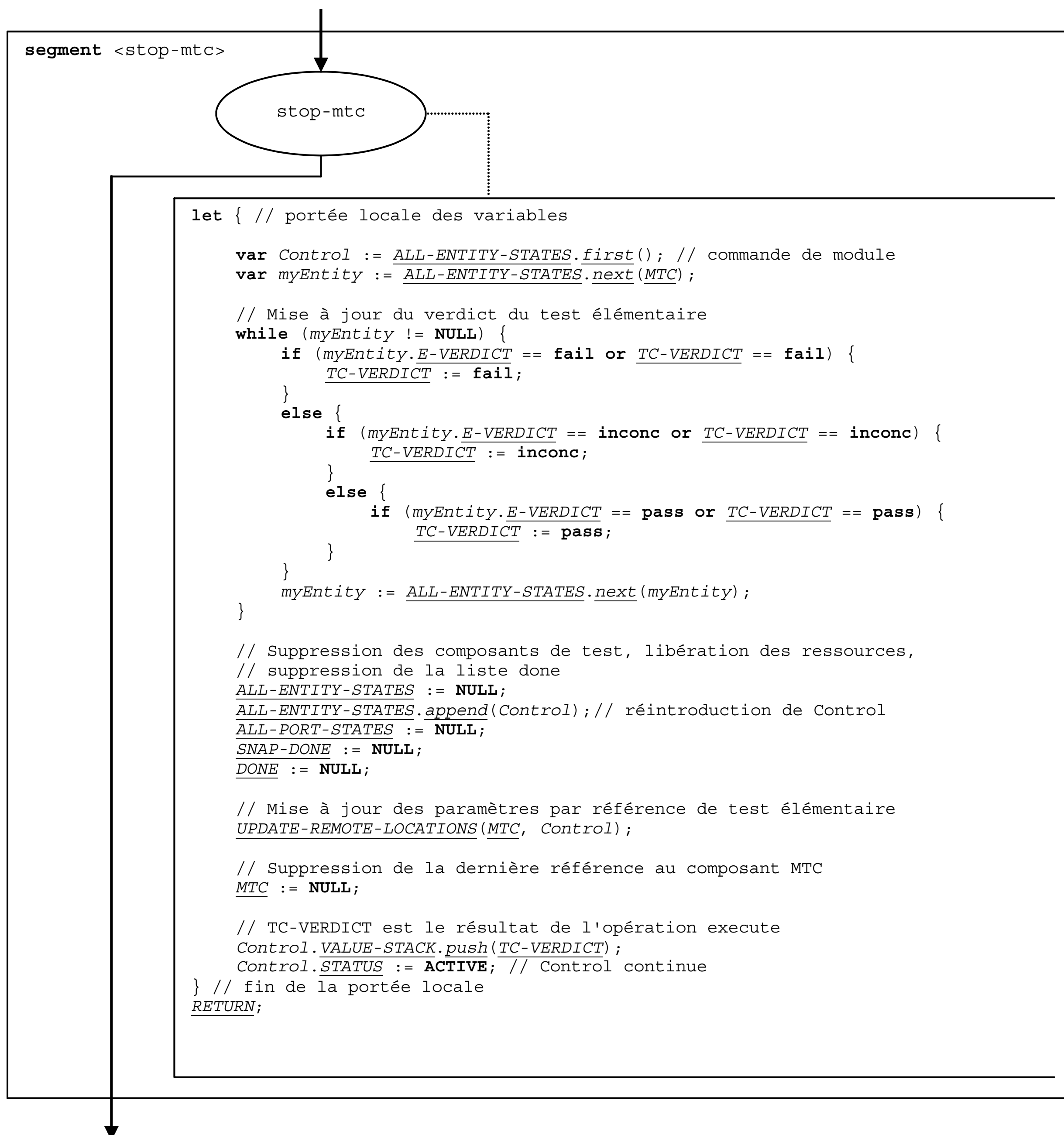


Figure 126/Z.143 – Segment de graphe de flux <stop-mtc-op>

9.49.2 Segment de graphe de flux <stop-component>

Le segment de graphe de flux `<stop-component>` sur la Figure 127 décrit l'arrêt d'un composant de test en parallèle, autrement dit pas le composant MTC ni la commande de module. Cet arrêt a pour effet que le verdict de test élémentaire *TC-VERDICT* et la liste des composants de test terminés (*DONE*) sont mis à jour et que le composant est supprimé de l'état de module. Dans le segment de graphe de flux `<stop-component>`, on suppose que l'identificateur du composant à arrêter se trouve au sommet de la pile de valeurs du composant qui exécute le segment.

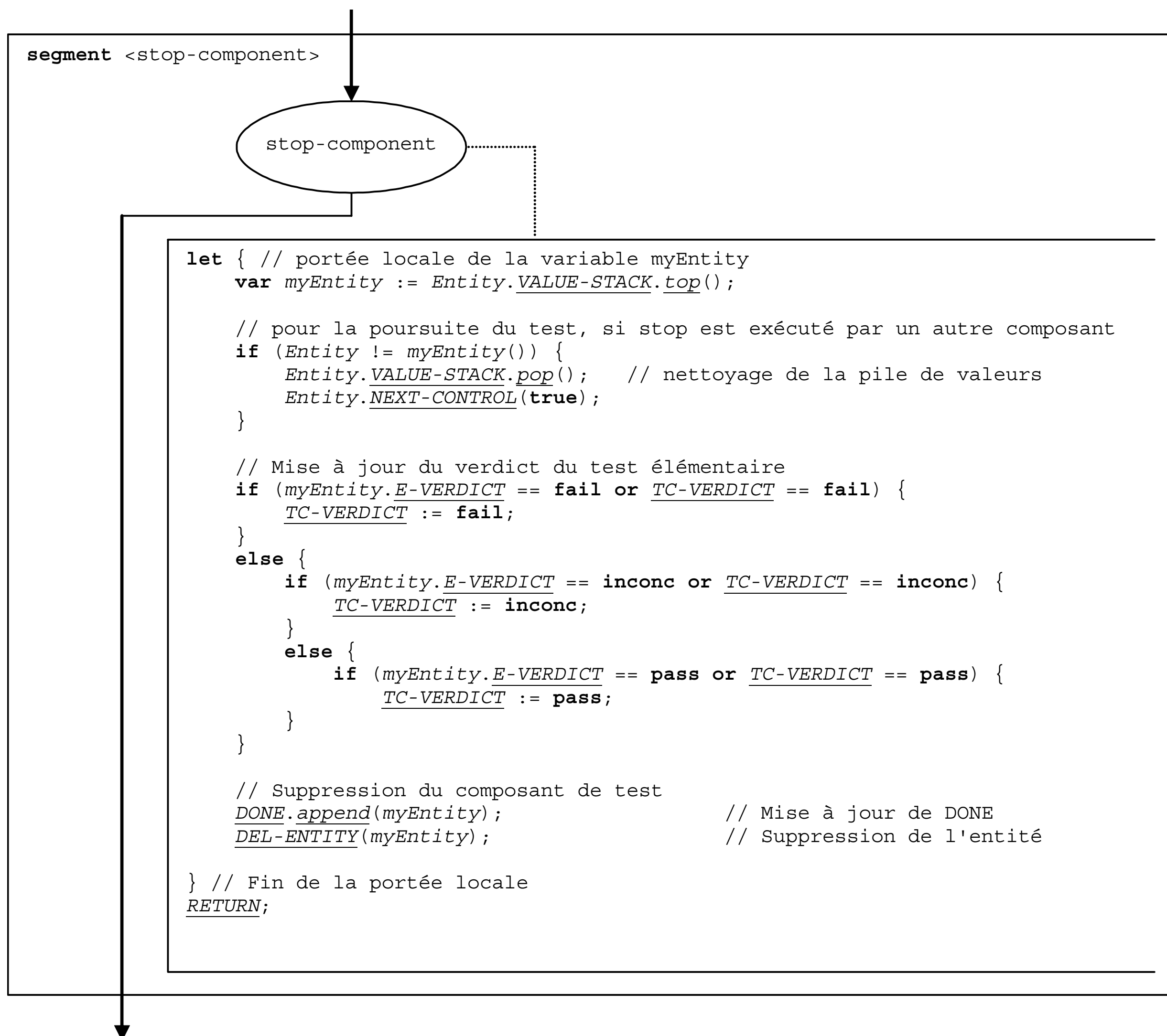


Figure 127/Z.143 – Segment de graphe de flux <stop-component>

9.49.3 Segment de graphe de flux <stop-all-comp>

Le segment de graphe de flux `<stop-all-comp>` sur la Figure 128 décrit l'arrêt des composants de test en parallèle d'un test élémentaire.

