Union internationale des télécommunications

UIT-T

Z.140

(03/2006)

SECTEUR DE LA NORMALISATION
DES TÉLÉCOMMUNICATIONS
DE L'UIT

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

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

---

Notation de test et de commande de test version 3 (TTCN-3): langage noyau

Recommandation UIT-T Z.140

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.140

Notation de test et de commande de test version 3 (TTCN-3): langage noyau

Résumé

La présente Recommandation définit la notation de test et de commande de test version 3 (TTCN-3, *testing and test control notation 3*) qui est destinée à la spécification de suites de tests qui soient indépendantes des plates-formes, des méthodes de test, des couches protocolaires et des protocoles. La notation TTCN-3 peut servir à la spécification de tous les types de test de système réactif, effectués en divers ports de communication. Les domaines d'application typiques ont trait aux tests des protocoles (y compris les protocoles de communications mobiles et Internet), aux tests des services (y compris les services complémentaires), aux tests des modules ainsi qu'aux tests des plates-formes en architecture CORBA et des interfaces API. La spécification de suites de tests pour protocoles de la couche Physique est hors du domaine d'application de la présente Recommandation.

Le langage noyau de la notation TTCN-3 peut être exprimé dans divers formats de présentation. Alors que la présente Recommandation définit le langage noyau, la Rec. UIT-T Z.141 définit le format de présentation tabulaire en notation TTCN (TFT, *tabular format for TTCN*) et la Rec. UIT-T Z.142 définit le format de présentation graphique en notation TTCN (GFT, *graphical format for TTCN*). La spécification de ces formats est hors du domaine d'application de la présente Recommandation. Le langage noyau répond à trois objets:

1) constituer un langage de test généralisé en mode alphanumérique;
2) constituer un format d'échange normalisé de suites de tests en notation TTCN entre utilitaires TTCN;
3) constituer la base sémantique (et, le cas échéant, la base syntaxique) des divers formats de présentation.

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

Source

La Recommandation UIT-T Z.140 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 normatives 1
3 Définitions et abréviations 2
3.1 Définitions 2
3.2 Abréviations 4
4 Introduction 4
4.0 Généralités 4
4.1 Langage noyau et formats de présentation 5
4.2 Unicité de la spécification 6
4.3 Conformité 6
5 Eléments de langage fondamentaux 6
5.0 Généralités 6
5.1 Séquencement des éléments de langage 7
5.2 Paramétrage 8
5.3 Règles de portée 11
5.4 Identificateurs et mots clés 13
6 Types et valeurs 13
6.0 Généralités 13
6.1 Types et valeurs de base 14
6.2 Sous-typage de types de base 16
6.3 Types et valeurs structurés 19
6.4 Le type anytype 26
6.5 Matrices 26
6.6 Types récursifs 28
6.7 Compatibilité des types 28
7 Modules 32
7.0 Généralités 32
7.1 Nommage des modules 32
7.2 Paramètres de module 33
7.3 Partie d'un module relative aux définitions 33
7.4 Partie d'un module relative à la commande 34
7.5 Importation à partir de modules 35
8 Configurations de test 42
8.0 Généralités 42
8.1 Modèle de communication entre ports 43
8.2 Restrictions relatives aux connexions 43
8.3 Interface du système de test abstrait 46
8.4 Définition des types de port de communication 46
8.5 Définition des types de composant 47
8.6 Adressage d'entités à l'intérieur du système SUT 49
8.7 Références de composant 50
8.8 Définition de l'interface du système de test 51
9 Déclaration des constantes 51
10 Déclaration de variables 52
10.0 Généralités 52
10.1 Variables de valeur 52
10.2 Variables de modèle 52
11 Déclaration des temporisations 53
11.0 Généralités 53
11.1 Temporisations utilisées comme paramètres 53
12 Déclaration des messages 54

Page

13 Déclaration des signatures de procédure ... 54
13.0 Généralités ... 54
13.1 Signatures pour communications bloquantes et non bloquantes ... 54
13.2 Paramètres des signatures de procédure ... 54
13.3 Procédures distantes retournant une valeur ... 54
13.4 Spécification des exceptions ... 55

14 Déclaration des modèles ... 55
14.0 Généralités ... 55
14.1 Déclaration des modèles de message ... 56
14.2 Déclaration des modèles de signature ... 57
14.3 Mécanismes d'appariement de modèles ... 58
14.4 Paramétrage de modèles ... 62
14.5 Vide ... 63
14.6 Modèles modifiés ... 63
14.7 Modification des champs de modèle ... 64
14.8 Opération d'appariement ... 64
14.9 Opération de valuation ... 65

15 Opérateurs ... 65
15.0 Généralités ... 65
15.1 Opérateurs arithmétiques ... 66
15.2 Opérateurs de concaténation ... 67
15.3 Opérateurs relationnels ... 67
15.4 Opérateurs logiques ... 69
15.5 Opérateurs binaires ... 69
15.6 Opérateurs de décalage ... 70
15.7 Opérateurs de rotation ... 70

16 Fonctions et variantes ... 71
16.1 Fonctions ... 71
16.2 Variantes ... 75
16.3 Fonctions et variantes pour des types différents de composant ... 77

17 Tests élémentaires ... 77
17.0 Généralités ... 77
17.1 Paramétrage de tests élémentaires ... 77

18 Aperçu général des instructions de programmation et des opérations ... 78

19 Expressions et instructions de programmation de base ... 80
19.0 Généralités ... 80
19.1 Expressions ... 81
19.2 Affectations ... 81
19.3 L'instruction "Log" ... 81
19.4 L'instruction "Label" ... 83
19.5 L'instruction "Goto" (saut) ... 83
19.6 L'instruction "If-else" (échappement conditionnel) ... 84
19.7 L'instruction for (pour) ... 85
19.8 L'instruction "While" (tant que) ... 85
19.9 L'instruction "Do-while" (exécution tant que) ... 85
19.10 L'instruction "Stop" (arrêt d'exécution) ... 86
19.11 L'instruction "Select Case" ... 86

20 Instructions de programmation comportementales ... 87
20.0 Généralités ... 87
20.1 Comportement à options ... 87
20.2 L'instruction "Repeat" (répétition) ... 91
20.3 Comportement entrelacé ... 92
20.4 L'instruction "Return" (retour) ... 93

Page

21 Manipulation des valeurs par défaut 94
21.0 Généralités 94
21.1 Le mécanisme de comportement par défaut 94
21.2 Références de valeurs par défaut 94
21.3 L'opération Activate (activation) 95
21.4 L'opération Deactivate (désactivation) 96

22 Opération de configuration 96
22.0 Généralités 96
22.1 L'opération "Create" (création) 97
22.2 Les opérations "Connect" (connexion) et "Map" (mappage) 98
22.3 Les opérations "Disconnect" (déconnexion) et "Unmap" (démappage) 99
22.4 Les opérations "MTC", "System" et "Self" 100
22.5 L'opération "Start" (lancement de composant de test) 101
22.6 L'opération comportementale "Stop" (arrêt de composant de test) 101
22.7 L'opération "Running" (exécution) 102
22.8 L'opération "done" (fin d'exécution) 103
22.9 L'opération "Kill" (suppression de composant de test) 104
22.10 L'opération "Alive" (sous tension) 104
22.11 Opération "Killed" (supprimé) 104
22.12 Utilisation de matrices de composants 105
22.13 Résumé de l'utilisation de "any" et "all" avec des composants 106

23 Opérations de communication 106
23.0 Généralités 106
23.1 Format général des opérations de communication 107
23.2 Communication en mode message 109
23.3 Communication en mode procédure 112
23.4 L'opération "Check" (vérification) 122
23.5 Opérations de commande des ports de communication 123
23.6 Utilisation de "any" et de "all" avec des ports 124

24 Opérations de temporisation 125
24.0 Généralités 125
24.1 L'opération "Start timer" (lancement de temporisation) 125
24.2 L'opération "Stop timer" (arrêt de temporisateur) 126
24.3 L'opération "Read timer" (lecture de temporisation) 126
24.4 L'opération "Running timer" (temporisation en cours) 126
24.5 L'opération "Timeout" (expiration de temporisation) 126
24.6 Résumé de l'utilisation de "any" et de "all" avec des temporisations 127

25 Opérations de verdict de test 127
25.0 Généralités 127
25.1 Verdict de test élémentaire 127
25.2 Valeurs de verdict et règles de surécriture 128

26 Actions externes 129

27 Partie d'un module relative à la commande 129
27.0 Généralités 129
27.1 Exécution de tests élémentaires 129
27.2 Terminaison de tests élémentaires 130
27.3 Contrôle de l'exécution de tests élémentaires 130
27.4 Sélection de tests élémentaires 130
27.5 Utilisation de temporisateurs dans les commandes 131

28 Spécification des attributs 131
28.0 Généralités 131
28.1 Attributs d'affichage 132
28.2 Codage de valeurs 132

Page

28.3 Attributs d'extension 134
28.4 Portée des attributs 134
28.5 Règles d'écrasement pour attributs 135
28.6 Modification des attributs d'éléments de langage importés 137

Annexe A – Formalisme BNF et sémantique statique 138
A.1 Formalisme BNF de la notation TTCN-3 138

Annexe B – Appariement de valeurs entrantes 156
B.1 Mécanismes d'appariement de modèles 156

Annexe C – Fonctions de notation TTCN-3 prédéfinies 166
C.0 Procédures générales de traitement d'exception 166
C.1 Conversion d'entier en caractère 166
C.2 Conversion de caractère en entier 166
C.3 Conversion d'entier en caractère universel 166
C.4 Conversion de caractère universel en entier 166
C.5 Conversion de chaîne binaire en entier 166
C.6 Conversion de chaîne hexadécimale en entier 166
C.7 Conversion de chaîne d'octets en entier 167
C.8 Conversion de chaîne de caractères en entier 167
C.9 Conversion d'entier en chaîne binaire 167
C.10 Conversion d'entier en chaîne hexadécimale 167
C.11 Conversion d'entier en chaîne d'octets 167
C.12 Conversion d'entier en chaîne de caractères 168
C.13 Longueur de type chaîne 168
C.14 Nombre d'éléments contenus dans une valeur structurée 168
C.15 La fonction "IsPresent" 169
C.16 La fonction "IsChosen" 169
C.17 La fonction "Regexp" 169
C.18 Conversion de chaîne binaire en chaîne de caractères 170
C.19 Conversion de chaîne hexadécimale en chaîne de caractères 170
C.20 Conversion de chaîne d'octets en chaîne de caractères 170
C.21 Conversion de chaîne de caractères en chaîne d'octets 170
C.22 Conversion de chaîne binaire en chaîne hexadécimale 171
C.23 Conversion de chaîne hexadécimale en chaîne d'octets 171
C.24 Conversion de chaîne binaire en chaîne d'octets 171
C.25 Conversion de chaîne hexadécimale en chaîne binaire 171
C.26 Conversion de chaîne d'octets en chaîne hexadécimale 172
C.27 Conversion de chaîne d'octets en chaîne binaire 172
C.28 Conversion d'entier en nombre à virgule flottante 172
C.29 Conversion de nombre à virgule flottante en entier 172
C.30 La fonction de générateur de nombre aléatoire 172
C.31 La fonction de sous-chaîne 173
C.32 Nombre d'éléments dans un type structuré 173
C.33 Conversion de chaîne de caractères en nombre à virgule flottante 173
C.34 La fonction "Replace" (remplacement) 174
C.35 Conversion de chaîne d'octets en chaîne de caractères 174
C.36 Conversion de chaîne de caractères en chaîne d'octets 174

Annexe D (informative) – Vide 175

Annexe E (informative) – Bibliothèque de types utiles 176
E.1 Limitations 176
E.2 Types utiles en notation TTCN-3 176

Annexe F (informative) – Opérations sur des objets actifs TTCN-3 180
F.1 Généralités 180
F.2 Composants de test 180

Page

F.3 Temporisations 183
F.4 Ports 184

Annexe G (informative) – Caractéristiques de langage déconseillées 187
G.1 Définition des paramètres de module fondée sur des groupes 187
G.2 Importation récursive 187
G.3 Utilisation du mot clé `all` dans les définitions de type de port 187

BIBLIOGRAPHIE 188

Recommandation UIT-T Z.140

Notation de test et de commande de test version 3 (TTCN-3): langage noyau

1 Domaine d'application

La présente Recommandation définit le langage noyau de la notation TTCN-3, laquelle peut servir à la spécification de tous les types de test de système réactif, effectués en divers ports de communication. Les domaines d'application typiques ont trait aux tests des protocoles (y compris les protocoles de communications mobiles et Internet), aux tests des services (y compris les services complémentaires), aux tests des modules, aux tests des plates-formes en architecture CORBA, aux tests des interfaces API etc. La notation TTCN-3 n'est pas limitée aux tests de conformité et peut servir à de nombreuses autres sortes de tests, y compris ceux d'interopérabilité, de robustesse, de régression, de système et d'intégration. La spécification de suites de tests pour protocoles de couche Physique est hors du domaine d'application de la présente Recommandation.

La notation TTCN-3 est destinée à permettre la spécification de suites de tests qui soient indépendantes des méthodes de test, des couches et des protocoles. Divers formats de présentation sont définis pour la notation TTCN-3, comme un format de présentation tabulaire (Rec. UIT-T Z.141 [1]) et un format de présentation graphique (Rec. UIT-T Z.142 [2]). La spécification de ces formats est hors du domaine d'application de la présente Recommandation.

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

2 Références normatives

La présente Recommandation se réfère à certaines dispositions des Recommandations UIT-T et textes suivants qui, de ce fait, en sont partie intégrante. Les versions indiquées étaient en vigueur au moment de la publication de la présente Recommandation. Toute Recommandation ou tout texte étant sujet à révision, les utilisateurs de la présente Recommandation sont invités à se reporter, si possible, aux versions les plus récentes des références normatives suivantes. La liste des Recommandations de l'UIT-T en vigueur est régulièrement publiée. La référence à un document figurant dans la présente Recommandation ne donne pas à ce document, en tant que tel, le statut d'une Recommandation.