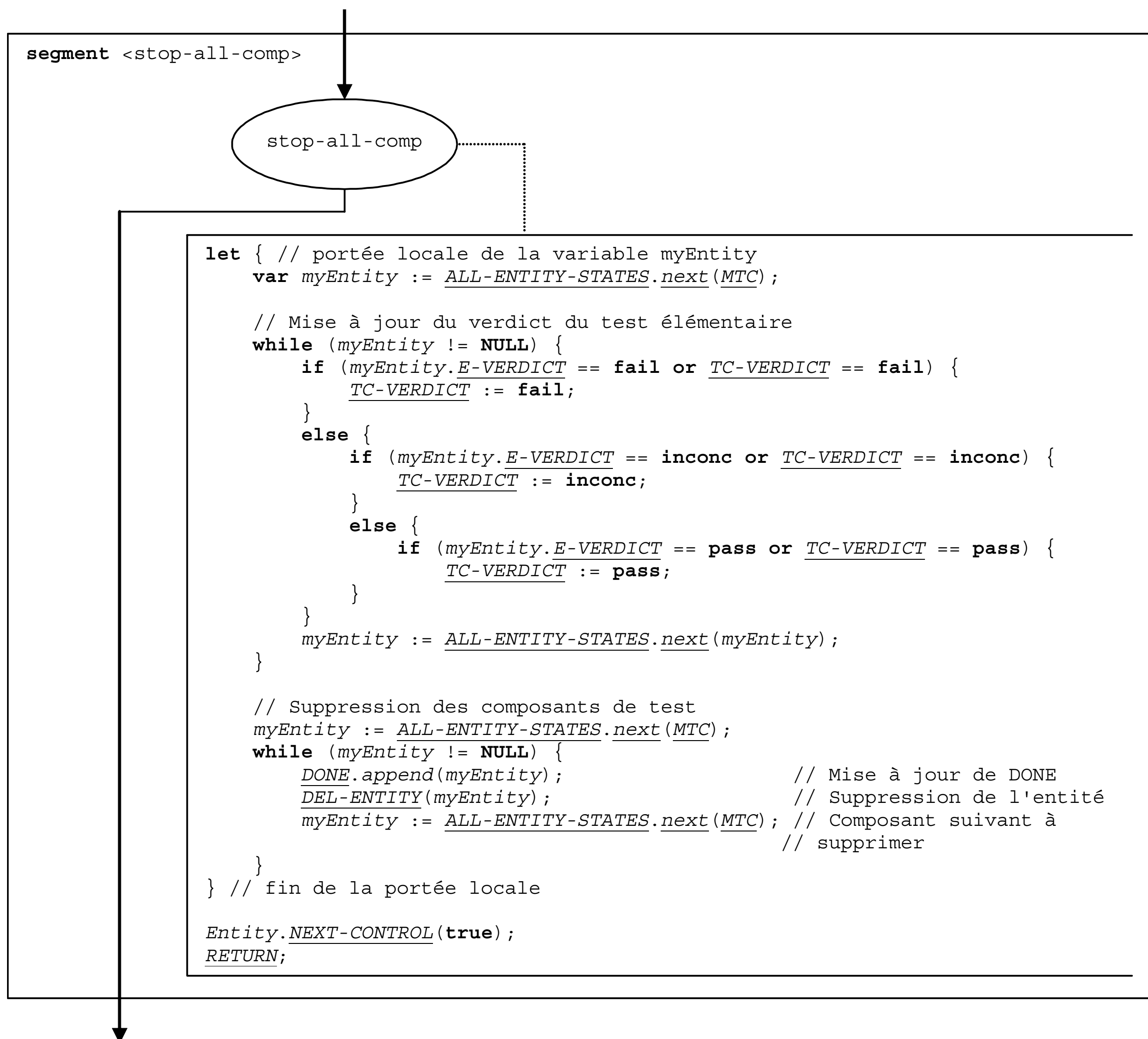


Figure 128/Z.143 – Segment de graphe de flux <stop-all-comp>

9.50 Instruction stop applicable à l'exécution

La structure syntaxique de l'instruction **`stop`** applicable à l'exécution est la suivante:

```
stop
```

L'effet de l'instruction **`stop`** applicable à l'exécution dépend de l'entité qui l'exécute:

a) si c'est la commande de module qui l'exécute, la campagne de tests prend fin, autrement dit tous les composants de test et la commande de module disparaissent de l'état de module.

b) si c'est le composant MTC qui l'exécute, l'exécution de tous les composants de test en parallèle et du composant MTC est arrêtée. Le verdict de test élémentaire global est mis à jour et placé dans la pile de valeurs de la commande de module. Enfin, la commande est redonnée à la commande de module et le composant MTC prend fin.

c) si c'est un composant de test qui l'exécute, le verdict de test élémentaire global *TC-VERDICT* et la liste *DONE* globale sont mis à jour. Puis le composant disparaît complètement du module.

Le segment de graphe de flux <stop-exec-stmt> sur la Figure 129 décrit l'exécution de l'instruction **`stop`**.

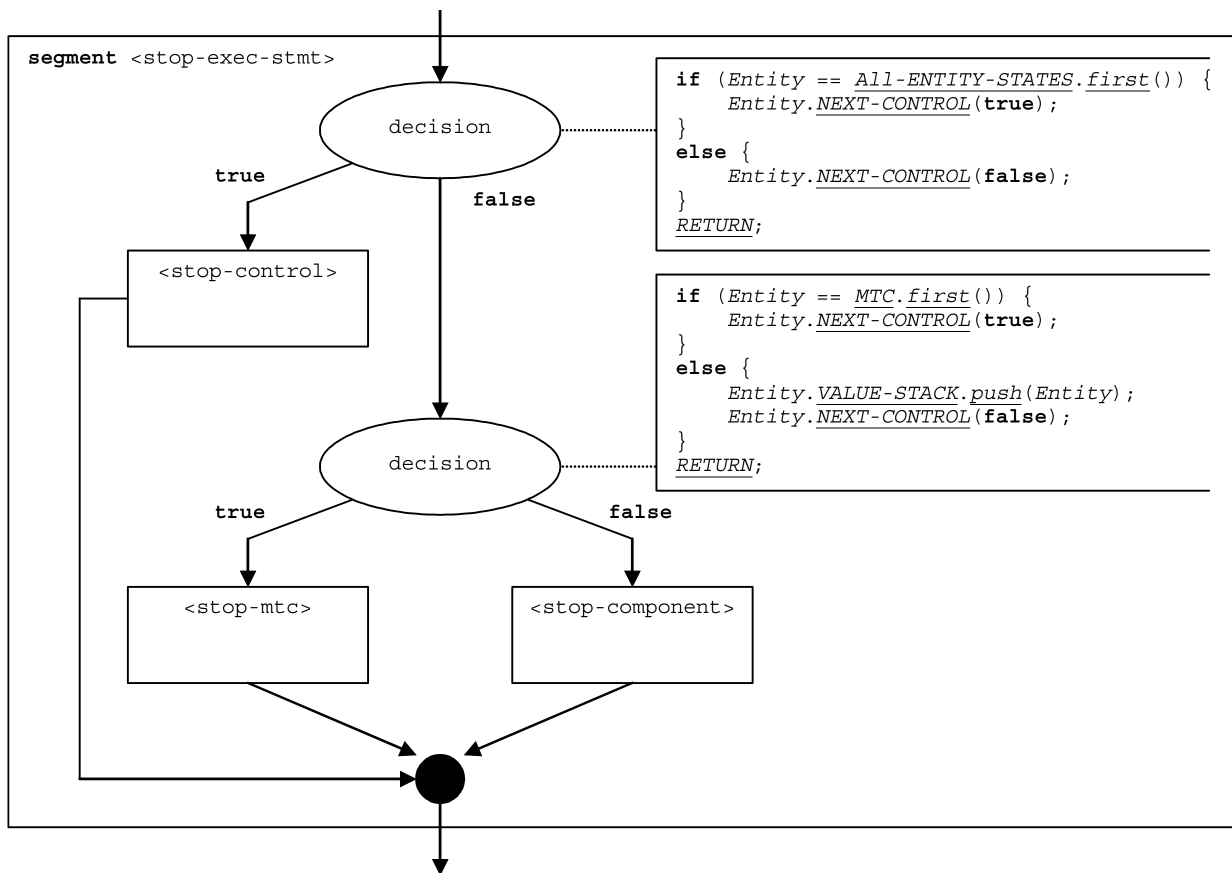


Figure 129/Z.143 – Segment de graphe de flux <stop-exec-stmt>

9.50.1 Segment de graphe de flux <stop-control>

Le segment de graphe de flux `<stop-control>` sur la Figure 130 décrit l'arrêt de la commande de module. L'effet est que *ALL-ENTITY-STATES* est mis à **`NULL`**, à savoir que la condition de terminaison de la procédure d'évaluation de module (voir § 8.6) est remplie.

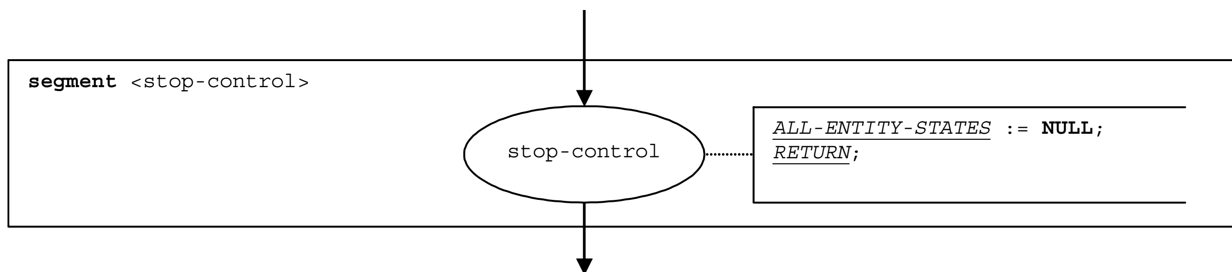


Figure 130/Z.143 – Segment de graphe de flux <stop-control>

9.51 Opération stop applicable aux ports

La structure syntaxique de l'opération **`stop`** applicable aux ports est la suivante:

```
<portId>.stop
```

Le segment de graphe de flux `<stop-port-op>` sur la Figure 131 définit l'exécution de l'opération **`stop`** applicable aux ports.

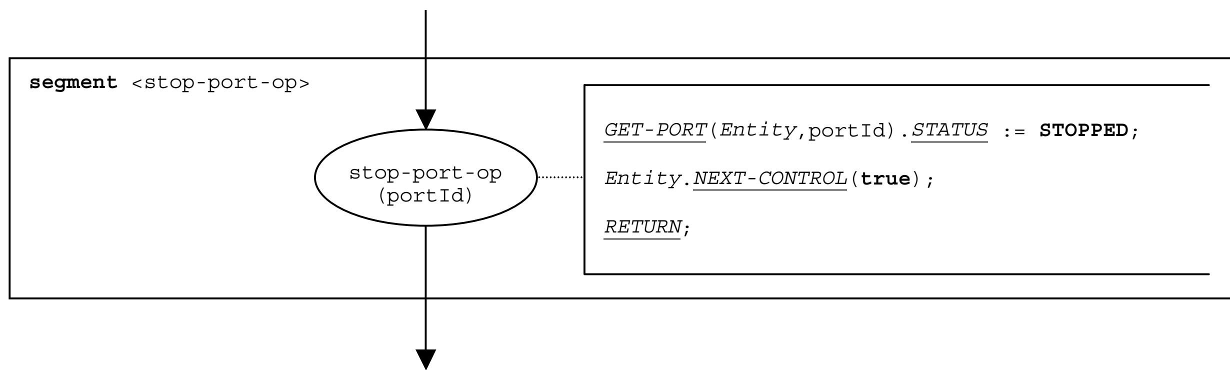


Figure 131/Z.143 – Segment de graphe de flux <stop-port-op>

9.52 Opération stop applicable aux temporisations

La structure syntaxique de l'opération **stop** applicable aux temporisations est la suivante:

```
<timerId>.stop
```

Le segment de graphe de flux `<stop-timer-op>` sur la Figure 132 définit l'exécution de l'opération **stop** applicable aux temporisations.

Le mot clé **all** est traité comme une valeur spéciale de `timerId`.

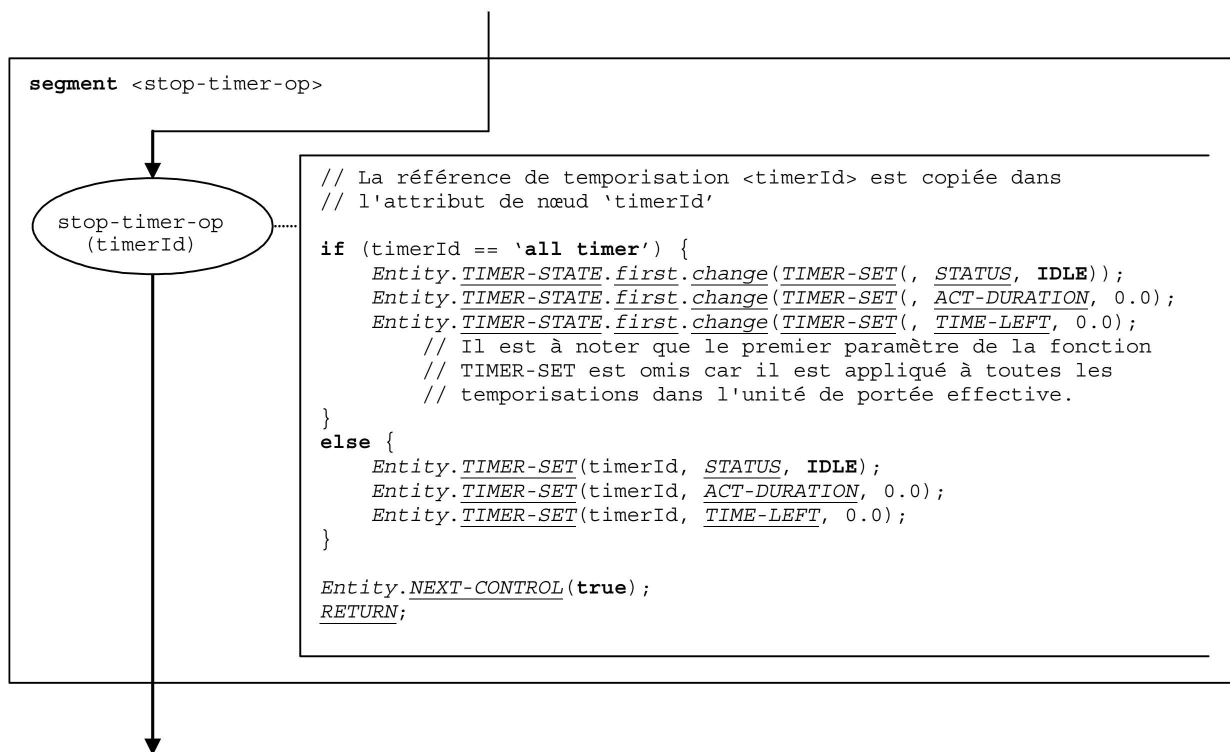


Figure 132/Z.143 – Segment de graphe de flux <stop-timer-op>

9.53 Opération system

La structure syntaxique de l'opération **system** est la suivante:

```
system
```

Le segment de graphe de flux `<system-op>` sur la Figure 133 définit l'exécution de l'opération **system**.

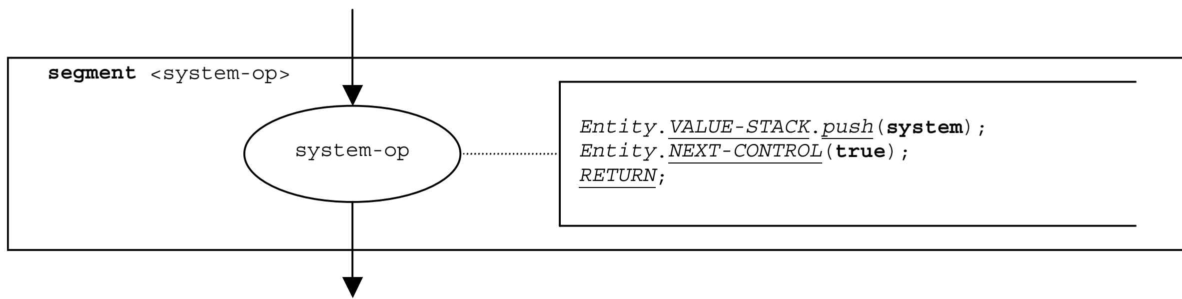


Figure 133/Z.143 – Segment de graphe de flux <system-op>

9.54 Déclaration d'une temporisation

La structure syntaxique de la déclaration de **timer** est la suivante:

```
timer <timerId> [:= <float-expression>]
```

L'effet de la déclaration d'une temporisation est la création d'un nouveau lien de temporisation. La déclaration d'une durée par défaut est facultative. La valeur par défaut est considérée comme une expression dont l'évaluation donne une valeur du type **float**.