[1] Recommandation UIT-T Z.141 (2006), *Notation de test et de commande de test version 3 (TTCN-3): format de présentation tabulaire*.

[2] Recommandation UIT-T Z.142 (2006), *Notation de test et de commande de test version 3 (TTCN-3): format de présentation graphique*.

[3] Recommandation UIT-T Z.143 (2006), *Notation de test et de commande de test version 3 (TTCN-3): sémantique opérationnelle*.

[4] Recommandation UIT-T Z.144 (2006), *Notation de test et de commande de test version 3 (TTCN-3): interface d'exécution*.

[5] Recommandation UIT-T Z.145 (2006), *Notation de test et de commande de test version 3 (TTCN-3): interface de commande*.

[6] Recommandation UIT-T Z.146 (2006), *Notation de test et de commande de test version 3 (TTCN-3): utilisation de la notation ASN.1 avec la notation TTCN-3*.

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

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

[8] Recommandation UIT-T X.292 (2002), *Cadre et méthodologie des tests de conformité OSI pour les Recommandations sur les protocoles pour les applications de l'UIT-T – Notation combinée arborescente et tabulaire (TTCN)*.

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

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

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

[10] ISO/CEI 10646:2003, *Technologies de l'information – Jeu universel de caractères codés sur plusieurs octets (JUC)*.

[11] ISO/CEI 6429:1992, *Technologies de l'information – Fonctions de commande pour les jeux de caractères codés*.

3 Définitions et abréviations

3.1 Définitions

Dans la présente Recommandation, les termes et définitions figurant dans les Recommandations UIT-T X.290 [7] et X.292 [8] s'appliquent en plus de ce qui suit:

3.1.1 paramètre effectif: valeur, expression, modèle ou référence nominative (identificateur) à transmettre comme paramètre à l'entité invoquée (fonction, test élémentaire, variante **`altstep`**, etc.) comme défini au départ de l'invocation

NOTE – Le nombre, l'ordre et le type de tous les paramètres effectifs à transmettre lors d'une même invocation doivent être conformes à la liste des paramètres formels définis dans l'entité invoquée.

3.1.2 types de base: ensemble des types TTCN-3 prédéfinis qui sont décrits aux § 6.1.0 et 6.1.1

NOTE – Les types de base sont désignés par leur nom.

3.1.3 type compatible: la notation TTCN-3 n'est pas fortement typée mais ce langage nécessite quand même la compatibilité des types.

NOTE – Les variables, constantes, modèles, etc., ont des types compatibles si les conditions du § 6.7 sont satisfaites.

3.1.4 port de communication: mécanisme abstrait facilitant la communication entre composants de test

NOTE – Un port de communication est modélisé comme une file d'attente du type premier entré premier sorti dans le sens réception. Les ports peuvent utiliser le mode message, le mode procédure, ou un mélange des deux modes.

3.1.5 types de données: nom commun désignant les simples types de base, les types de chaîne de base, les types structurés, le type de données spécial "anytype" et tous les types définis par l'utilisateur sur la base des précédents (voir Tableau 3).

3.1.6 types définis (types TTCN-3 définis): ensemble de tous les types TTCN-3 prédéfinis (types de base, tous les types structurés, le type "anytype", les types adresse, port et composant et le type par défaut) ainsi que tous les types définis par l'utilisateur, déclarés dans le module ou importés à partir d'autres modules TTCN-3.

3.1.7 paramétrage dynamique: sorte de paramétrage dans laquelle les paramètres effectifs dépendent d'événements d'exécution; par exemple la valeur du paramètre effectif est reçue pendant l'exécution ou dépend d'une valeur reçue par une relation logique.

3.1.8 exception: en cas de communication en mode procédure, une exception (si elle est définie) est propagée par une entité répondante si celle-ci ne peut pas répondre à un appel de procédure distant au moyen de la réponse normalement attendue.

3.1.9 paramètre formel: référence (identificateur) de nom typé ou de modèle typé non résolue au moment de la définition d'une entité (fonction, test élémentaire, variante altstep, etc.) mais au moment de son invocation.

NOTE – Les valeurs ou modèles (ou leur nom) à utiliser effectivement à la place de paramètres formels sont transmis à partir du lieu d'invocation de l'entité (voir également la définition du paramètre effectif).

3.1.10 visibilité globale: attribut d'une entité (paramètre de module, constante, modèle, etc.) tel que son identificateur puisse être désigné n'importe où dans le module dans lequel cette entité est définie, y compris toutes les fonctions, tous les tests élémentaires et toutes les variantes définis dans le même module et dans la partie commande de ce module.

3.1.11 déclaration de conformité d'instance (ICS, *implementation conformance statement*): voir la Rec. UIT-T X.290 [7].

3.1.12 informations complémentaires sur l'instance destinées au test (IXIT, *implementation extra information for testing*): voir la Rec. UIT-T X.290 [7].

3.1.13 implémentation sous test (IUT, *implementation under test*): voir la Rec. UIT-T X.290 [7].

3.1.14 types connus: ensemble de tous les types TTCN-3 prédéfinis, des types définis dans un module TTCN-3 et des types importés dans ce module à partir d'autres modules TTCN-3 ou à partir de modules non TTCN-3.

3.1.15 partie gauche (d'une affectation): valeur ou identificateur de variable de modèle ou nom de champ d'une valeur de type structuré ou variable de modèle (avec indice de matrice, le cas échéant), situé à gauche d'un symbole d'affectation (:=)

NOTE – Une constante, un paramètre de module, un temporisateur, un nom de champ de type structuré ou un en-tête de modèle (avec type de modèle, nom et liste de paramètres formels) situés à gauche d'un symbole d'affectation (:=) dans des déclarations et/ou des définitions d'un modèle modifié n'entrent pas dans le cadre de la présente définition étant donné qu'ils ne font pas partie d'une affectation.

3.1.16 visibilité locale: attribut d'une entité (constante, variable etc.) tel que son identificateur ne puisse être désigné que dans la fonction, dans le test élémentaire ou dans la variante `altstep` où cette entité est définie.

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

3.1.18 transmission de paramètre par valeur: mode de transmission de paramètres où les arguments sont évalués avant qu'une entité paramétrable soit introduite.

NOTE – Seules les valeurs des arguments sont transmises. Les modifications apportées aux arguments dans l'entité appelée n'ont aucun effet sur les arguments effectivement perçus par l'appelant.

3.1.19 transmission de paramètre par référence: mode de transmission de paramètres où les arguments ne sont pas évalués avant que la fonction, la variante altstep, etc. soit introduite et où une référence au paramètre est transmise par la procédure appelante (fonction, variante, etc.) à la procédure appelée.

NOTE – Toutes les modifications apportées aux arguments dans la procédure appelée ont un effet sur les arguments effectivement perçus par l'appelant.

3.1.20 composant de test parallèle (PTC, *parallel test component*): voir la Rec. UIT-T X.292 [8].

3.1.21 partie droite (d'une affectation): expression, référence de modèle ou identificateur de paramètre de signature situé à droite d'un symbole d'affectation (:=).

NOTE – Les expressions et les références de modèle situées à droite d'un symbole d'affectation (:=) dans des déclarations de constante, de paramètre de module, de temporisateur, de modèle ou de modèle modifié n'entrent pas dans le cadre de la présente définition étant donné qu'elles ne font pas partie d'une affectation.

3.1.22 type racine: type de base, type structuré, type de données spécial, type de configuration spécial ou type par défaut spécial auquel le type TTCN-3 défini par l'utilisateur peut être rapporté.

3.1.23 paramétrage statique: sorte de paramétrage dans laquelle les paramètres effectifs sont indépendants des événements d'exécution; c'est-à-dire qu'ils sont connus au moment de la compilation ou, dans le cas de paramètres de module, qu'ils sont connus au début de l'exécution de la suite de tests (par exemple connus à partir de la spécification de la suite de tests, par comptage des définitions importées ou par signalisation de leur valeur au système de test avant l'instant de l'exécution).

NOTE – Tous les types sont connus au moment de la compilation, c'est-à-dire qu'ils sont statiquement associés.

3.1.24 typage fort: stricte application de la compatibilité des types par équivalence des noms de type, sans aucune exception.

3.1.25 système sous test (SUT, *system under test*): voir la Rec. UIT-T X.290 [7].

3.1.26 modèle: les modèles TTCN-3 sont des structures de données spécifiques pour les tests; ils servent soit à transmettre un ensemble de valeurs distinctes ou à vérifier si un ensemble de valeurs reçues correspond au modèle spécifié.

3.1.27 comportement de test (ou comportement): test élémentaire ou fonction lancée sur un composant de test durant l'exécution d'une instruction `execute` ou `start` avec appel récursif de toutes les fonctions et variantes.

NOTE – Durant l'exécution d'un test élémentaire, les composants de chaque test ont leur propre comportement ce qui explique que plusieurs comportements de test peuvent être mis en œuvre simultanément dans le système de test (en d'autres termes, un test élémentaire peut être vu comme un ensemble de comportements de test).

3.1.28 test élémentaire: voir la Rec. UIT-T X.290 [7].

3.1.29 erreur de test élémentaire: voir la Rec. UIT-T X.290 [7].

3.1.30 suite de tests: ensemble de modules TTCN-3 qui contient un ensemble de tests élémentaires complètement défini, complété à titre facultatif d'une ou de plusieurs parties commande TTCN-3.

3.1.31 système de test: voir la Rec. UIT-T X.290 [7].

3.1.32 interface avec le système de test: composant de test qui assure un mappage des ports disponibles dans le système de test (abstrait) TTCN-3 vers les ports offerts par le système SUT.

3.1.33 compatibilité des types: élément de langage qui permet d'employer des valeurs, des expressions ou des modèles d'un type donné comme valeurs effectives d'un autre type (par exemple lors d'affectations, comme paramètres effectifs lors de l'appel d'une fonction, lors de la désignation d'un modèle etc., ou comme valeur de retour d'une fonction).

NOTE – Le type et la valeur, l'expression ou le modèle actuel doivent être compatibles avec l'autre type.

3.1.34 paramétrage par valeur: capacité de transmettre une valeur ou un modèle comme paramètre effectif dans un objet paramétré.

NOTE – Ce paramètre de valeur effectif complète donc la spécification de cet objet.

3.1.35 type défini par l'utilisateur: type qui est défini par sous-typage d'un type de base ou par déclaration d'un type structuré.

NOTE – Les types définis par l'utilisateur sont désignés par leur identificateur (nom).

3.1.36 notation de valeur: notation par laquelle un identificateur est associé à une valeur ou à une étendue donnée d'un type particulier

NOTE – Les valeurs peuvent être des constantes ou des variables.

3.2 Abréviations

La présente Recommandation utilise les abréviations suivantes:

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

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

BMP table multilingue de base (*basic multilingual plane)*

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

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

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

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

ICS déclaration de conformité d'une instance (*implementation conformance statement*)

IRV version internationale de référence (*international reference version*)

IUT implémentation sous test (*implementation under test*)

IXIT informations supplémentaires sur l'implémentation destinées au test (*implementation extra information for testing*)

MTC composante de test principale (*main test component*)

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

SUT système sous test (*system under test*)

TSI interface du système de test (*test system interface*)

4 Introduction

4.0 Généralités

La notation TTCN-3 est un langage souple et puissant qui est applicable à la spécification de tous les types de tests de système réactif, à diverses interfaces de communication. Les domaines d'application typiques ont trait aux tests des protocoles (y compris les protocoles de communication mobile et Internet), aux tests des services (y compris les services complémentaires), aux tests des modules, aux tests de plates-formes en architecture CORBA, aux tests d'interface API, etc. La notation TTCN-3 n'est pas limitée aux tests de conformité et peut servir à de nombreuses autres sortes de tests, y compris ceux d'interopérabilité, de robustesse, de régression, de système et d'intégration.

La notation TTCN-3 comprend les caractéristiques essentielles ci-après:

- la capacité de spécifier des configurations de test dynamiques simultanées;
- des opérations de communication en mode procédure et en mode message;
- la capacité de spécifier des informations de codage et d'autres attributs (y compris l'extensibilité de l'utilisateur);
- la capacité de spécifier des modèles de données et de signature avec de puissants mécanismes d'appariement;
- le paramétrage par valeur;
- l'affectation et le traitement de verdicts de test;
- le paramétrage des suites de tests et des mécanismes de sélection de tests élémentaires;
- l'utilisation combinée de la notation TTCN-3 avec d'autres langages;
- des définitions correctes de la syntaxe, du format d'échange et de la sémantique statique;
- différent formats de présentation (par exemple tabulaire et graphique);
- un algorithme d'exécution précis (sémantique opérationnelle).

4.1 Langage noyau et formats de présentation

La spécification de la notation TTCN-3 est subdivisée en plusieurs parties. La première, définie dans la présente Recommandation, est le langage noyau. La deuxième, définie dans la Rec. UIT-T Z.141 [1], est le format de présentation tabulaire. La troisième partie, définie dans la Rec. UIT-T Z.142 [2], est le format de présentation graphique. La quatrième partie (Rec. UIT-T Z.143 [3]), contient la sémantique opérationnelle du langage. La cinquième partie (Rec. UIT-T Z.144 [4]), définit l'interface d'exécution TTCN-3 (TRI, *TTCN-3 runtime interface*), la sixième partie (Rec. UIT-T Z.145 [5]), définit les interfaces de commande TTCN-3 (TCI, *TTCN-3 control interface*) et la septième partie, Rec. UIT-T Z.146 [6], spécifie l'utilisation des définitions de la notation ASN.1 avec la notation TTCN-3.