Le segment de graphe de flux <timer-declaration> sur la Figure 134 définit l'exécution de la déclaration d'une temporisation.

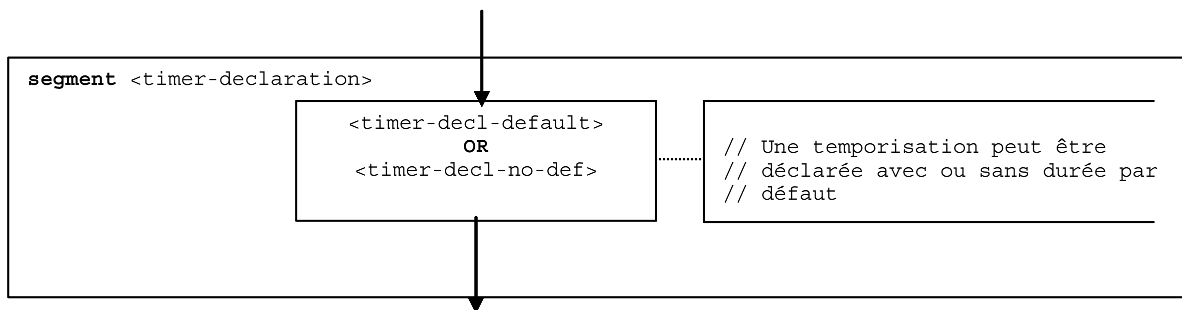


Figure 134/Z.143 – Segment de graphe de flux <timer-declaration>

9.54.1 Segment de graphe de flux <timer-decl-default>

Le segment de graphe de flux <timer-decl-default> sur la Figure 135 définit l'exécution de la déclaration d'une temporisation pour laquelle une durée par défaut est fournie sous la forme d'une expression.

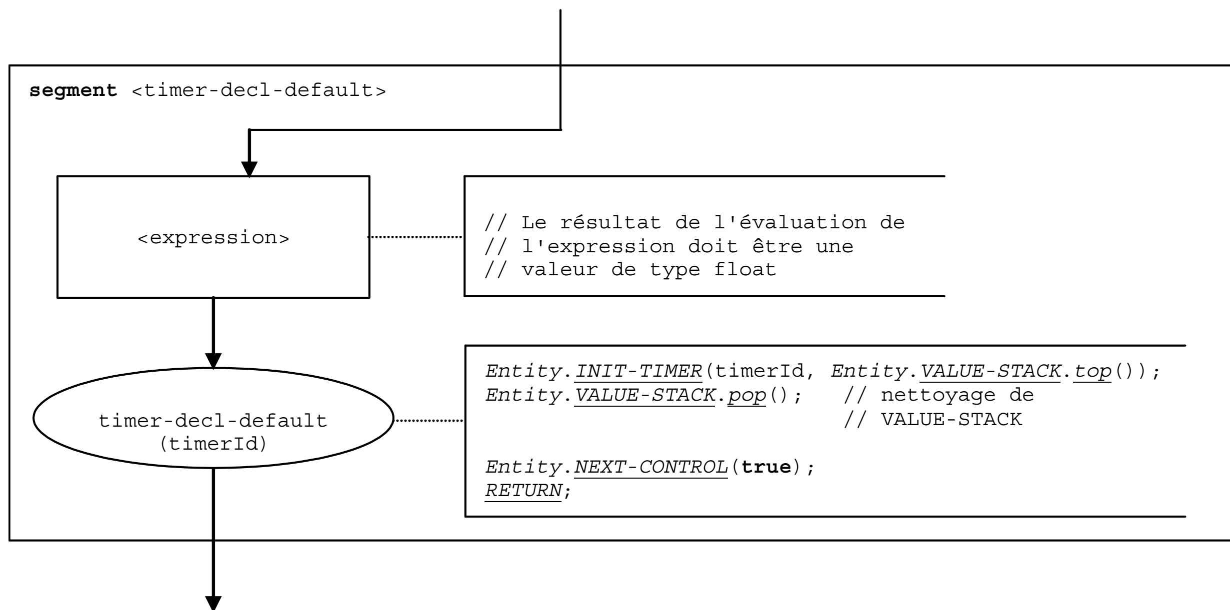


Figure 135/Z.143 – Segment de graphe de flux <timer-decl-default>

9.54.2 Segment de graphe de flux <timer-decl-no-def>

Le segment de graphe de flux `<timer-decl-no-def>` sur la Figure 136 définit l'exécution de la déclaration d'une temporisation pour laquelle aucune durée par défaut n'est fournie, autrement dit la durée par défaut de la temporisation n'est pas définie.

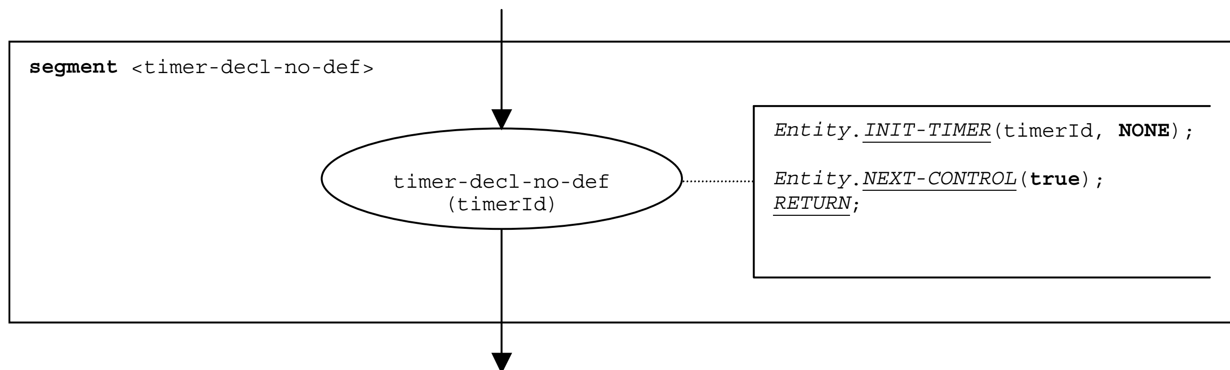


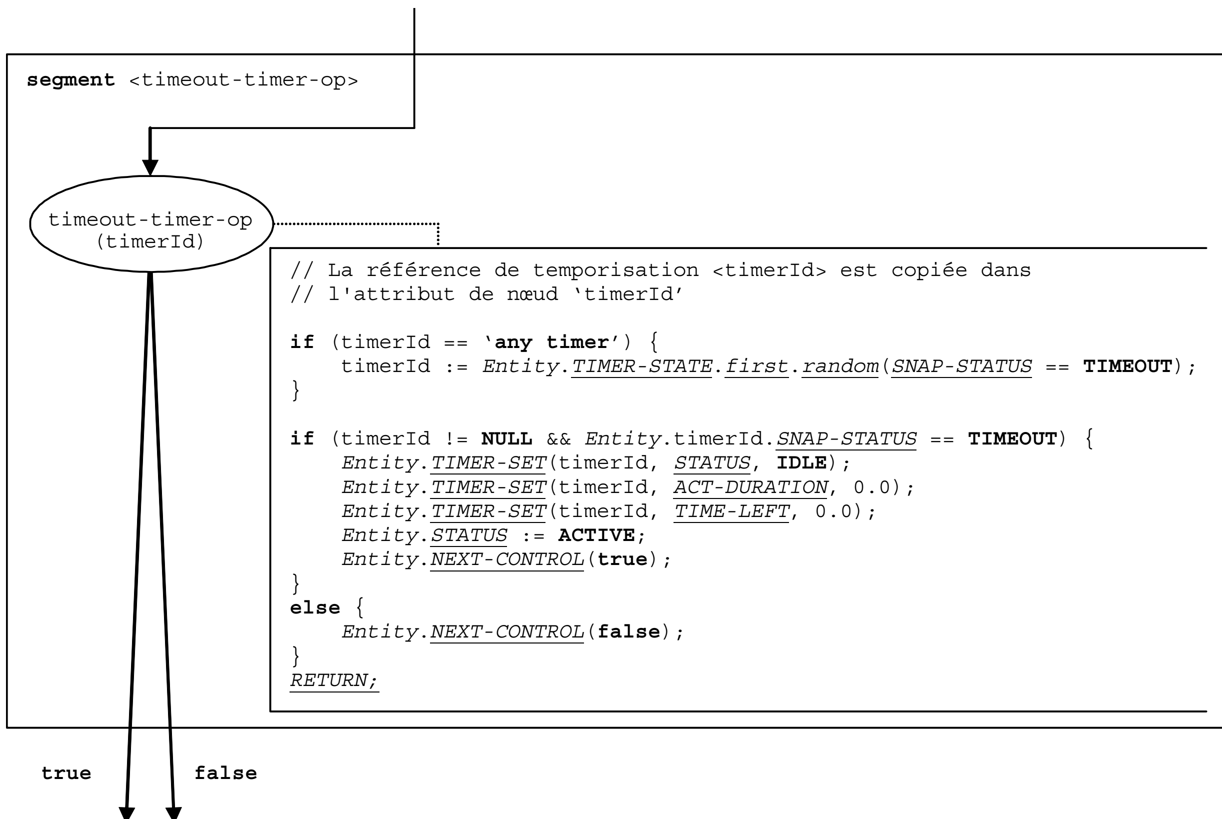
Figure 136/Z.143 – Segment de graphe de flux <timer-decl-no-def>