Le langage noyau répond à trois objets:

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

b) constituer un format d'échange normalisé de suites de tests TTCN-3 entre utilitaires TTCN-3;

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

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

Les formats de présentation tabulaire et graphique sont les premiers d'un ensemble prévu de différents formats de présentation. Ces autres formats pourront être normalisés ou seront propres à un fournisseur, selon les définitions données par les utilisateurs de la notation TTCN-3 eux-mêmes. Ces formats additionnels ne sont pas définis dans la présente Recommandation.

La notation TTCN-3 peut facultativement être utilisée avec d'autres notations de type et de valeur, auquel cas les définitions d'autres langages pourront être utilisées sous la forme d'une variante syntaxique de type et de valeur de données. D'autres parties de la présente norme spécifient l'utilisation d'autres langages avec la notation TTCN-3. La prise en charge d'autres langages n'est pas limitée aux langages spécifiés dans les Recommandations de la série Z.140, mais pour la prise en charge de langages dont l'utilisation combinée avec la notation TTCN-3 est définie, il faut appliquer les règles énoncées dans la présente Recommandation.

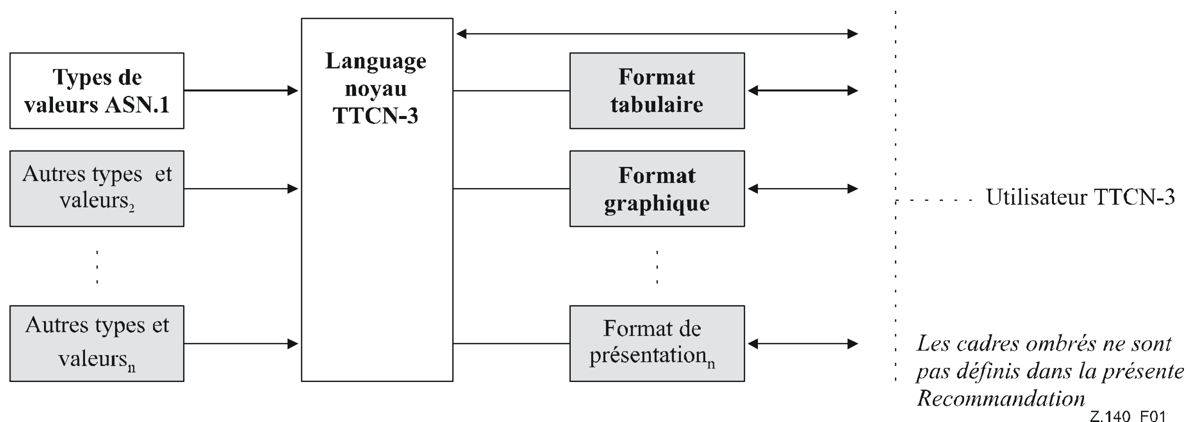


Figure 1/Z.140 – Vue d'utilisateur du langage noyau et des divers formats de présentation

Le langage noyau est défini par une syntaxe complète (voir Annexe A) et par une sémantique opérationnelle (Rec. UIT-T Z.143 [3]). Il contient une sémantique statique minimale (reproduite dans le corps de la présente Recommandation et dans son Annexe A) qui ne limite pas l'emploi du langage en raison d'un domaine d'application sous-jacent ou d'une méthode d'application sous-jacente.

4.2 Unicité de la spécification

Le langage est spécifié syntaxiquement et sémantiquement sous la forme d'une description textuelle dans le corps de la présente Recommandation (paragraphes 5 à 28) et de manière formalisée dans l'Annexe A. Dans chaque cas, quand la description textuelle n'est pas exhaustive, la description formelle la complète. Si les spécifications textuelles et formelles sont contradictoires, la spécification formelle a priorité.

4.3 Conformité

Pour une implémentation déclarée conforme à la présente version du langage, tous les éléments spécifiés dans la présente Recommandation doivent être implémentés conformément aux prescriptions énoncées dans la présente Recommandation et dans la Rec. UIT-T Z.143 [3].

5 Eléments de langage fondamentaux

5.0 Généralités

L'unité de niveau supérieur de la notation TTCN-3 est un module. Celui-ci ne peut pas être structuré en sous modules. Un module peut importer des définitions à partir d'autres modules. Ceux-ci peuvent contenir des paramètres de module de façon à permettre le paramétrage des suites de tests.

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

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

La notation TTCN-3 contient un certain nombre de types prédéfinis de données de base ainsi que des types structurés comme des enregistrements, des ensembles, des réunions, des types énumérés et des matrices.

Une sorte spéciale de structure de données, appelée modèle, offre des mécanismes de paramétrage et d'appariement permettant de spécifier des données de test à envoyer ou à recevoir aux ports de test. Les opérations effectuées en ces ports fournissent des capacités de communication aussi bien en mode message qu'en mode procédure. Des appels de procédure peuvent servir à tester des mises en œuvre qui ne sont pas en mode message.

Le comportement dynamique de test est exprimé par des tests élémentaires. Les instructions de programmation TTCN-3 comprennent de puissants mécanismes de description de comportement comme la réception en variante d'événements

de communication et de temporisation, l'entrelacement et le comportement par défaut. Les mécanismes d'affectation de verdict de test et de journalisation sont également pris en charge.

Enfin, les éléments de langage de la notation TTCN-3 peuvent se voir affecter des attributs comme ceux d'information de codage et ceux d'affichage. Il est également possible de spécifier des attributs définis par l'utilisateur (non normalisés).

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

Elément de language	Mot clé associé	Spécifié dans la partie définitions du module	Spécifié dans la partie commande du module	Spécifiés dans fonctions/ variantes/tests élémentaires	Spécifiés dans type de composant de test
Définition de module TTCN-3	**module**				
Importation de définitions à partir d'un autre module	**import**	Oui			
Groupement de définitions	**group**	Oui			
Définitions de type de donnée	**type**	Oui			
Définitions de port de communication	**port**	Oui			
Définitions de composant de test	**component**	Oui			
Définitions de signature	**signature**	Oui			
Définitions de fonction/constante externe	**external**	Oui			
Définitions de constante	**const**	Oui	Oui	Oui	Oui
Définitions de modèle de données/signature	**template**	Oui	Oui	Oui	Oui
Définitions de fonction	**function**	Oui			
Définitions de variante	**altstep**	Oui			
Définitions de test élémentaire	**testcase**	Oui			
Déclarations de variable de valeur	**var**		Oui	Oui	Oui
Déclarations de variable de modèle	**var template**		Oui	Oui	Oui
Déclarations de temporisateur	**timer**		Oui	Oui	Oui
NOTE – Les notions de "définition" et de "déclaration" de variables, de constantes, de types et d'autres éléments de langage sont utilisées de manière interchangeable dans la présente norme. La distinction entre ces deux notions n'est utile qu'à des fins d'implémentation, comme c'est le cas dans les langages de programmation de type C et C++. Au niveau de la notation TTCN-3, ces notions ont la même signification.					

5.1 Séquencement des éléments de langage

Généralement, l'ordre dans lequel les déclarations peuvent être effectuées est arbitraire. A l'intérieur d'un bloc d'instructions et de déclarations, comme un corps de fonction ou une branche d'instruction **if-else**, toutes les déclarations (éventuelles) ne doivent être effectuées qu'au début du bloc.

EXEMPLE:

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

Les déclarations contenues dans la partie d'un module relative aux définitions peuvent être énoncées dans un ordre quelconque. Toutefois, à l'intérieur de la partie d'un module relative à la commande, dans les définitions de tests élémentaires, dans les fonctions et dans les variantes, toutes les déclarations requises doivent être indiquées à l'avance.

Il s'ensuit notamment que les variables locales, les temporisateurs locaux et les constantes locales ne doivent jamais être utilisés avant d'avoir été déclarés. La seule exception à cette règle concerne les étiquettes. Des références anticipées à une étiquette peuvent être utilisées dans des instructions **`goto`** avant que l'étiquette n'apparaisse (voir § 19.5).

5.2 Paramétrage

5.2.0 Paramétrage statique et paramétrage dynamique

La notation TTCN-3 prend en charge le paramétrage *par valeur* conformément aux limitations suivantes:

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

b) l'élément de langage **`module`** permet un paramétrage *statique* par valeur afin de prendre en charge des paramètres de suite de tests; c'est-à-dire que ce paramétrage peut éventuellement être résolu en phase de compilation mais doit l'être au commencement de l'exécution (c'est-à-dire que le paramétrage doit être *statique* au moment de l'exécution). Autrement dit, à ce moment, les valeurs des paramètres du module sont visibles globalement mais ne sont pas modifiables;

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

d) les éléments de langage **`template`**, **`signature`**, **`testcase`**, **`altstep`** et **`function`** prennent en charge le paramétrage *dynamique* par valeur (c'est-à-dire que ce paramétrage doit pouvoir être résolu au moment de l'exécution).

Le Tableau 2 récapitule les éléments de langage qui peuvent être paramétrés et les valeurs qui peuvent leur être affectées comme paramètres.

Tableau 2/Z.140 – Aperçu général des éléments de langage TTCN-3 paramétrables

Mot clé	Paramétrage en valeur	Types de valeurs autorisés à apparaître dans les listes de paramètres formels/effectifs
`module`	Statique au début de l'exécution	*Valeurs de:* tous types de base, tous types définis par l'utilisateur et type **`address`**.
`type` (Note 1)	Statique au moment de la compilation	*Valeurs de:* tous types de base, tous types définis par l'utilisateur et type **`address`**.
`template`	Dynamique au moment de l'exécution	*Valeurs de:* tous types de base, tous types définis par l'utilisateur, type **`address`** et **`template`**.
`function`	Dynamique au moment de l'exécution	*Valeurs de:* tous types de base, tous types définis par l'utilisateur, type **`address`**, type **`component`**, types **`port`**, **`default`**, **`template`** et **`timer`**.
`altstep`	Dynamique au moment de l'exécution	*Valeurs de:* tous types de base, tous types définis par l'utilisateur, type **`address`**, type **`component`**, types **`port`**, **`default`**, **`template`** et **`timer`**.
`testcase`	Dynamique au moment de l'exécution	*Valeurs de:* tous types de base et tous types définis par l'utilisateur, types **`address`** et **`template`**.
`signature`	Dynamique au moment de l'exécution	*Valeurs de:* tous types de base, tous types définis par l'utilisateur et types **`address`** et **`component`**.
NOTE 1 – Les définitions de **`record of`**, **`set of`**, **`enumerated`**, **`port`**, **`component`** et **`subtype`** ne permettent pas le paramétrage. NOTE 2 – Des exemples de syntaxe et d'utilisation spécifique du paramétrage avec les différents éléments de langage sont donnés dans les paragraphes correspondants de la présente Recommandation.		

5.2.1 Transmission de paramètre par référence et par valeur

5.2.1.0 Généralités

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

Les temporisateurs et les ports sont toujours transmis par référence. Les paramètres de temporisation sont identifiés par le mot clé **`timer`**. Les paramètres de port sont identifiés par leur type de port. Le mot clé **`inout`** peut facultativement servir à indiquer une transmission par référence.

5.2.1.1 Paramètres transmis par référence

La transmission de paramètres par référence a les limitations suivantes:

a) seules les listes de paramètres formels associées à des variantes, à des fonctions, à des signatures et à des tests élémentaires peuvent contenir des paramètres transmis par référence;

   NOTE – D'autres restrictions s'appliquent à la façon d'utiliser les paramètres transmis par référence dans des signatures (voir § 23) et des variantes (voir § 16.2.1 et 21.3.1).

b) les paramètres effectifs ne doivent être que des variables de valeur ou de modèle, des ports ou des temporisateurs.

EXEMPLE:

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

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

5.2.1.2 Paramètres transmis par valeur

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

```
function MyFunction(in template MyTemplateType MyValueParameter){ … };
// MyValueParameter est transmis par valeur, le mot clé in est facultatif
```

5.2.2 Listes de paramètres formels et de paramètres effectifs

Le nombre d'éléments et l'ordre dans lequel ils apparaissent dans une liste de paramètres effectifs doivent être les mêmes que le nombre d'éléments et l'ordre dans lequel ils apparaissent dans la liste de paramètres formels correspondante. Par ailleurs, le type de chaque paramètre effectif doit être compatible avec le type de chaque paramètre formel correspondant.

EXEMPLE:

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

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

5.2.3 Liste vide de paramètres formels

Si la liste de paramètres formels des éléments de langage TTCN-3 **`function`**, **`testcase`**, **`signature`**, **`altstep`** ou **`external function`** est vide, alors les parenthèses vides doivent être incluses à la fois dans la déclaration et dans l'invocation de cet élément. Dans tous les autres cas, les parenthèses vides doivent être omises.

EXEMPLE:

```
// Une définition de fonction avec une liste vide de paramètres doit être
// écrite comme
function MyFunction(){ … }

// Une définition d'enregistrement avec une liste vide de paramètres doit
// être écrite comme
type record MyRecord { … }
```

5.2.4 Listes de paramètres imbriqués

Toutes les entités paramétrées qui sont spécifiées sous forme de paramètre effectif doivent avoir leurs propres paramètres résolus dans la liste des paramètres effectifs de niveau sommital.

EXEMPLE:

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

// Un modèle de message pourrait être
template MyMessageType MyTemplate(integer MyValue) :=
{
  field1 := MyValue,
  field2 := pattern "abc*xyz",
  field3 := true
}

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

// Quand le test élémentaire est appelé dans la partie commande et que le
// modèle paramétré est utilisé comme paramètre effectif, les paramètres
// effectifs de ce modèle doivent impérativement être fournis
control
{ :
  execute( TC001(MyTemplate(7)));
  :
}
```

5.2.5 Paramètres formels de type modèle

5.2.5.1 Paramétrage avec modèles et attributs d'appariement

Afin que l'on puisse transmettre des modèles ou des symboles d'appariement sous forme de paramètres effectifs, le mot clé supplémentaire **template** doit être ajouté avant le champ de type du paramètre formel correspondant. Le paramètre est alors de type modèle et, en fait, les paramètres autorisés pour le type associé sont élargis et englobent l'ensemble approprié d'attributs d'appariement (voir Annexe B) ainsi que l'ensemble de valeurs normal. Les champs des paramètres de modèle ne doivent pas être appelés par référence.

EXEMPLE:

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

// pourrait être utilisé comme suit
pco1.receive(MyTemplate(?));
// Ou comme suit
pco1.receive(MyTemplate(omit));
```

5.2.5.2 Eléments de langage utilisant des paramètres de type modèle

Seules les définitions de **function**, **testcase**, **altstep** et **template** peuvent avoir des paramètres formels de type modèle.

EXEMPLE:

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

5.3 Règles de portée

5.3.0 Généralités

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

a) la partie définitions d'un module;

b) la partie commande d'un module;

c) les types de composant;

d) les fonctions;

e) les variantes;

f) les tests élémentaires;

g) les "blocs d'instructions et de déclarations" contenus dans des instructions composites.

NOTE 1 – Une règle supplémentaire de portée est donnée dans le § 7.3.1 pour les groupes.

NOTE 2 – Une règle supplémentaire de portée est donnée dans le § 19.7 pour les compteurs de boucles **`for`**.

Chaque unité de portée se compose de déclarations (facultatives). Les unités de portée de la partie commande d'un module, des fonctions, des tests élémentaires, des variantes et des "blocs d'instructions et de déclarations" contenues dans des instructions composites peuvent en outre spécifier une certaine forme de comportement au moyen des instructions de programmation et opérations de la notation TTCN-3 (voir § 18).

Les définitions données dans la partie définitions d'un module mais à l'extérieur d'autres unités de portée sont visibles globalement, c'est-à-dire qu'elles peuvent être utilisées ailleurs dans le module, y compris toutes les fonctions, tous les tests élémentaires et toutes les variantes définis dans le module et dans la partie commande. Les identificateurs importés à partir d'autres modules sont également visibles globalement dans tout le module d'importation.

Les définitions données dans la partie commande d'un module ont une visibilité locale, c'est-à-dire qu'elles ne peuvent être utilisées que dans la partie commande.

Les définitions données dans un type de composant de test ne peuvent être utilisées que dans des fonctions, des tests élémentaires et des variantes faisant référence à ce type de composant ou à un type de composant de test compatible (voir § 16.3) par une clause **`runs on`**.

Les tests élémentaires, les variantes et les fonctions sont des unités de portée individuelles sans aucune relation hiérarchique entre elles, c'est-à-dire que les déclarations effectuées au début de leur corps ont une visibilité locale et ne doivent être utilisées que dans le test élémentaire, la variante altstep ou la fonction indiquée (par exemple une déclaration effectuée dans un test élémentaire n'est pas visible dans une fonction appelée par ce test élémentaire ou dans une variante altstep utilisée par ce test élémentaire).

Les instructions composites, par exemple **`if-else`**-, **`while-`**, **`do-while`**- ou **`alt`**- comprennent des "blocs d'instructions et de déclarations". Elles peuvent être utilisées dans la partie commande d'un module, dans des tests élémentaires, dans des variantes, dans des fonctions, ou peuvent être imbriquées dans d'autres instructions composites, par exemple une instruction **`if-else`** qui est utilisée dans une boucle **`while`**.

Les "blocs d'instructions et de déclarations" des instructions composites imbriquées ou non imbriquées ont une relation hiérarchique aussi bien avec l'unité de portée y compris le "bloc d'instructions et de déclarations" donné qu'avec tout "bloc d'instructions et de déclarations" imbriqué. Les déclarations effectuées dans un "bloc d'instructions et de déclarations" ont une visibilité locale.

La hiérarchie des unités de portée est représentée dans la Figure 2. Les déclarations d'une unité de portée de niveau hiérarchique supérieur sont visibles par toutes les unités situées à des niveaux inférieurs de la même branche hiérarchique. Les déclarations d'une unité de portée située dans un niveau hiérarchique inférieur ne sont pas visibles par les unités situées à un niveau hiérarchique supérieur.

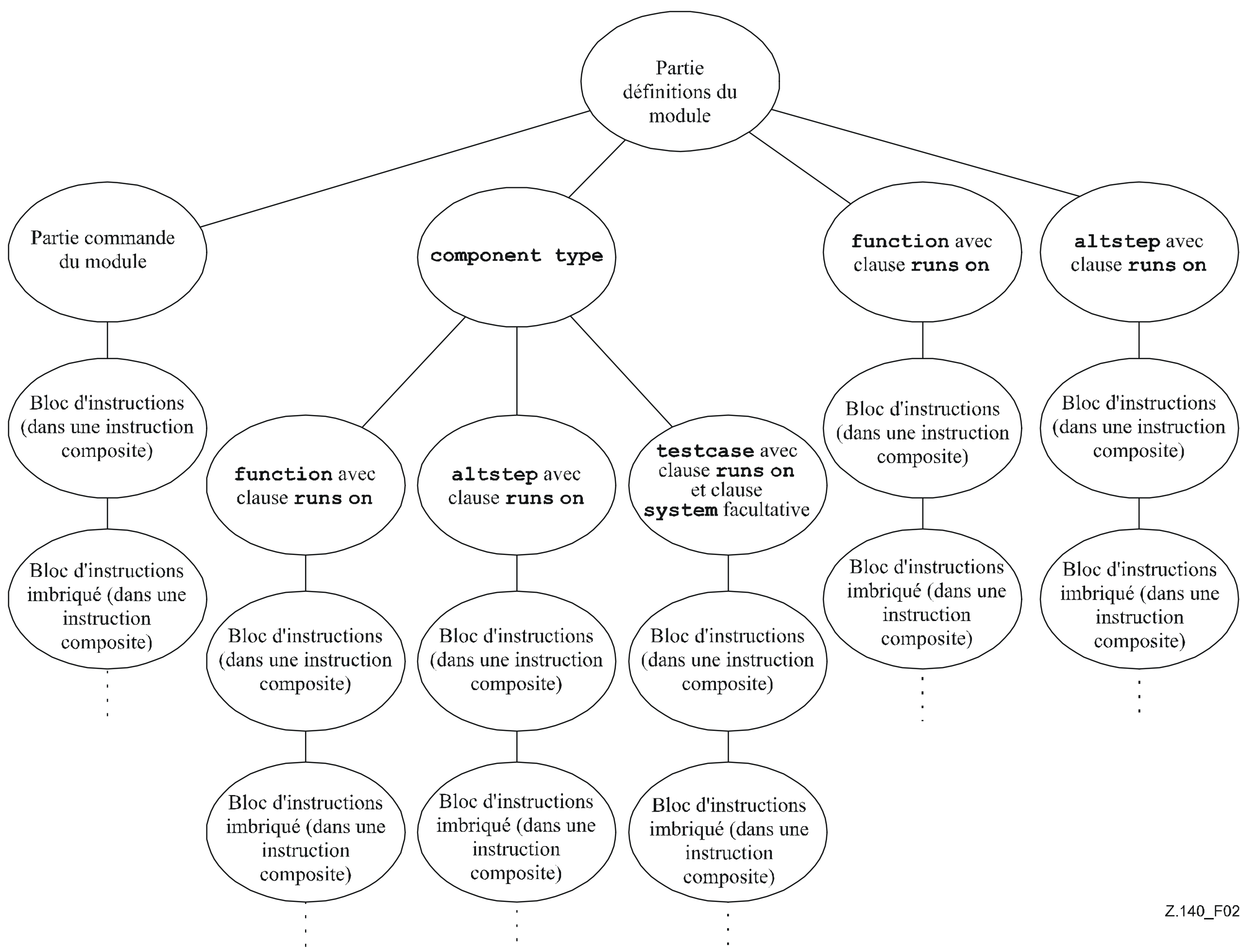


Figure 2/Z.140 – Hiérarchie des unités de portée

EXEMPLE:

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

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

   }
}
```

5.3.1 Portée des paramètres formels

La portée des paramètres formels contenus dans un élément de langage paramétré (par exemple dans un appel de fonction) doit être limitée à la définition dans laquelle ces paramètres apparaissent et aux niveaux inférieurs de portée de la même hiérarchie de portée. Autrement dit, ils suivent les règles normales de portée (voir § 5.3.0).

5.3.2 Unicité des identificateurs

La notation TTCN-3 exige l'unicité des identificateurs c'est-à-dire que tous les identificateurs contenus dans la même hiérarchie de portée doivent être différents. Autrement dit, une déclaration contenue dans un niveau inférieur de portée ne doit pas réutiliser le même identificateur qu'une déclaration située dans un niveau supérieur de portée dans la même branche de la hiérarchie de portée. Les identificateurs des champs de types structurés, de valeurs d'énumération et de groupes ne sont pas tenus d'être globalement uniques; cependant, dans le cas de valeurs d'énumération, les

identificateurs ne doivent être réutilisés que pour les valeurs d'énumération contenues dans d'autres types énumérés. Les règles d'unicité des identificateurs doivent également s'appliquer aux identificateurs des paramètres formels.

EXEMPLE:

```
module MyModule
{  :
   const integer A := 1;
   :
   function MyBehaviourA()
   {   :
      const integer A := 1; // N'est PAS autorisé
      :
      if(…)
      {    :
       const boolean A := true; // N'est PAS autorisé
       :
      }
   }
}

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

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

5.4 Identificateurs et mots clés

Les identificateurs TTCN-3 sont sensibles à l'inversion majuscules/minuscules. Les mots clés TTCN-3 doivent être écrits en minuscules uniquement (voir Annexe A). Les mots clés TTCN-3 ne doivent être utilisés ni comme identificateurs d'objets TTCN-3 ni comme identificateurs d'objets importés à partir de modules écrits dans d'autres langages.

6 Types et valeurs

6.0 Généralités

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

Le type de données spécial **`anytype`** est défini comme la réunion de tous les types de données connus et du type adresse dans un module.

Les types spéciaux associés à des configurations de test comme **`address, port`** et **`component`** peuvent servir à définir l'architecture du système de test (voir § 22).

Le type spécial **`default`** peut servir au traitement des valeurs par défaut (voir § 21).

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

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

Classe de type	Mot clé	Sous-type
Simples types de base	`integer`	étendue, liste
	`float`	étendue, liste
	`boolean`	liste
	`objid`	liste
	`verdicttype`	liste
Types de chaîne de base	`bitstring`	liste, longueur
	`hexstring`	liste, longueur
	`octetstring`	liste, longueur
	`charstring`	étendue, liste, longueur, structure
	`universal charstring`	étendue, liste, longueur, structure
Types structurés	`record`	liste (voir Note)
	`record of`	liste (voir Note), longueur
	`set`	liste (voir Note)
	`set of`	liste (voir Note), longueur
	`enumerated`	liste (voir Note)
	`union`	liste (voir Note)
Types spéciaux de données	`anytype`	liste (voir Note)
Types de configuration spéciaux	`address`	
	`port`	
	`component`	
Types par défaut spéciaux	`default`	
NOTE – Un sous-typage de ces types sous forme de liste est possible lors de la définition d'un nouveau type contraint à partir d'un type parent préexistant mais pas directement lors de la déclaration du premier type parent.		

6.1 Types et valeurs de base

6.1.0 Simples types et valeurs de base

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

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

Les valeurs de type entier (integer) doivent être indiquées par un ou plusieurs chiffres; le premier chiffre ne doit pas être zéro à moins que la valeur ne soit 0; la valeur zéro doit être représentée par un seul zéro.

b) `float`: type permettant de décrire des nombres à virgule flottante.

En règle générale, les nombres à virgule flottante peuvent être définis comme suit: $\langle mantisse \rangle \times \langle base \rangle^{\langle exposant \rangle}$,

où *<mantisse>* est un entier positif ou négatif, *<base>* un entier positif (dans la plupart des cas 2, 10 ou 16) et *<exposant>* un entier positif ou négatif.

Dans la notation TTCN-3, la notation de valeur d'un nombre à virgule flottante est limitée à une base ayant la valeur de 10. Les valeurs à virgule flottante peuvent être exprimées en utilisant deux formes de notations de valeur:

– la notation décimale avec une virgule dans une séquence de nombres comme 1,23 (qui représente 123×10^{-2}), 2,783 (c'est-à-dire 2783×10^{-3}) ou –123,456789 (qui représente $-123\,456\,789 \times 10^{-6}$); ou

– par deux nombres séparés par E, où le premier nombre spécifie la mantisse et où le second spécifie l'exposant, par exemple: 12,3E4 (qui représente 123×10^{3}) ou –12,3E-4 (qui représente -123×10^{-5}).

NOTE – A la différence de la définition générale des valeurs à virgule flottante, la mantisse de la notation de valeur TTCN-3 autorise également les nombres décimaux en plus des entiers.

c) `boolean`: type composé de deux valeurs distinctives.

Les valeurs de type booléen sont `true` et `false`.

d) vide.

e) **`verdicttype`**: type à utiliser avec des verdicts de test composés de 5 valeurs distinctives. Les valeurs de **`verdicttype`** sont **`pass`**, **`fail`**, **`inconc`**, **`none`** et **`error`**.

6.1.1 Types et valeurs de chaîne de base

La notation TTCN-3 prend en charge les types de chaîne de base ci-après:

NOTE 1 – Le terme général chaîne ou type de chaîne en notation TTCN-3 se rapporte aux mots clés **`bitstring`**, **`hexstring`**, **`octetstring`**, **`charstring`** et **`universal charstring`**.

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

Les valeurs de type **`bitstring`** correspondent à un nombre arbitraire (éventuellement zéro) de chiffres binaires: 0 et 1, précédé par une apostrophe (') et suivi par la paire de caractères 'B.

EXEMPLE 1: `'01101'B`.

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

Les valeurs de type **`hexstring`** correspondent à un nombre arbitraire (éventuellement zéro) de chiffres hexadécimaux (les lettres majuscules et minuscules peuvent être utilisées indifféremment comme chiffres hexadécimaux):

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

précédé par une apostrophe (') et suivi par la paire de caractères 'H; chaque chiffre hexadécimal sert à indiquer la valeur d'un demi-octet en représentation hexadécimale.