9.55 Opération timeout applicable aux temporisations

La structure syntaxique de l'opération **`timeout`** applicable aux temporisations est la suivante:

```
<timerId>.timeout
```

Le segment de graphe de flux `<timeout-timer-op>` sur la Figure 137 définit l'exécution de l'opération **`timeout`** applicable aux temporisations.



NOTE 1 – Une opération **timeout** est imbriquée dans une instruction **alt**. Son évaluation est fondée sur l'instantané effectif, autrement dit la décision est fondée sur l'entrée *SNAP-STATUS* du lien de temporisation. Si l'opération **timeout** aboutit, à savoir *SNAP-STATUS* == TIMEOUT, la temporisation passe à l'état IDLE et l'état de composant passe de SNAPSHOT à ACTIVE.

NOTE 2 – Une fois que l'évaluation de l'opération **timeout** donne **true** ou **false**, l'exécution continue avec l'instruction qui suit l'opération **timeout** (branche **true**) ou l'alternative suivante figurant dans l'instruction **alt** doit être vérifiée (branche **false**).

NOTE 3 – Le mot clé **any** est traité comme une valeur spéciale de timerId.

Figure 137/Z.143 – Segment de graphe de flux <timeout-timer-op>

9.56 Opération unmap

La structure syntaxique de l'opération **unmap** est la suivante:

```
unmap(<component_expression>:<portId1>,system:<portId2>)
```

Les identificateurs <portId1> et <portId2> sont considérés comme étant des identificateurs de port du composant de test et de l'interface du système de test correspondant. Il est fait référence au composant associé à l'identificateur <portId1> au moyen de la référence de composant <component-expression>. La référence peut être stockée dans une variable ou retournée par une fonction, autrement dit il s'agit d'une expression dont l'évaluation donne une référence de composant. La pile de valeurs est utilisée pour le stockage de la référence de composant.

NOTE – En ce qui concerne l'opération **unmap**, peu importe si l'instruction **system**:<portId> apparaît comme le premier ou le deuxième paramètre. Dans un souci de simplicité, on suppose qu'il s'agit toujours du deuxième paramètre.

L'exécution de l'opération **unmap** est définie par le segment de graphe de flux <unmap-op> illustré sur la Figure 138.

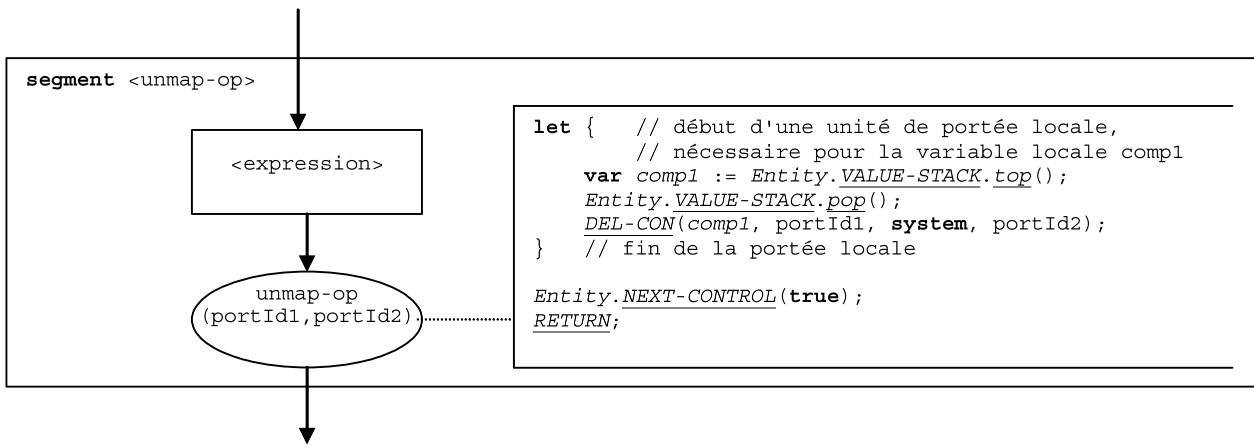


Figure 138/Z.143 – Segment de graphe de flux <unmap-op>

9.57 Déclaration d'une variable

La structure syntaxique de la déclaration d'une **variable** est la suivante:

```
var <varType> <varId> [:= <varType-expression>]
```

L'initialisation d'une variable par la fourniture d'une valeur initiale (sous la forme d'une expression) est facultative. La valeur initiale est considérée comme une expression dont l'évaluation donne une valeur du type de la variable.

Le segment de graphe de flux `<variable-declaration>` sur la Figure 139 définit l'exécution de la déclaration d'une variable.

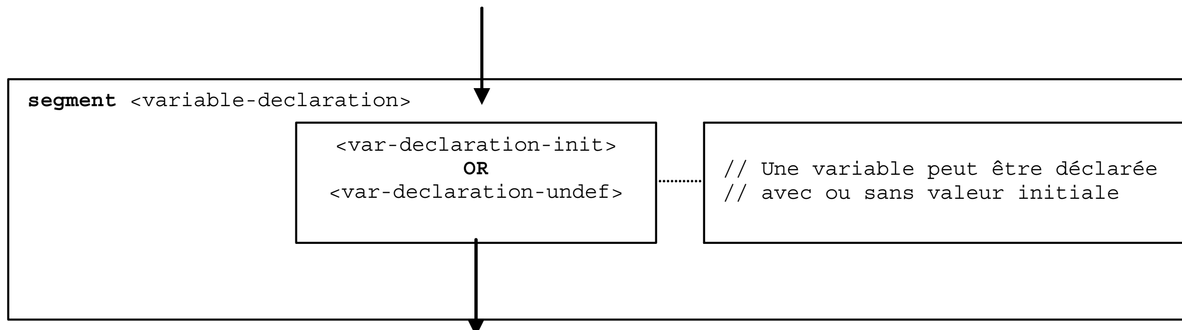


Figure 139/Z.143 – Segment de graphe de flux <variable-declaration>

9.57.1 Segment de graphe de flux <var-declaration-init>

Le segment de graphe de flux `<var-declaration-init>` sur la Figure 140 définit l'exécution de la déclaration d'une variable pour laquelle une valeur initiale est fournie sous la forme d'une expression.

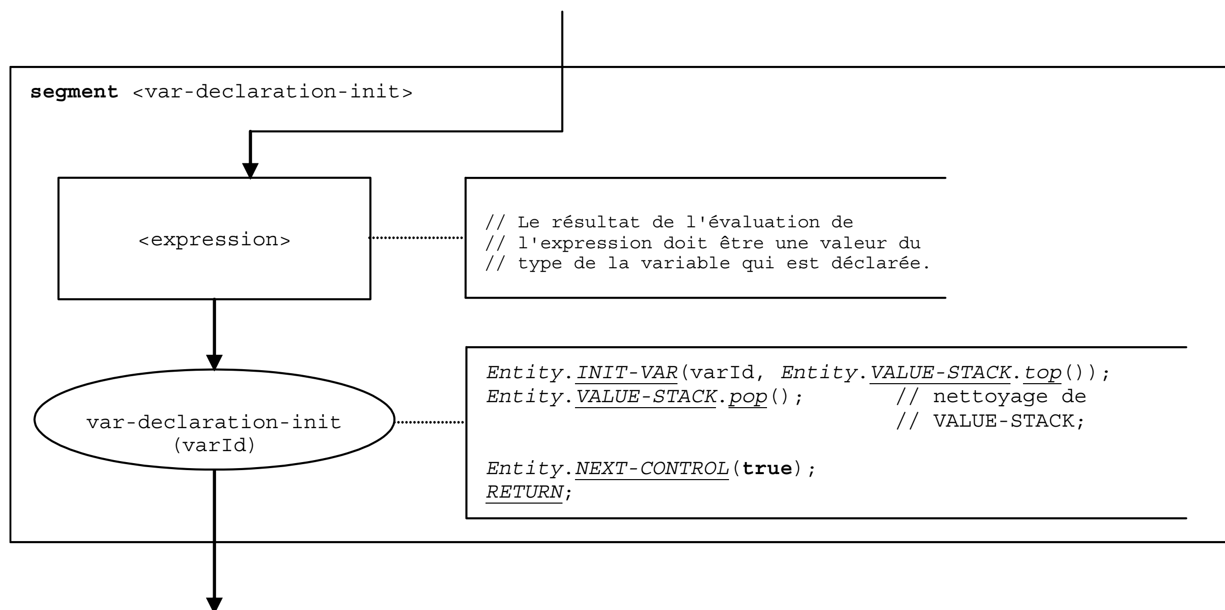


Figure 140/Z.143 – Segment de graphe de flux <var-declaration-init>

9.57.2 Segment de graphe de flux <var-declaration-undef>

Le segment de graphe de flux `<var-declaration-init>` sur la Figure 141 définit l'exécution de la déclaration d'une variable pour laquelle aucune valeur initiale n'est fournie, autrement dit la valeur de la variable n'est pas définie.

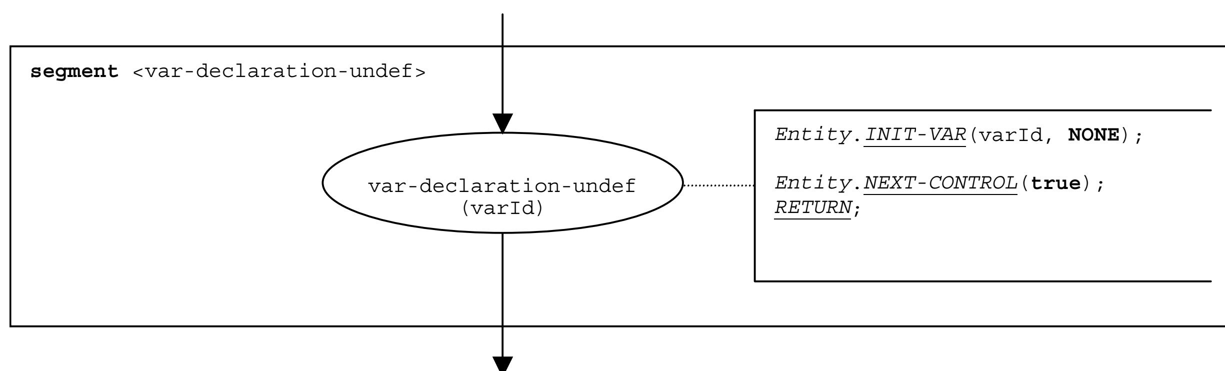


Figure 141/Z.143 – Segment de graphe de flux <var-declaration-undef>

9.58 Instruction while

La structure syntaxique de l'instruction **`while`** est la suivante:

```
while (<boolean-expression>)  <statement-block>
```

L'exécution d'une instruction **`while`** est définie par le segment de graphe de flux `<while-stmt>` illustré sur la Figure 142.

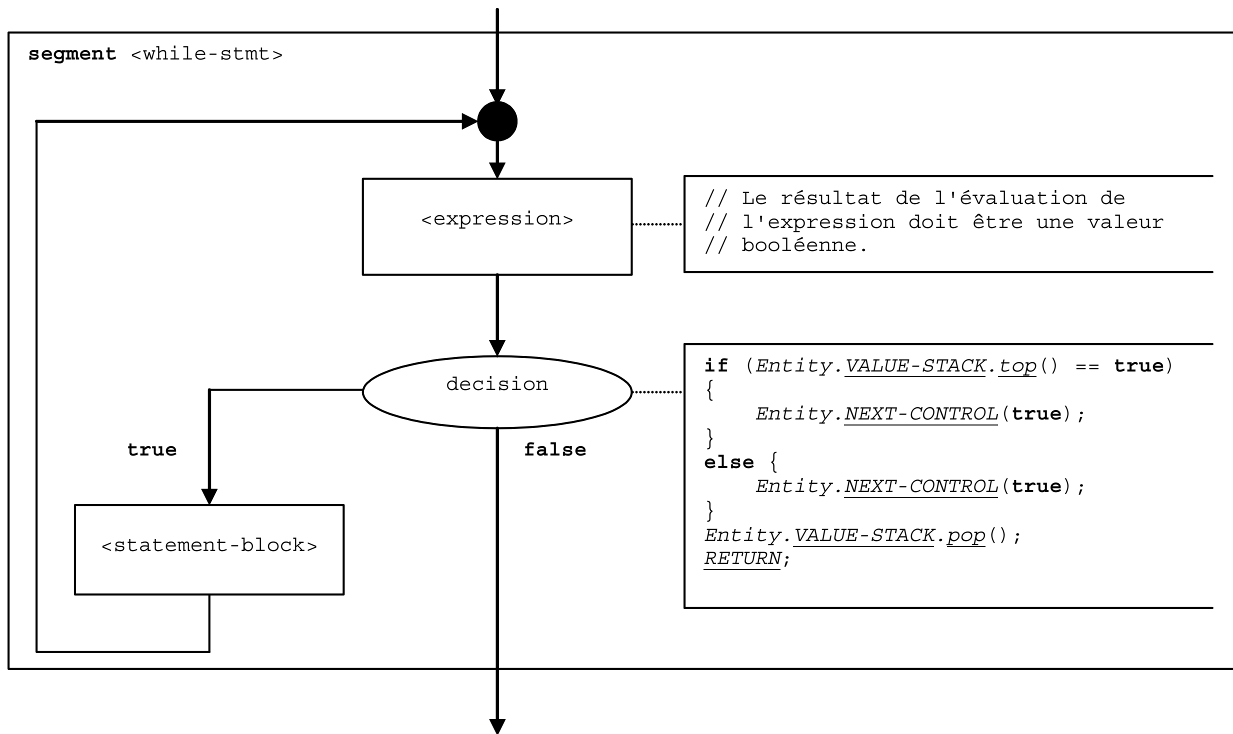


Figure 142/Z.143 – Segment de graphe de flux <while-stmt>

10 Listes des composantes de la sémantique opérationnelle

10.1 Fonctions et états

Nom	Description	Paragraphe
ACT-DURATION	Durée avec laquelle une temporisation active a été lancée	8.3.2.4
add	Opération applicable aux listes: ajoute un item comme premier élément d'une liste	8.3.1.1
ADD-CON	Ajoute une connexion à un état de port	8.3.3.2
ALL-ENTITY-STATES	Etats de composant dans un état de module	8.3.1
ALL-PORT-STATES	Etats de port dans un état de module	8.3.1
append	Opération applicable aux listes: ajoute un item comme dernier élément d'une liste	8.3.1.1
APPLY-OPERATOR	Application d'opérateurs tels que +, – ou /	8.6.2
change	Opération applicable aux listes: modifie tous les éléments d'une liste	8.3.1.1
clear	Opération applicable aux piles: supprime une pile	8.3.2.1
clear	Opération applicable aux files d'attente: supprime tous les éléments d'une file d'attente	8.3.3.2
clear-until	Opération applicable aux piles: supprime des items jusqu'à ce qu'un item particulier soit l'élément sommital de la pile.	8.3.2.1
CONNECTIONS-LIST	Liste des connexions d'un port	8.3.3
CONSTRUCT-ITEM	Construit un item à envoyer	8.4.4
CONTINUE-COMPONENT	Le composant effectif poursuit son exécution	8.6.2
CONTROL-STACK	Pile des nœuds de graphe de flux indiquant l'état de commande effectif d'une entité	8.3.2
DATA-STATE	Etat de données dans un état d'entité	8.3.2
DEF-DURATION	Durée par défaut d'une temporisation	8.3.2.4
DEFAULT-LIST	Liste des comportements par défaut actifs dans un état d'entité	8.3.2
DEFAULT-POINTER	Pointe sur le comportement par défaut effectif pendant l'évaluation du comportement par défaut	8.3.2
DEL-CON	Supprime une connexion d'un état de port	8.3.3.2
DEL-ENTITY	Supprime une entité d'un état de module	8.3.4