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

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

Les valeurs de type **`octetstring`** correspondent à un nombre arbitraire, mais pair (éventuellement zéro) de chiffres hexadécimaux (les lettres majuscules et minuscules peuvent être utilisées indifféremment comme chiffres hexadécimaux):

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

précédé par une apostrophe (') et suivi par la paire de caractères 'O; chaque chiffre hexadécimal sert à indiquer la valeur d'un demi-octet en représentation hexadécimale.

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

d) **`charstring`**: type dont les valeurs distinctives sont zéro, un ou plusieurs caractères de la Rec. UIT-T T.50 [9], conformes à la Version internationale de référence (IRV, *international reference version*) comme spécifié dans le § 8.2/T.50 [9].

NOTE 2 – La version IRV de l'Alphabet international de référence (ancien Alphabet international n° 5 – IA5), décrite dans la Rec. UIT-T T.50 [9], équivaut à la version IRV de l'ISO/CEI 646.

Les valeurs de type **`charstring`** correspondent à un nombre arbitraire (éventuellement zéro) de caractères extraits du jeu de caractères approprié, précédé et suivi par un guillemet droit (") ou sont calculées au moyen de la fonction de conversion prédéfinie int2char utilisant comme argument la valeur d'entier positif de leur codage (voir § C.1).

NOTE 3 – La fonction de conversion prédéfinie ne peut retourner que des valeurs ayant pour longueur un seul caractère.

S'il est nécessaire de définir des chaînes qui comprennent le caractère guillemet droit ("), ce caractère est représenté par une paire de guillemets droits sur la même ligne sans caractères espace intermédiaires.

EXEMPLE 4: `""abcd""` représente la chaîne littérale "`abcd`".

e) un type chaîne de caractères précédé par le mot clé **`universal`** est un type dont les valeurs distinctives sont zéro, un ou plusieurs caractères extraits de l'ISO/CEI 10646 [10].

Les valeurs de **`universal charstring`** peuvent également correspondre à un nombre arbitraire (éventuellement zéro) de caractères extraits du jeu de caractères approprié, précédé et suivi par un

guillemet droit ("), être calculées au moyen d'une fonction de conversion prédéfinie (voir § C.3) utilisant comme argument la valeur d'entier positif de leur codage, ou correspondre à un "quadruplet".

NOTE 4 – La fonction de conversion prédéfinie ne peut retourner que des valeurs ayant pour longueur un seul caractère.

S'il est nécessaire de définir des chaînes qui comprennent le caractère guillemet droit ("), ce caractère est représenté par une paire de guillemets droits sur la même ligne sans caractères espace intermédiaires.

Le "quadruplet" ne peut correspondre qu'à un seul caractère, exprimé par les valeurs décimales de son groupe, de son plan, de sa rangée et de sa cellule conformément à l'ISO/CEI 10646 [10], précédées par le mot clé **`char`,** incluses entre parenthèses et séparées par des virgules (par exemple **`char`** (0, 0, 1, 113) indique le caractère hongrois "ű"). S'il est nécessaire d'indiquer le caractère guillemet droit (") dans une chaîne affectée conformément à la première méthode (entre guillemets droits), ce caractère est représenté par une paire de guillemets droits sur la même ligne sans caractères espace intermédiaires. Ces deux méthodes peuvent être combinées en une seule notation de valeur de chaîne, au moyen de l'opérateur de concaténation.

EXEMPLE 5: l'affectation des chaînes: "le caractère Braille" & **char** (0, 0, 40, 48) & "ressemble à cela:" représente la chaîne littérale: le caractère Braille ressemble à cela: ⠰.

NOTE 5 – Les caractères de commande peuvent être indiqués au moyen de la fonction de conversion prédéfinie ou de la forme quadruplet.

Par défaut, le type **`universal charstring`** doit être conforme à la représentation codée du jeu UCS-4 qui est spécifié dans le § 14.2 de l'ISO/CEI 10646 [10].

NOTE 6 – Le jeu UCS-4 est un format de codage qui représente chaque caractère UCS sur un champ de longueur fixe de 32 bits.

Ce codage par défaut peut être neutralisé au moyen des attributs de type "variant" définis (voir § 28.2.3). Les types de chaîne de caractères utiles suivants: utf8string, bmpstring, utf16string et iso8859string, qui utilisent ces attributs, sont définis dans l'Annexe E.

6.1.2 Accès à des éléments individuels de chaîne

Les différents éléments d'un type de chaîne sont accessibles au moyen d'une syntaxe de type matrice. Il n'est possible d'accéder qu'à un seul élément de la chaîne.

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

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

EXEMPLE:

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

6.2 Sous-typage de types de base

6.2.0 Généralités

Les types définis par l'utilisateur doivent être indiqués par le mot clé **`type`**. Avec les types définis par l'utilisateur, il est possible de créer des sous-types (comme des listes, des étendues et des restrictions de longueur) à partir des types de base, des types structurés et du type **`anytype`** conformément au Tableau 3.

6.2.1 Listes de valeurs

La notation TTCN-3 permet la spécification d'une liste de valeurs distinctives pour les types de base, les types structurés et le type **`anytype`** énumérés dans le Tableau 3. Les valeurs contenues dans la liste doivent être du type racine et doivent être un sous-ensemble vrai des valeurs définies par le type racine. Le sous-type défini par cette liste limite les valeurs autorisées du sous-type à celles qui figurent dans la liste.

EXEMPLE:

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

6.2.2 Etendues

6.2.2.0 Généralités

La notation TTCN-3 permet la spécification de contraintes d'étendue pour les types **`integer, charstring, universal charstring`** et **`float`** (ou des dérivés de ces types). Pour les types **`integer`** et **`float`**, le sous-type défini par l'étendue limite les valeurs autorisées du sous-type à celles qui se trouvent dans l'étendue située entre la limite inférieure et la limite supérieure, incluses. Dans le cas des types **`charstring`** et **`universal charstring`**, l'étendue limite les valeurs autorisées pour chaque caractère distinct contenu dans les chaînes. Les limites doivent correspondre à des positions de caractère valides, conformément à la ou aux tables du jeu codé de caractères de ces types (par exemple la position donnée ne doit pas être vide). Des positions vides entre les limites inférieure et supérieure ne sont pas considérées comme étant des valeurs valides de l'étendue spécifiée.

EXEMPLE 1:

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

EXAMPLE 2:

```
type charstring MyCharString ("a" .. "z");
// Définit un type de chaîne de longueur quelconque, chaque caractère
// étant contenu dans l'étendue spécifiée
type universal charstring MyUCharString1 ("a" .. "z");
// Définit un type de chaîne de longueur quelconque, chaque caractère
// étant contenu dans l'étendue de "a" à "z" (codes de caractère de 97
// à 122), comme "abxyz");
// les chaînes contenant tout autre caractère (y compris des caractères de
// commande) comme "abc2" ne sont pas autorisées
type universal charstring MyUCharString2 (char(0, 0, 1, 111) .. char(0, 0, 1, 113));
// Définit un type de chaîne de longueur quelconque, chaque caractère
// étant contenu dans l'étendue spécifiée au moyen de la notation
// quadruplet.
```

6.2.2.1 Etendues infinies

Afin de spécifier une étendue infinie d'entiers ou de nombres à virgule flottante, le mot clé **`infinity`** peut être utilisé au lieu d'une valeur indiquant qu'il n'y a aucune limite inférieure ou supérieure. La limite supérieure doit être supérieure ou égale à la limite inférieure.

EXEMPLE:

```
type integer MyIntegerRange (-infinity .. -1);
// Tous les nombres entiers négatifs
```

NOTE – La 'valeur' de l'infini dépend de l'implémentation. L'utilisation de cet élément peut conduire à des problèmes de portabilité.

6.2.2.2 Mélange de listes et d'étendues

Vide.

NOTE – Le présent paragraphe est remplacé par le § 6.2.5.

6.2.3 Restrictions de longueur de chaîne

La notation TTCN-3 permet la spécification de restrictions de longueur relatives aux types de chaîne. Les limites de longueur sont fondées sur des unités différentes selon le type de chaîne avec lequel elles sont utilisées. Dans tous les cas, ces limites doivent correspondre à des valeurs non négatives de type **`integer`** (ou des valeurs dérivées de type **`integer`**).

EXEMPLE:

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

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

Tableau 4/Z.140 – Unités de longueur utilisées dans la spécification de longueur de champ

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

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

6.2.4 Sous-typage par séquence des types de chaîne de caractères

La notation TTCN-3 permet d'utiliser les séquences de caractères spécifiées dans le § B.1.5 pour limiter les valeurs autorisées des types **charstring** et **universal charstring**. La contrainte de type doit utiliser le mot clé **pattern** suivi d'une séquence de caractères. Toutes les valeurs indiquées par cette séquence doivent être un sous-ensemble vrai du type soumis au sous-typage.

NOTE – Le sous-typage par séquence peut être considéré comme une forme spéciale de contrainte de liste, où les membres de la liste ne sont pas définis par énumération des chaînes de caractères spécifiques mais par le biais d'un mécanisme permettant de créer les éléments de la liste.

EXEMPLE:

```
type charstring MyString (pattern "abc*xyz");
  // toutes les valeurs autorisées de MyString ont le préfixe abc et le
  // sufixe xyz.

type universal charstring MyUString (pattern "*\r\n")
  // toutes les valeurs autorisées de MyUString se terminent par CR/LF

type charstring MyString2 (pattern "abc?\q{0,0,1,113}");
  // provoque une erreur car le caractère indiqué par le quadruplet
  // {0,0,1,113} n'est pas un caractère légal du type "charstring" TTCN-3

type MyString MyString3 (pattern "d*xyz");
  // provoque une erreur car le type MyString ne contient pas de valeur
  // commençant par le caractère d
```

6.2.5 Mélange de mécanismes de sous-typage

6.2.5.1 Mélange de structures, de listes et d'étendues

Dans les définitions des sous-types **integer** et **float** (ou des dérivés de ces types), il est permis de mélanger des listes et des étendues. Le chevauchement de contraintes différentes n'est pas une erreur.

EXEMPLE 1:

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

Dans les définitions des sous-types **charstring** et **universal charstring**, il n'est pas permis de mélanger des contraintes de structure, de liste ou d'étendue.

EXEMPLE 2:

```
type charstring MyCharStr0 ('gr', 'xyz');
  // contient les chaînes de caractères gr et xyz;

type charstring MyCharStr1 ('a'..'z');
  // contient des chaînes de caractères de longueur arbitraire contenant
  // des caractères de a à z.

type charstring MyCharStr2 (pattern '[a-z]#(3,9)');
  // contient des chaînes de caractère d'une longueur de 3 à 9 caractères
  // contenant des caractères de a à z
```

6.2.5.2 Utilisation de la restriction de longueur avec d'autres contraintes

Dans les définitions des sous-types **bitstring**, **hexstring, octetstring**, des listes et une restriction de longueur peuvent être mélangées dans la même définition de sous-type.

Dans les définitions des sous-types **charstring** et **universal charstring**, il est permis d'ajouter une restriction de longueur aux contraintes contenant un sous-typage par liste, par étendue ou par structure dans la même définition de sous-type.

Lorsqu'elle est mélangée avec d'autres contraintes, la restriction de longueur doit être le dernier élément de la définition de sous-type. La restriction de longueur prend effet conjointement avec les autres mécanismes de sous-typage (c'est-à-dire que l'ensemble de valeurs du type se compose du sous-ensemble commun des ensembles de valeurs identifiés par le sous-typage par liste, par étendue ou par structure et la restriction de longueur).

EXEMPLE:

```
type charstring MyCharStr5 ('gr', 'xyz') length (1..9);
  // contient les chaînes de caractères gr et xyz;

type charstring MyCharStr6 ( 'a'..'z') length (3.9);
  // contient des chaînes de caractères d'une longueur de 3 à 9 caractères
  // de l'ensemble de caractères de a à z.

type charstring MyCharStr7 (pattern '[a-z]#(3,9)') length (1..9);
  // contient des chaînes de caractères d'une longueur de 3 à 9 caractères
  // de l'ensemble de caractères de a à z.

type charstring MyCharStr8 (pattern '[a-z]#(3,9)') length (1..8);
  // contient des chaînes de caractères d'une longueur de 3 à 8 caractères
  // de l'ensemble de caractères de a à z.

type charstring MyCharStr9 (pattern '[a-z]#(1,8)') length (1..9);
  // contient des chaînes de caractères d'une longueur de 1 à 8 caractères
  // de l'ensemble de caractères de a à z.

type charstring MyCharStr10 ('gr', 'xyz') length (4);
  // ne contient aucune valeur (type vide).
```

6.3 Types et valeurs structurés

6.3.0 Généralités

Le mot clé **type** sert également à spécifier des types structurés comme les suivants: **record**, **record of**, **set**, **set of**, **enumerated** et **union**.

Les valeurs de ces types peuvent être indiquées au moyen d'une notation par affectation explicite ou d'une notation à liste de valeurs abrégées.