Nom	Description	Paragraphe
DEL-TIMER-SCOPE	Supprime une portée de temporisation	8.3.2.5
DEL-VAR-SCOPE	Supprime une portée de variable	8.3.2.3
delete	Opération applicable aux listes: supprime un item d'une liste:	8.3.1.1
dequeue	Opération applicable aux files d'attente: supprime le premier élément d'une file d'attente:	8.3.3.2
DONE	Identificateurs des composants de tests terminés (partie de l'état de module)	8.3.1
E-VERDICT	Verdict de test local d'un composant de test	8.3.2
enqueue	Opération applicable aux files d'attente: place un item comme dernier élément dans une file d'attente:	8.3.3.2
first	Opération applicable aux files d'attente: retourne le premier élément d'une file d'attente	8.3.3.2
first	Opération applicable aux listes: retourne le premier élément d'une liste:	8.3.1.1
GET-FLOW-GRAPH	Extrait le nœud de début d'un graphe de flux	8.2.7
GET-PORT	Extrait une référence de port	8.3.3.2
GET-REMOTE-PORT	Extrait la référence d'un port distant	8.3.3.2
GET-TIMER-LOC	Extrait l'emplacement d'une temporisation	8.3.2.5
GET-UNIQUE-ID	Retourne un nouvel identificateur unique lorsque cette opération est appelée	8.6.2
GET-VAR-LOC	Extrait l'emplacement d'une variable	8.3.2.3
INIT-CALL-RECORD	Initialise les variables des paramètres pour la communication en mode procédure dans l'unité de portée effective du composant de test	8.5.1
INIT-FLOW-GRAPHS	Initialise le traitement des graphes de flux	8.6.2
INIT-TIMER	Crée un nouveau lien de temporisation	8.3.2.5
INIT-TIMER-LOC	Crée un nouveau lien de temporisation avec un emplacement existant	8.3.2.5
INIT-TIMER-SCOPE	Initialise une nouvelle portée de temporisation	8.3.2.5
INIT-VAR	Crée un nouveau lien de variable	8.3.2.3
INIT-VAR-LOC	Crée un nouveau lien de variable avec un emplacement existant	8.3.2.3
INIT-VAR-SCOPE	Initialise une nouvelle portée de variable	8.3.2.3
length	Opération applicable aux listes: retourne la longueur d'une liste:	8.3.1.1
M-CONTROL	Identificateur de commande de module dans l'état de module	8.3.1
MATCH-ITEM	Vérifie l'appariement avec une opération de réception d'un message, d'un appel, d'une réponse ou d'une exception reçu:	8.4.5
member	Opération applicable aux listes: vérifie si un item est un élément d'une liste:	8.3.1.1
MTC	Référence au composant MTC dans l'état de module	8.3.1
NEW-CALL-RECORD	Crée un enregistrement d'appel pour un appel de fonction	8.5.1
NEW-ENTITY	Crée un nouvel état d'entité	8.3.2.1
NEW-PORT	Crée un nouveau port	8.3.3.2
NEXT	Extrait le nœud successeur d'un nœud donné dans un graphe de flux	8.1.6
next	Opération applicable aux listes: retourne l'élément suivant d'une liste:	8.3.1.1
NEXT-CONTROL	Supprime, dans la pile de commande, le nœud de graphe de flux sommital et place, dans cette pile de commande, le nœud de graphe de flux suivant:	8.3.2.1
OWNER	Propriétaire d'un port	8.3.3
pop	Opération applicable aux piles: supprime un item d'une pile:	8.3.2.1
PORT-NAME	Nom d'un port	8.3.3
push	Opération applicable aux piles: place un item dans une pile:	8.3.2.1
random	Opération applicable aux listes: retourne un élément d'une liste choisi au hasard:	8.3.1.1
REMOTE-ENTITY	Entité distante d'une connexion dans un état de port	8.3.3.1
REMOTE-PORT-NAME	Nom d'un port d'une connexion dans un état de port	8.3.3.1
RETRIEVE-INFO	Extrait les informations d'un message, d'un appel, d'une réponse ou d'une exception qui est reçu:	8.4.6
RETURN	Redonne la commande à la procédure d'évaluation de module	8.6.2
SNAP-ACTIVE	Nombre de composants de test actifs lorsque le composant MTC prend un instantané (partie de l'état de module)	8.3.1

Nom	Description	Paragraphe
SNAP-DONE	Liste des composants de test terminés au moment où un instantané est pris	8.3.2
SNAP-PORTS	Offre la fonctionnalité d'instantané, autrement dit met à jour SNAP-VALUE.	8.3.3.2
SNAP-STATUS	Statut d'une temporisation au moment où un instantané est pris	8.3.2.4
SNAP-TIMER	Offre la fonctionnalité d'instantané et met à jour SNAP-VALUE et SNAP-STATUS	8.3.2.5
SNAP-VALUE	Valeur d'une temporisation au moment où un instantané est pris	8.3.2.4
SNAP-VALUE	Pour la sémantique des instantanés, mise à jour lorsqu'un instantané est pris.	8.3.3
STATUS	Statut (**ACTIVE**, **SNAPSHOT**, **REPEAT** ou **BLOCKED**) de la commande de module ou d'un composant de test	8.3.2
STATUS	Statut (**IDLE**, **RUNNING** ou **TIMEOUT**) d'une temporisation	8.3.2.4
STATUS	Statut (**STARTED** ou **STOPPED**) d'un port	8.3.3
TC-VERDICT	Verdict de test élémentaire dans un état de module	8.3.1
TIME-LEFT	Durée qu'il reste à une temporisation avant qu'elle expire	8.3.2.4
TIMER-GUARD	Temporisation de protection pour les instructions `execute` et les opérations `call`	8.3.2
TIMER-NAME	Nom d'une temporisation	8.3.2.4
TIMER-SET	Fixation des valeurs d'une temporisation	8.3.2.5
TIMER-STATE	Etat de temporisation dans un état d'entité	8.3.2
top	Opération applicable aux piles: retourne l'item sommital d'une pile.	8.3.2.1
UPDATE-REMOTE-REFERENCES	Met à jour les temporisations et les variables ayant le même emplacement dans des entités différentes pour leur donner la même valeur	8.3.4
VALUE	Valeur d'une variable	8.3.2.2
VALUE-QUEUE	File d'attente d'un port	8.3.3
VALUE-STACK	Pile de valeurs pour le stockage des résultats d'expressions, d'opérandes, d'opérations et de fonctions.	8.3.2
VAR-NAME	Nom d'une variable	8.3.2.2
VAR-SET	Fixation de la valeur d'une variable	8.3.2.3
*****DYNAMIC-ERROR*****	Décrit l'occurrence d'une erreur dynamique	8.6.2
<identifier>	Identificateur unique d'un composant de test	8.3.2
<location>	Prend en charge les unités de portée, les paramètres par référence et les paramètres de temporisation. Représente un emplacement de stockage pour les temporisations et les variables.	8.3.2.2, 8.3.2.4

10.2 Mots clés spéciaux

Mot clé	Description	Paragraphe
ACTIVE	*STATUS* d'un état d'entité	8.3.2
BLOCKED	*STATUS* d'un état d'entité	8.3.2
IDLE	*STATUS* d'un état de temporisation	8.3.2.4
MARK	Utilisé comme marque pour *VALUE-STACK*	8.3.2
NONE	Utilisé pour décrire une valeur non définie	8.3.2.3, 8.3.2.5, 8.3.3.2
NULL	Valeur symbolique pour les types pointeur et assimilés pour indiquer que rien n'est adressé	8.3.1.1, 8.3.2.1, 8.3.3, 8.3.3.2, 8.6.1.1
REPEAT	*STATUS* d'un état d'entité	8.3.2
RUNNING	*STATUS* d'un état de temporisation	8.3.2.4
SNAPSHOT	*STATUS* d'un état d'entité	8.3.2
STARTED	*STATUS* d'un port	8.3.3
STOPPED	*STATUS* d'un port	8.3.3
TIMEOUT	*STATUS* d'un état de temporisation	8.3.2.4

10.3 Graphes de flux des descriptions de comportement TTCN-3

	Référence	
	Figure	Paragraphe
Commande de module	18	8.2.2
Tests élémentaires	19	8.2.3
Fonctions	20	8.2.4
Variantes	21	8.2.5
Définitions de type de composant	22	8.2.6

10.4 Segments de graphe de flux

Identificateur	Construction TTCN-3 associée	Référence	
		Figure	Paragraphe
<action-stmt>	instruction **action**	36	9.1
<activate-stmt>	instruction **activate**	37	9.2
<alt-stmt>	instruction **alt**	38	9.3
<altstep-call>	invocation d'une variante	44	9.4
<altstep-call-branch>	instruction **alt**	41	9.3.3
<assignment-stmt>	instruction **assignment**	45	9.5
<b-call-with-duration>	opération **call**	52	9.6.4
<b-call-without-duration>	opération **call**	51	9.6.3
<blocking-call-op>	opération **call**	47	9.6
<call-op>	opération **call**	46	9.6
<call-reception-part>	opération **call**	53	9.6.5
<catch-op>	opération **catch**	55	9.7
<catch-timeout-exception>	opération **call**	54	9.6.6
<check-op>	opération **check**	56	9.8
<check-with-sender>	opération **check**	57	9.8.1
<check-without-sender>	opération **check**	58	9.8.2
<clear-port-op>	opération **clear** applicable aux ports	59	9.9
<connect-op>	opération **connect**	60	9.10
<constant-definition>	définition de **constant**	61	9.11
<create-op>	opération **create**	62	9.12
<deactivate-stmt>	instruction **deactivate**	63	9.13
<default-evocation>	instruction **alt**	43	9.3.5
<disconnect-op>	opération **disconnect**	64	9.14
<do-while-stmt>	instruction **do-while**	65	9.15
<done-component-op>	opération **done** applicable aux composants	66	9.16
<else-branch>	instruction **alt**	42	9.3.4
<execute-stmt>	instruction **execute**	67	9.17
<execute-timeout>	instruction **execute**	69	9.17.2
<execute-without-timeout>	instruction **execute**	68	9.17.1
<expression>	expression	70	9.18
<finalize-component-init>	utilisé dans les définitions de type de composant	75	9.19
<for-stmt>	instruction **for**	79	9.23
<func-op-call>	expression	73	9.18.3
<function-call>	appel d'une fonction	80	9.24
<getcall-op>	opération **getcall**	86	9.25
<getreply-op>	opération **getreply**	87	9.26
<getverdict-op>	opération **getverdict**	88	9.27
<goto-stmt>	instruction **goto**	89	9.28

Identificateur	Construction TTCN-3 associée	Référence	
		Figure	Paragraphe
<if-else-stmt>	instruction `if-else`	90	9.29
<init-component-scope>	utilisé dans les définitions de type de composant	76	9.20
<label-stmt>	instruction `label`	91	9.30
<lit-value>	expression	71	9.18.1
<log-stmt>	instruction `log`	92	9.31
<map-op>	opération `map`	93	9.32
<mtc-op>	opération `mtc`	94	9.33
<nb-call-without-receiver>	opération `call`	50	9.6.2
<nb-call-with-receiver>	opération `call`	49	9.6.1
<non-blocking-call-op>	opération `call`	48	9.6
<operator-appl>	expression	74	9.18.4
<parameter-handling>	traitement des paramètres de fonctions, de variantes et de tests élémentaires.	77	9.21
<port-declaration>	déclaration de port	95	9.34
<predef-ext-func-call>	appel d'une fonction (appel d'une fonction prédéfinie ou externe)	85	9.24.5
<raise-op>	opération `raise`	96	9.35
<raise-with-receiver-op>	opération `raise`	97	9.35.1
<raise-without-receiver-op>	opération `raise`	98	9.35.2
<read-timer-op>	opération `read` applicable aux temporisations	99	9.36
<receive-assignment>	opération `receive`	103	9.37.3
<receive-op>	opération `receive`	100	9.37
<receive-with-sender>	opération `receive`	101	9.37.1
<receive-without-sender>	opération `receive`	102	9.37.2
<receiving-branch>	instruction `alt`	40	9.3.2
<ref-par-var-calc>	appel d'une fonction (traitement des paramètres par référence)	82	9.24.2
<ref-par-timer-calc>	appel d'une fonction (traitement des paramètres de temporisation)	83	9.24.3
<repeat-stmt>	instruction `repeat`	104	9.38
<reply-op>	opération `reply`	105	9.39
<reply-with-receiver-op>	opération `reply`	106	9.39.1
<reply-without-receiver-op>	opération `reply`	107	9.39.2
<return-stmt>	instruction `return`	108	9.40
<return-with-value>	instruction `return`	109	9.40.1
<return-without-value>	instruction `return`	110	9.40.2
<running-component-op>	opération `running` applicable aux composants	111	9.41
<running-comp-act>	opération `running` applicable aux composants	112	9.41.1
<running-comp-snap>	opération `running` applicable aux composants	113	9.41.2
<running-timer-op>	opération `running` applicable aux temporisations	114	9.42
<self-op>	opération `self`	115	9.43
<send-op>	opération `send`	116	9.44
<send-with-receiver-op>	opération `send`	117	9.44.1
<send-without-receiver-op>	opération `send`	118	9.44.2
<setverdict-op>	opération `setverdict`	119	9.45
<start-component-op>	opération `start` applicable aux composants	120	9.46
<start-port-op>	opération `start` applicable aux ports	121	9.47
<start-timer-op>	opération `start` applicable aux temporisations	122	9.48
<start-timer-op-default>	opération `start` applicable aux temporisations	123	9.48.1
<start-timer-op-duration>	opération `start` applicable aux temporisations	124	9.48.2
<statement-block>	bloc d'instructions dans des instructions composites	78	9.22

Identificateur	Construction TTCN-3 associée	Référence	
		Figure	Paragraphe
<stop-component-op>	opération **stop** applicable aux composants	125	9.49
<stop-mtc>	opération **stop** applicable aux composants (arrêt du composant MTC)	126	9.49.1
<stop-component>	opération **stop** applicable aux composants (arrêt d'un seul composant de test)	127	9.49.2
<stop-all-comp>	opération **stop** applicable aux composants (arrêt de tous les composants)	128	9.49.3
<stop-exec-stmt>	instruction d'exécution **stop**	129	9.50
<stop-control>	instruction d'exécution **stop** (arrêt de la commande de module)	130	9.50.1
<stop-port-op>	opération **stop** applicable aux ports	131	9.51
<stop-timer-op>	opération **stop** applicable aux temporisations	132	9.52
<system-op>	opération **system**	133	9.53
<take-snapshot>	instruction **alt**	39	9.3.1
<timeout-timer-op>	opération **timeout**	137	9.55
<timer-declaration>	déclaration de temporisation	134	9.54
<timer-decl-default>	déclaration de temporisation	135	9.54.1
<timer-decl-no-def>	déclaration de temporisation	136	9.54.2
<timeout-timer-op>	opération **timeout**	137	9.55
<unmap-op>	opération **unmap**	138	9.56
<user-def-func-call>	appel d'une fonction (appel d'une fonction définie par l'utilisateur)	84	9.24.4
<value-par-calculation>	appel d'une fonction (traitement des paramètres par valeur)	81	9.24.1
<var-declaration-init>	déclaration de variable	140	9.57.1
<var-declaration-undef>	déclaration de variable	141	9.57.2
<var-value>	expression	72	9.18.2
<variable-declaration>	déclaration de variable	139	9.57
<while-stmt>	instruction **while**	140	9.58

SÉRIES DES RECOMMANDATIONS UIT-T

Série A	Organisation du travail de l'UIT-T
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	Gestion des télécommunications y compris le RGT et maintenance des réseaux
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, communication entre systèmes ouverts et sécurité
Série Y	Infrastructure mondiale de l'information, protocole Internet et réseaux de prochaine génération
Série Z	**Langages et aspects généraux logiciels des systèmes de télécommunication**

Imprimé en Suisse
Genève, 2007