EXEMPLE 1:

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

// Ou
const MyRecordType MyRecordValue:= {'11001'B, true, "A string"}
// notation à liste de valeurs
```

Lors de la spécification de valeurs partielles (c'est-à-dire lors du réglage de la valeur de seulement un sous-ensemble des champs d'une variable structurée) au moyen de la notation par affectation, seuls les champs qui doivent recevoir des valeurs doivent impérativement être spécifiés. Les champs non mentionnés sont implicitement non spécifiés. Il est également possible d'indiquer explicitement que des champs sont non spécifiés, en utilisant le symbole de non-utilisation "-". Si l'on utilise la notation à liste de valeurs, tous les champs contenus dans la structure doivent être spécifiés, soit avec une valeur, à savoir le symbole de non-utilisation "-", soit avec le mot clé **omit**.

EXEMPLE 2:

```
var MyRecordType MyVariable:=          // notation par affectation
{
  field1 := '11001'B,
  // le champ field2 est implicitement non spécifié
  field3 := "A string"
}
```

```
// Ou:
var MyRecordType MyVariable:=            // notation par affectation
{
  field1 := '11001'B,
  field2 := -, // le champ field2 est implicitement non spécifié
  field3 := "A string"
}

// Ou:
var MyRecordType MyVariable:= {'11001'B, -, "A string"}  // notation à liste de valeurs
```

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

EXEMPLE 3:

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

Dans les deux notations – par affectation et à liste de valeurs – les champs facultatifs doivent être omis au moyen de la valeur explicite d'omission **omit** dans le champ approprié. Le mot clé **omit** ne doit pas être utilisé pour des champs obligatoires. En cas de réaffectation d'une valeur déjà initialisée, l'utilisation du symbole de non-utilisation ou le saut d'un champ dans une notation par affectation fera que les champs considérés demeureront inchangés.

EXEMPLE 4:

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

MyVariable := { '10111'B, -, - };
// MyVariable contient alors { '10111'B, false /* unchanged */, <undefined> }

MyVariable :=
{
  field2 := true
}
// MyVariable contient alors { '10111'B, true, <undefined> }

MyVariable :=
{
  field1 := -,
  field2 := false,
  field3 := -
}
// MyVariable contient alors { '10111'B, false, <undefined> }
```

6.3.1 Types et valeurs d'enregistrement

6.3.1.0 Généralités

La notation TTCN-3 prend en charge les types structurés et ordonnés qui sont désignés par le mot clé **record**. Les éléments d'un type **record** peuvent être l'un quelconque des types de base ou les types de données définis par l'utilisateur (comme d'autres enregistrements, des ensembles ou des matrices). Les valeurs d'un type **record** doivent être compatibles avec le type des champs **record**. Les identificateurs d'élément sont localisés dans le type **record** et doivent y être uniques (mais n'ont pas à être globalement uniques). Une constante qui est de type **record** ne doit contenir aucune variable (y compris les paramètres de module) en tant que valeur de champ, soit directement soit indirectement.

EXEMPLE 1:

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

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

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

EXEMPLE 2:

```
type record MyEmptyRecord { }
```

Une valeur de type **`record`** est affectée sur la base de chaque élément individuel. L'ordre des valeurs de champ dans la notation à liste de valeurs doit être identique à l'ordre des champs dans la définition de type correspondante.

EXEMPLE 3:

```
var integer MyIntegerValue := 1;

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

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

Valeur identique spécifiée avec une liste de valeurs.

EXEMPLE 4:

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

6.3.1.1 Référencement des champs de type `record`

Les éléments d'un type **`record`** doivent être désignés par la notation à points *`TypeOrValueId.ElementId`*, où *`TypeOrValueId`* correspond au nom d'un type structuré ou à une variable. L'identificateur ElementId doit correspondre au nom d'un champ contenu dans un type structuré.

EXEMPLE:

```
MyVar1 := MyRecord1.myElement1;
// Si un enregistrement est imbriqué dans un autre type, alors la référence
// peut ressembler à ce qui suit:
MyVar2 := MyRecord1.myElement1.myElement2;
```

6.3.1.2 Eléments facultatifs dans un enregistrement

Les éléments facultatifs dans un enregistrement **`record`** doivent être spécifiés au moyen du mot clé **`optional`**.

EXEMPLE 1:

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

Les champs facultatifs doivent être omis au moyen du symbole d'omission.

EXEMPLE 2:

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

// Il est à noter que cela ne revient pas à écrire:
// MyRecordValue:= {MyIntegerValue, -, "A string"};
// ce qui signifierait que la valeur de field2 est inchangée
```

6.3.1.3 Définition de types de champ imbriqués

La notation TTCN-3 prend en charge la définition de types de champ d'enregistrement imbriqués dans la définition d'un enregistrement **`record`**. La définition de nouveaux types structurés (**`record`**, **`set`**, **`enumerated`**, **`set of`** et **`record of`**) et la spécification de contraintes de sous-type sont toutes deux possibles.

EXEMPLE:

```
// définition de type d'enregistrement avec types structurés imbriqués
type record MyNestedRecordType
{
  record
  {
     integer nestedField1,
     float   nestedField2
} outerField1,
enumerated {
     nestedEnum1,
     nestedEnum2
  } outerField2,
  record of boolean outerField3
}

// définition de type d'enregistrement avec sous-types imbriqués
type record MyRecordTypeWithSubtypedFields
{
integer    field1 (1 .. 100),
charstring field2 length ( 2 .. 255 )
}
```

6.3.2 Types et valeurs d'ensemble

6.3.2.0 Généralités

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

EXEMPLE:

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

Les identificateurs de champ sont localisés dans l'ensemble et doivent y être uniques (mais sans avoir à être globalement uniques).

La notation à liste de valeurs ne doit pas servir au réglage des valeurs de type **`set`**.

6.3.2.1 Référence aux champs d'un type ensemble

Les éléments d'un ensemble doivent être désignés par la notation à points (voir § 6.3.1.1).

EXEMPLE:

```
MyVar3 := MySet1.myElement1;
// Si un ensemble est imbriqué dans un autre type, alors la référence peut
// ressembler à ce qui suit
MyVar4 := MyRecord1.myElement1.myElement2;
// Il est à noter que le type ensemble, dont le champ désigné par
// l'identificateur 'myElement2' est référencé, est imbriqué dans un type
// enregistrement
```

6.3.2.2 Eléments facultatifs dans un ensemble

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

6.3.2.3 Définition de types de champ imbriqués

La notation TTCN-3 prend en charge la définition de types de champ d'ensemble imbriqués dans la définition d'un ensemble **`set`**, selon un mécanisme analogue à celui qui est utilisé pour les types d'enregistrement décrit dans le § 6.3.1.3.

6.3.3 Enregistrements et ensembles de types particuliers

6.3.3.0 Généralités

La notation TTCN-3 prend en charge la spécification d'enregistrements et d'ensembles dont les éléments sont tous du même type. Ces éléments sont indiqués au moyen du mot clé **`of`**. Ces enregistrements et ensembles ne possèdent pas

d'identificateurs d'élément et peuvent être considérés comme respectivement analogues à une matrice ordonnée et à une matrice non ordonnée.

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

EXEMPLE 1:

```
type record length(10) of integer MyRecordOfType;
 // est un enregistrement d'exactement 10 entiers

type record length(0..10) of integer MyRecordOfType;
 // est un enregistrement d'un maximum de 10 entiers

type record length(10..infinity) of integer MyRecordOfType;
// enregistrement d'au moins 10 entiers

type set of boolean MySetOfType; // is an unlimited set of boolean values
// ensemble illimité de valeurs booléennes

type record length(0..10) of charstring StringArray length(12);
 // enregistrement d'un maximum de 10 chaînes ayant chacune exactement
 // 12 caractères
```

La notation de valeur dans les types **record of** et **set of** doit être une notation à liste de valeurs ou une notation indexée pour un élément individuel (c'est-à-dire la même notation de valeur que pour les matrices, voir § 6.5). Il existe une exception à cette règle générale: dans le cas de la définition de modèles modifiés, l'utilisation de la notation par affectation est également autorisée (voir § 14.6.0).

Quand la notation à liste de valeurs est utilisée, la première valeur de la liste est affectée au premier élément, la seconde valeur de la liste est affectée au second élément, etc. Aucune affectation vide n'est autorisée (par exemple deux virgules, la seconde faisant suite à immédiatement la première ou seulement avec un espace blanc entre elles). Les éléments à exclure de l'affectation doivent être explicitement sautés ou omis de la liste.

Les notations indexées de valeurs peuvent être utilisées aussi bien dans la partie droite que dans la partie gauche des affectations. L'indice du premier élément doit être zéro et la valeur d'indice ne doit pas dépasser la limite imposée par le sous-typage de longueur. Si la valeur de l'élément indiqué par l'indice situé à droite d'une affectation est indéfinie, cela doit provoquer une erreur sémantique ou exécutoire. Si un opérateur d'indexation situé dans la partie gauche d'une affectation se rapporte à un élément inexistant, la valeur se trouvant dans la partie droite est affectée à cet élément et tous les éléments ayant un indice inférieur à l'indice actuel et ne possédant pas de valeur affectée sont créés avec une valeur indéfinie. Les éléments indéfinis ne sont autorisés que dans les états transitoires (alors que la valeur reste invisible). L'envoi d'une valeur de type **record of** avec des éléments indéfinis doit provoquer une erreur dynamique de test élémentaire.

EXEMPLE 2:

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

MyVar := MyRecordVar[0];   // le premier élément de la valeur 'record of' (entier 0)
                           // est affecté à MyVar

// Des valeurs indexées sont également autorisées dans la partie gauche des
// affectations:
MyRecordVar[1] := MyVar; // La value MyVar est affectée au second élément
// la valeur MyRecordVar est la suivante { 0, 0, 2, 3, 4};

// L'affectation
MyRecordVar := { 0, 1, -, 2, omit };
// fera passer la valeur de MyRecordVar à { 0, 1, 2 <unchanged>, 2};
// Il est à noter que le 3e élément sera indéfini si aucune valeur ne lui
// a été précédemment affectée.

// L'affectation
MyRecordVar[6] := 6;

// fera passer la valeur de MyRecordVar à { 0, 1, 2 , 2, <undefined>, <undefined>, 6 };
// Il est à noter que les 5e et 6e éléments (d'indice 4 et 5) n'ont eu
// aucune valeur affectée avant cette dernière affectation et sont par
// conséquent indéfinis.
```

NOTE – Cela permet de copier des valeurs de type **record of,** élément par élément, dans une boucle de type for. Par exemple, la fonction ci-dessous inverse les éléments d'une valeur de type **record of**:

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

L'imbrication des types **record of** et **set of** produira une structure de données similaire à des matrices multidimensionnelles (voir § 6.5).

EXEMPLE 3:

```
// soit:
type record of integer MyBasicRecordOfType;
type record of MyBasicRecordOfType  MyRecordOfType;

// alors, la variable myRecordOfArray aura des attributs similaires à ceux
// d'une matrice à deux dimensions:
var MyRecordOfType myRecordOfArray;
// et la référence à un élément particulier aura l'allure suivante:
// (valeur du second élément de la troisième construction //'MyBasicRecordOfType')
   myRecordOfArray [2][1] := 1;
```

6.3.3.1 Définition de types imbriqués

La notation TTCN-3 prend en charge la définition du type agrégé imbriqué dans la définition des types **record of** ou **set of**. La définition de nouveaux types structurés (**record, set, enumerated, set of** et **record of**) et la spécification de contraintes de sous-type sont toutes deux possibles.

EXEMPLE:

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

6.3.4 Types et valeurs énumérés

La notation TTCN-3 prend en charge les types énumérés (**enumerated**) qui servent à modéliser les types qui ne prennent qu'un ensemble nommé de valeurs distinctes. De telles valeurs distinctes sont appelées "énumérations". Chaque énumération doit avoir un identificateur. Les opérations sur types énumérés ne doivent utiliser que ces identificateurs et sont limitées aux opérateurs d'affectation, d'équivalence et de séquencement. Les identificateurs d'énumération doivent être uniques dans le type énuméré (mais sans avoir à être globalement uniques) et ne sont par conséquent visibles que dans le contexte du type indiqué. Les identificateurs d'énumération ne doivent être réutilisés que dans d'autres définitions de type structuré et ne doivent pas servir d'identificateurs de visibilité locale ou globale à un niveau égal ou inférieur de la même branche de la hiérarchie de portée (voir la hiérarchie de portée dans le § 5.3.0).

EXEMPLE 1:

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

type integer Monday;
// Cette définition est illégale, car le nom du type possède une visibilité
// locale ou globale

type enumerated MySecondEnumType {
  Saturday, Sunday, Monday
};
// Cette définition est légale car elle réutilise l'identificateur
//d'énumération "Monday" dans un type énuméré différent

type record MyRecordType {
  integer      Monday
};
// Cette définition est légale car elle réutilise l'identificateur
// d'énumération "Monday" dans un type structuré distinct, comme
// identificateur d'un certain champ de ce type
```

```
type record MyNewRecordType {
  MyFirstEnumType firstField,
  integer      secondField
};

var MyNewRecordType newRecordValue := { Monday, 0 }
// MyFirstEnumType est implicitement désigné via l'élément firstField
// de MyNewRecordType

const integer Monday := 7
// Cette définition est illégale car elle réutilise l'identificateur
// d'énumération "Monday" pour un objet TTCN-3 différent à l'intérieur de
// la même unité de portée
```

Chaque énumération peut facultativement avoir une valeur d'entier affectée, qui est définie après le nom de l'énumération entre parenthèses. Chaque nombre entier affecté doit être distinct à l'intérieur d'un même type **enumerated**. Pour chaque énumération sans valeur d'entier affectée, le système associe successivement un nombre entier dans l'ordre textuel des énumérations, à partir de la gauche, en commençant par zéro dans l'étape 1 et en sautant tout nombre occupé dans l'une quelconque des énumérations par une valeur affectée manuellement. Ces valeurs ne sont utilisées par le système qu'afin de permettre l'utilisation d'opérateurs relationnels.

NOTE 1 – La valeur d'entier peut également être utilisée par le système afin de coder/décoder des valeurs énumérées mais cela est hors du domaine d'application de la présente Recommandation (sauf que la notation TTCN-3 permet l'association d'attributs de codage à des items TTCN-3).

Pour toute instanciation ou référence de valeur de type **enumerated**, le type indiqué doit être désigné implicitement ou explicitement.

NOTE 2 – Si le type énuméré est un élément d'un type structuré défini par l'utilisateur, ce type énuméré est implicitement désigné via l'élément donné (c'est-à-dire par l'identificateur de l'élément ou par la position de la valeur dans une notation à liste de valeurs) lors de l'attribution de valeur, lors de l'instanciation etc.

EXEMPLE 2:

```
// Des instanciations valides de MyFirstEnumType et de MySecondEnumType seraient
var MyFirstEnumType Today := Tuesday;
var MySecondEnumType Tomorrow := Monday;

// Mais l'instruction suivante est illégale car les deux types
// d'énumération ne sont pas compatibles
Today := Tomorrow
```

6.3.5 Réunions

6.3.5.0 Généralités

La notation TTCN-3 prend en charge le type réunion (**union**) qui est un ensemble de champs dont chacun est désigné par un identificateur. Un seul des champs spécifiés sera jamais présent en une valeur effective de réunion. Les types Réunion sont utiles afin de modéliser une structure séquentielle qui peut prendre un type parmi un nombre fini de types connus.

EXEMPLE:

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

// Une instanciation valide de MyUnionType serait
var MyUnionType age, oneYearOlder;
var integer ageInMonths;

age.number := 34;       // notation de valeur par référence au champ.
                        // Noter que cette notation fait que le champ
                        // indiqué est celui qui est choisi
oneYearOlder := {number := age.number+1};

ageInMonths := age.number * 12;
```

La notation à liste de valeurs ne doit pas servir au réglage de valeurs de type **union**.

6.3.5.1 Référence à des champs de type réunion

Les champs d'un type **union** doivent être désignés par la notation à points (voir § 6.3.1.1).

EXEMPLE:

```
MyVar5 := MyUnion1.myChoice1;
 // Si un type réunion est imbriqué dans un autre type, alors la référence
 // peut avoir l'allure suivante
MyVar6 := MyRecord1.myElement1.myChoice2;
 // Noter que le type réunion dont le champ est désigné par
 // l'identificateur myChoice2 est imbriqué dans un type enregistrement
```

6.3.5.2 Offre d'options et réunion

Des champs facultatifs ne sont pas autorisés dans le type réunion **`union`**, c'est-à-dire que le mot clé **`optional`** ne doit pas être utilisé dans les types **`union`**.

6.3.5.3 Définition de types de champ imbriqués

La notation TTCN-3 prend en charge la définition de types de champ de réunion imbriqués dans la définition d'un type réunion, selon un mécanisme analogue à celui qui est utilisé pour les types enregistrement décrits dans le § 6.3.1.3.

6.4 Le type anytype

Le type spécial **`anytype`** est défini comme un abrégé pour la réunion de tous les types de données connus et du type adresse qui sont contenus dans un module TTCN-3. La définition du terme *types connus* est donnée dans le § 3.1, c'est-à-dire que le type **`anytype`** doit comprendre tous les types de données connus, mais pas les types de **`port`**, de **`component`** et par **`default`**. Le type **`address`** doit être inclus s'il a été défini explicitement dans ce module.

Les noms de champ du type **`anytype`** doivent être identifiés de façon univoque par les noms de type correspondants.

NOTE 1 – En raison de cette règle, les types importés qui ont des noms incompatibles (avec un identificateur d'une définition figurant dans le module d'importation ou avec un identificateur importé à partir d'un troisième module) ne peuvent pas être obtenus via le type **`anytype`** du module d'importation.

EXEMPLE:

```
// Une utilisation valide du type anytype serait
var anytype MyVarOne, MyVarTwo;
var integer MyVarThree;

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

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

Le type **`anytype`** est défini localement pour chaque module et (comme les autres types prédéfinis) ne peut pas être directement importé par un autre module. Cependant, un type **`anytype`** défini par l'utilisateur peut être importé par un autre module. Il en résulte que tous les types de ce module sont importés.

NOTE 2 – Le type **`anytype`** défini par l'utilisateur "contient" tous les types importés dans le module où il est déclaré. L'importation d'un tel type défini par l'utilisateur dans un module peut provoquer des effets secondaires et il conviendrait donc de prêter une attention appropriée à de tels cas.

6.5 Matrices

En commun avec de nombreux langages de programmation, les matrices ne sont pas considérées comme étant des types en notation TTCN-3. Elles peuvent en revanche être spécifiées au niveau d'une déclaration de variable. Les matrices peuvent être déclarées être unidimensionnelles ou multidimensionnelles. Les dimensions d'une matrice doivent être spécifiées au moyen d'expressions de constantes, qui doivent correspondre à des valeurs positives de type **`integer`**.

EXEMPLE 1:

```
var integer MyArray1[3];    // Instancie une matrice d'entiers de 3
                            // éléments ayant les indices 0 à 2
var integer MyArray2[2][3];// Instancie une matrice bidimensionnelle
                            // d'entiers à 2 × 3 éléments
                            // ayant des indices allant de (0,0)
                            // à (1,2)
```

L'accès aux éléments d'une matrice est assuré au moyen de la notation indexée (`[]`), qui doit spécifier un indice valide dans l'étendue de la matrice. L'accès aux éléments individuels des matrices multidimensionnelles se fait par utilisation répétée de la notation indexée. L'accès à des éléments situés en dehors de l'étendue de la matrice provoquera une erreur au moment de la compilation ou une erreur de test élémentaire.

EXEMPLE 2:

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

MyArray1[4] := 12;    // ERROR: l'indice doit être compris entre 0 et 2
MyArray2[3][2] := 15; // ERROR: le premier indice doit être de 0 ou de 1
```

Les dimensions de matrice peuvent également être spécifiées au moyen d'étendues. Dans de tels cas, les valeurs supérieure et inférieure de l'étendue définissent les valeurs inférieure et supérieure des indices.

EXEMPLE 3

```
var integer MyArray3[1 .. 5]; // Instancie une matrice d'entiers de
                              // 5 éléments ayant les indices 1 à 5
MyArray3[1] := 10; // Indice le plus bas
MyArray3[5] := 50; // Indice le plus haut

var integer MyArray4[1 .. 5][2 .. 3 ];  // Instancie une matrice
                                        // bidimensionnelle d'entiers à
                                        // 5 × 2 éléments d'indices
                                        // de (1,2) à (5,3)
```

Les valeurs des éléments de matrice doivent être compatibles avec la déclaration de variable correspondante. Les valeurs peuvent être affectées individuellement par notation à liste de valeurs ou par notation indexée, ou peuvent être affectées collectivement (en partie ou totalement) par notation à liste de valeurs. Quand la notation à liste de valeurs est utilisée, la première valeur de la liste est affectée au premier élément de la matrice (l'élément ayant l'indice 0), la seconde valeur au second élément, etc. Les éléments à exclure de l'affectation doivent être explicitement sautés ou omis de la liste. Afin d'affecter des valeurs à des matrices multidimensionnelles, chaque dimension affectée doit correspondre à un ensemble de valeurs englobées entre des accolades. Lors de la spécification de valeurs pour des matrices multidimensionnelles, la dimension de gauche correspond à la structure située le plus à l'extérieur de la valeur et la dimension de droite à la structure située le plus à l'intérieur. L'utilisation de tranches de matrices multidimensionnelles, c'est-à-dire avec un nombre d'indices de la valeur de la matrice inférieur au nombre de dimensions de la définition de la matrice correspondante, est autorisée. Les indices des tranches doivent correspondre aux dimensions de la définition de la matrice de gauche à droite (le premier indice de la tranche correspondant à la première dimension de la définition). Les indices de tranche doivent être conformes aux dimensions de la définition de la matrice correspondante.

EXEMPLE 4:

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

// ou au moyen d'une liste de valeurs
MyArray1:= {10, 20, -, 30};

MyArray4:= {{1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10}};
  // La valeur de la matrice est complètement définie

var integer MyArray5[2][3][4] :=
{
  {
    {1, 2, 3, 4}, // affecte une valeur à la tranche [0][0] de MyArray5
    {5, 6, 7, 8}, // affecte une valeur à la tranche [0][1] de MyArray5
    {9, 10, 11, 12} // affecte une valeur à la tranche [0][2] de MyArray5
  }, // met fin à l'affectation de valeurs à la tranche [0] de MyArray5
  {
    {13, 14, 15, 16}, {17, 18, 19, 20}, {21, 22, 23, 24}
  } // affecte une valeur à la tranche [1] de MyArray5
};

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

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

var integer MyArrayInvalid[2][2];
MyArrayInvalid := { 1, 2, 3, 4 }
  // non valide étant donné que la dimension de la notation de valeur ne
  // correspond pas aux dimensions de la définition
```

```
MyArrayInvalid[2] := { 1, 2 }
  // non valide étant donné que l'indice de la tranche devrait être 0 ou 1
```

NOTE – Une autre façon d'utiliser des structures de données multidimensionnelles consiste à utiliser les types record, record of, set ou set of.

EXEMPLE 5:

```
// soit
type record MyRecordType
{
  integer        field1,
  MyOtherStruct  field2,
  charstring     field3
}
// Une matrice du type MyRecordType pourrait être:
var MyRecordType myRecordArray[10];
// Une référence à un élément particulier aura l'allure suivante
myRecordArray[1].field1 := 1;
```

6.6 Types récursifs

S'il y a lieu, les définitions de type TTCN-3 peuvent être récursives. L'utilisateur doit cependant veiller à ce que la récurrence de tous les types soit réalisable et à ce qu'aucune récurrence en boucle ne se produise.

6.7 Compatibilité des types

6.7.0 Généralités

Généralement, la notation TTCN-3 exige la compatibilité des types de valeurs lors des affectations, des instanciations et des comparaisons.

Dans le cadre du présent paragraphe, la valeur effective à affecter, à transmettre comme paramètre, etc., est appelée la valeur "b". Le type de la valeur "b" est appelé le type "B". Le type du paramètre formel, qui permettra d'obtenir la valeur "b" effective, est appelé le type "A".

6.7.1 Compatibilité des types non structurés

Dans les variables, les constantes, les modèles, etc., des simples types de base et des types bitstring, hexstring et octetstring, la valeur "b" est compatible avec le type "A" si le type "B" correspond au même type racine que le type "A" (par exemple au type **integer**) et ne viole pas le sous-typage (par exemple étendues, restrictions de longueur) du type "A".

EXEMPLE:

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

// Alors
y := 5; // est une affectation valide

x := y;
// est une affectation valide parce que "y" a le même type racine que "x"
// et qu'aucun sous-typage n'est violé

x := 20; // est une affectation valide
y := x;
// n'est PAS une affectation valide parce que la valeur de "x" est en
// dehors de l'étendue de MyInteger

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

// Si l'on a:
type charstring MyChar length (1);
type charstring MySingleChar length (1);
var MyChar myCharacter;
var charstring myCharString;
var MySingleChar mySingleCharString := "B";
```

```
// Alors
myCharString := mySingleCharString;
// est une affectation valide car la limitation de la longueur de la chaîne
// de caractères à 1 est compatible avec cette chaîne de caractères.
myCharacter := mySingleCharString;
// est une affectation valide car deux chaînes de caractères d'une longueur
// d'un caractère sont compatibles.

// Si l'on a:
myCharString := "abcd";

// Alors
myCharacter := myCharString[1];
// est valide car la notation r.h.s. s'applique à un seul élément de la chaîne.

// Si l'on a:
var charstring myCharacterArray [5] := {"A", "B", "C", "D", "E"}

// Alors
myCharString := myCharacterArray[1];
// est valide et affecte la valeur "B" à myCharString;
```

Dans les variables, les constantes, les modèles, etc., du type **`charstring`**, la valeur 'b' est compatible avec un type **`universal charstring`** 'A' sauf si elle viole une spécification de contrainte de type (étendue, liste ou longueur) du type "A".

Dans les variables, les constantes, les modèles, etc., du type **`universal charstring`**, la valeur 'b' est compatible avec un type **`charstring`** 'A' si à chacun des caractères utilisés dans la valeur 'b' correspond un caractère (c'est-à-dire le même caractère de commande ou graphique utilisant le même code de caractère) du type **`charstring`** et si cette valeur ne viole pas une spécification de contrainte de type (étendue, liste ou longueur) du type "A".

6.7.2 Compatibilité des types structurés

6.7.2.0 Généralités

Dans le cas des types structurés (sauf le type **`enumerated`**) une valeur "b" du type "B" est compatible avec le type "A" si les structures de valeur effective de type "B" et de type "A" sont compatibles, auquel cas les affectations, les instanciations et les comparaisons sont autorisées.

6.7.2.1 Compatibilité des types énumérés

Les types énumérés ne sont jamais compatibles avec d'autres types de base ou structurés (c'est-à-dire qu'un typage fort est requis pour les types énumérés).

6.7.2.2 Compatibilité des types "record" et "record of"

Dans les types **`record`**, les structures de valeur effective sont compatibles si le nombre et l'aspect facultatif des champs sont identiques dans l'ordre textuel de la définition, si les types de chaque champ sont compatibles et si la valeur de chaque champ de la valeur "b" est compatible avec le type du champ correspondant dans le type "A". La valeur de chaque champ contenu dans la valeur "b" est affectée au champ correspondant qui est contenu dans la valeur de type "A".

EXEMPLE 1:

```
// Si l'on a:
type record AType {
  integer        a(0..10)    optional,
  integer        b(0..10)    optional,
  boolean        c
}

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

type record CType {         // type avec différents noms de champ
  integer             d   optional,
  integer             e   optional,
  boolean             f

}
```

```
type record DType {         // type avec champ c facultatif
  integer            a   optional,
  integer            b   optional,
  boolean            c   optional
}

type record EType {         // type avec champ supplémentaire d
  integer            a   optional,
  integer            b   optional,
  boolean            c,
     float           d   optional
}

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

  // Alors

  MyVarA := MyVarB;     // est une affectation valide,
                       // la valeur de MyVarA est(a:= <undefined>, b:= 2, c:= true)
  MyVarC := MyVarB;     // est une affectation valide
                       // la valeur de MyVarC est( d := <undefined>, e:= 2, f:= true)
  MyVarA := MyVarD;     // n'est PAS une affectation valide car les
                       // options offertes par les champs ne correspondent pas
  MyVarA := MyVarE;     // n'est PAS une affectation valide car le nombre
                       // de champs ne correspond pas

  MyVarC := { d:= 20 };// la valeur effective de MyVarC est { d:=20, e:=2,f:= true }
  MyVarA := MyVarC     // n'est PAS une affectation valide car le champ 'd'
                        // de MyVarC viole le sous-typage du champ 'a' de AType
```

Dans les types **record of** et dans les matrices, les structures de valeur effective sont compatibles si leurs types de composant sont compatibles et si la valeur "b" du type "B" ne viole pas un quelconque sous-typage de longueur du type **record of** ou une dimension de la matrice de type "A". Les valeurs des éléments de la valeur "b" doivent être affectées séquentiellement à l'instance de type "A", y compris les éléments indéfinis.

Les types **record of** et les matrices unidimensionnelles sont compatibles avec les types **record** si leurs structures de valeur effective sont compatibles et si le nombre d'éléments de valeur "b" du type **record of** "B" ou la dimension de la matrice "b" est exactement identique au nombre d'éléments du type **record** "A". L'offre d'options des champs de type **record** n'a aucune importance lors de la détermination de la compatibilité, c'est-à-dire qu'elle n'a aucune incidence sur le comptage des champs (en d'autres termes, les champs facultatifs doivent toujours être inclus dans le décompte). L'affectation des valeurs d'élément du type **record of** ou de la matrice à l'instance d'un type **record** doit être effectuée dans l'ordre textuel de la définition correspondante de ce type **record**, y compris les éléments indéfinis. Si un élément ayant une valeur indéfinie est affecté à un élément facultatif du type **record**, cette affectation va provoquer l'omission de l'élément facultatif. Une tentative d'affecter un élément de valeur indéfinie à un élément obligatoire du type **record** doit provoquer une erreur.

NOTE – Si le type **record of** n'a aucune restriction de longueur ou si la restriction de longueur dépasse le nombre d'éléments du type **record** comparé et si l'indice d'un quelconque élément défini de la valeur de type **record of** est inférieur ou égal au nombre d'éléments du type **record** moins un, alors l'exigence de compatibilité est toujours satisfaite.

Les valeurs d'un type **record** peuvent également être affectées à une instance d'un type **record of** ou à une matrice unidimensionnelle si aucune restriction de longueur du type **record of** n'est violée ou si la dimension de la matrice est supérieure ou égale au nombre d'éléments du type **record**. Les éléments facultatifs faisant défaut dans la valeur du type **record** doivent être affectés en tant qu'éléments de valeur indéfinie.

EXEMPLE 2:

```
// Si l'on a:
type record HType {
  integer a,
  integer b optional,
  integer c
}

type record of integer IType

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

```
// Alors

MyArrayVar := MyVarH;
// est une affectation valide car le type de MyArrayVar et le type HType
// sont compatibles

MyVarI := MyVarH;
// est une affectation valide car les types sont compatibles et aucun
// sous-typage n'est violé

MyVarI := { 3, 4 };
MyVarH := MyVarI;
// n'est PAS une affectation valide car le champ obligatoire 'c' de Htype
// ne reçoit aucune valeur
```

6.7.2.3 Compatibilité des types "set" et "set of"

Les types **set** ne sont compatibles qu'avec d'autres types **set** et **set of**. Les mêmes règles de compatibilité que pour les types **record** et **record of** doivent s'appliquer aux types **set** et **set of**.

NOTE 1 – Cela implique que, bien que l'ordre des éléments à envoyer et à recevoir soit inconnu, l'ordre textuel des champs contenus dans la définition de type est décisif lors de la détermination de la compatibilité des types **set**.

NOTE 2 – Dans les valeurs de type **set**, l'ordre des champs peut-être arbitraire. Toutefois, cela n'a pas d'incidence sur la compatibilité des types car les noms de champ identifient sans ambiguïté les champs du type **set** considéré qui correspondent aux champs de valeur de type **set.**

EXEMPLE:

```
// Si l'on a:
type set FType {
  integer aoptional,
  integer boptional,
  boolean c
}

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

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

// Alors

MyVarF := MyVarG;  // est une affectation valide car les types FType et
                   // GType sont compatibles

MyVarF := MyVarA;  // n'est PAS une affectation valide car MyVarA est un
                   // type enregistrement
```

6.7.2.4 Compatibilité entre sous-structures

Les règles définies dans le présent paragraphe pour la compatibilité des types structurés sont également applicables à la sous-structure de tels types.

EXEMPLE:

```
// Si l'on a:
type record JType {
      HType H,
      integer  b optional,
      integer  c
}

var JType MyVarJ

// Compte tenu des déclarations ci-dessus


MyVarJ.H := MyVarH;
// est une affectation valide car le type du champ H du type J et le type H
// sont compatibles

MyVarI := MyVarJ.H;
// est une affectation valide car le Type I et le type du champ H du Type J
// sont compatibles
```

6.7.3 Compatibilité des types de composant

La compatibilité des types de composant doit être considérée sous deux aspects différents.

1) La compatibilité d'une valeur de référence de composant avec un type de composant (par exemple lors de la transmission d'une référence de composant comme paramètre effectif à une fonction ou à une variante ou lors de l'affectation d'une valeur de référence de composant à une variable d'un type de composant différent): une référence de composant "b" du type de composant "B" est compatible avec le type de composant "A" si toutes les définitions de "A" ont des définitions identiques dans "B".

2) Compatibilité relative à la clause **`runs on`**: une fonction ou une variante faisant référence au type de composant "A" dans sa clause **`runs on`** peut être appelée ou lancée sur une instance de composant de type "B" si toutes les définitions de "A" ont des définitions identiques dans "B".

L'identité des définitions de "A" avec les définitions de "B" est déterminée selon les règles suivantes:

- pour les instances de port, aussi bien le type que l'identificateur doivent être identiques;
- pour les instances de temporisation, les deux identificateurs doivent être identiques, avec l'un et l'autre une durée initiale identique ou aucune durée initiale;
- pour les instances de variable et les définitions de constante, les identificateurs, les types et les valeurs d'initialisation doivent être identiques (c'est-à-dire, dans le cas de variables, les mêmes dans les deux définitions ou absentes);
- pour les définitions de modèle local, les identificateurs, les types, les listes de paramètres formels et les valeurs affectées au modèle ou au champ de modèle doivent être identiques.

6.7.4 Compatibilité de types des opérations de communication

Les opérations de communication (voir § 23) **`send`**, **`receive`**, **`trigger`**, **`call`**, **`getcall`**, **`reply`**, **`getreply`** et **`raise`** font exception à la règle faible de compatibilité des types et nécessitent un typage fort. Les types de valeurs ou les modèles directement utilisés comme paramètres dans ces opérations doivent par ailleurs être définis explicitement dans la définition du type de port associé. Le typage fort s'applique également à la mémorisation de la valeur, de l'adresse ou de la référence de composant qui a été reçue pendant une opération **`receive`** ou **`trigger`**.

6.7.5 Conversion de type

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

EXEMPLE:

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

7 Modules

7.0 Généralités

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

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

7.1 Nommage des modules

Les noms des modules sont exprimés par un identificateur TTCN-3. De plus, une spécification de module peut comporter un attribut facultatif identifié par le mot clé **`language`** indiquant la version du langage TTCN-3 dans laquelle le module est spécifié. Actuellement, les chaînes de langage suivantes sont prises en charge: "TTCN-3:2001" pour une spécification de module conforme à la version TTCN-3 2001, "TTCN-3:2003" pour la version 2003 et "TTCN-3:2005" pour la version 2006.

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

EXEMPLE:

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

7.2 Paramètres de module

7.2.0 Généralités

La liste de paramètres de module définit un ensemble de valeurs qui sont fournies par l'environnement de test au moment de l'exécution. Pendant l'exécution du test, ces valeurs doivent être traitées comme des constantes. Les paramètres de module sont déclarés par indication du type et énumération de leurs identificateurs après le mot clé **modulepar**. Ils ne doivent pas être du type port, du type par défaut ou du type composant. Un paramètre de module doit uniquement être du type adresse, si le type adresse est défini explicitement dans le module associé. Les paramètres de module ne doivent être déclarés que dans la partie définitions d'un module. Plusieurs occurrences de déclaration des paramètres d'un module sont autorisées mais chaque paramètre ne doit être déclaré qu'une seule fois (c'est-à-dire que la redéfinition d'un paramètre de module n'est pas autorisée).

EXEMPLE:

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

7.2.1 Valeurs par défaut des paramètres de module

Il est permis de spécifier des valeurs par défaut pour les paramètres de module. Cela doit être effectué par une affectation dans la liste des paramètres du module. Une valeur par défaut ne peut être qu'une valeur littérale et ne peut être affectée que dans le cadre de la déclaration du paramètre. Si le système de test n'offre pas une valeur d'exécution effective pour le paramètre considéré, la valeur par défaut doit être utilisée pendant l'exécution du test; autrement, la valeur effective fournie par le système de test doit être utilisée.

EXEMPLE:

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

7.3 Partie d'un module relative aux définitions

7.3.0 Généralités

La partie d'un module relative aux définitions spécifie les définitions de niveau sommital du module et peut importer des identificateurs à partir d'autres modules. Les règles de portée applicables aux déclarations effectuées dans la partie définitions d'un module et les déclarations importées sont indiquées dans le § 5.3. Les éléments de langage qui peuvent être définis dans un module TTCN-3 sont énumérés dans le Tableau 1. La partie définitions du module peut être importée par d'autres modules.

EXEMPLE:

```
module MyModule
{ // Ce module ne contient que des définitions
  :
  const integer MyConstant := 1;
  type record MyMessageType { … }
  :
  function TestStep(){ … }
  :
}
```

Les déclarations d'éléments de langage dynamiques comme **var** ou **timer** ne doivent être effectuées que dans la partie commande, dans les tests élémentaires, dans les fonctions, dans les variantes ou dans les types de composant.

NOTE – La notation TTCN-3 ne prend pas en charge la déclaration de variables dans la partie d'un module relative aux définitions. Autrement dit, des variables globales ne peuvent pas être définies en notation TTCN-3. Cependant, des variables définies dans un composant de test peuvent être utilisées par tous les tests élémentaires, les fonctions, etc. fonctionnant sur ce

composant; les variables définies dans la partie commande offrent la capacité de conserver leur valeur indépendamment de l'exécution des tests élémentaires.

7.3.1 Groupes de définitions

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

Un groupement est effectué afin d'améliorer la lisibilité et d'ajouter une structure logique au module si nécessaire. Les groupes (imbriqués ou non) ne sont associés à aucune portée *sauf* dans le contexte d'identificateurs de groupe et d'attributs affectés à un groupe par une instruction associée de type **`with`**. Autrement dit:

- les identificateurs de groupe contenus dans le module entier n'ont pas forcément besoin d'être uniques. Cependant, tous les identificateurs de sous-groupes d'un même groupe doivent être uniques. Si nécessaire, la notation à points doit servir à identifier sans équivoque des sous-groupes dans la hiérarchie du groupe, par exemple pour l'importation d'un sous-groupe spécifique;
- les règles d'écrasement applicables aux attributs sont indiquées dans le § 28.4.

EXEMPLE:

```
module MyModule {
   :
   // Un recueil de définitions:
   group MyGroup {
       const integer MyConst:= 1;
        :
       type record MyMessageType { … };
       group MyGroup1 { // Sous-groupe avec définitions
            type record AnotherMessageType { … };
            const boolean MyBoolean := false
       }
   }

   // Un groupe de variantes:
   group MyStepLibrary {
       group MyGroup1 {  // Sous-groupe avec le même nom que le sous-groupe
                         // avec définitions
            altstep MyStep11() { … }
            altstep MyStep12() { … }
              :
            altstep MyStep1n() { … }
       }
       group MyGroup2 {
            altstep MyStep21() { … }
            altstep MyStep22() { … }
            :
            altstep MyStep2n() { … }
       }
   }
   :
}

// Une instruction d'importation qui importe MyGroup1 dans MyStepLibrary
import from MyModule {
   group MyStepLibrary.MyGroup1
}
```

7.4 Partie d'un module relative à la commande

La partie d'un module relative à la commande peut contenir des définitions locales. Elle décrit l'ordre d'exécution (éventuellement répétitif) des tests élémentaires effectifs. Un test élémentaire doit être défini dans la partie d'un module relative aux définitions et doit être appelé dans la partie commande.

EXEMPLE:

```
module MyTestSuite
{     // Ce module contient des définitions …
      :
      const integer MyConstant := 1;
      type record MyMessageType { … }
      template MyMessageType MyMessage := { … }
      :
      function MyFunction1() { … }
```

```
        function MyFunction2() { … }
        :
        testcase MyTestcase1() runs on MyMTCType { … }
        testcase MyTestcase2() runs on MyMTCType { … }
        :
        // … et contient une partie commande de sorte qu'il est exécutable
        control
        {
            var boolean MyVariable;   // Variable de commande locale
            :
            execute( MyTestCase1());  // exécution séquentielle de tests élémentaires
            execute( MyTestCase2());
            :
        }
    }
```

7.5 Importation à partir de modules

7.5.0 Généralités

Il est possible de réutiliser des définitions spécifiées dans différents modules au moyen de l'instruction **`import`**. La notation TTCN-3 ne contient aucune construction d'exportation explicite; donc, par défaut, toutes les définitions contenues dans la partie d'un module relative aux définitions peuvent être importées. Une instruction **`import`** peut être utilisée n'importe où dans la partie d'un module relative aux définitions. Elle ne doit pas être utilisée dans la partie commande.

Si une définition importée possède des attributs (définis au moyen d'une instruction **`with`**), alors ces attributs doivent également être importés. Le mécanisme de modification des attributs de définitions importées est expliqué dans le § 28.6.

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

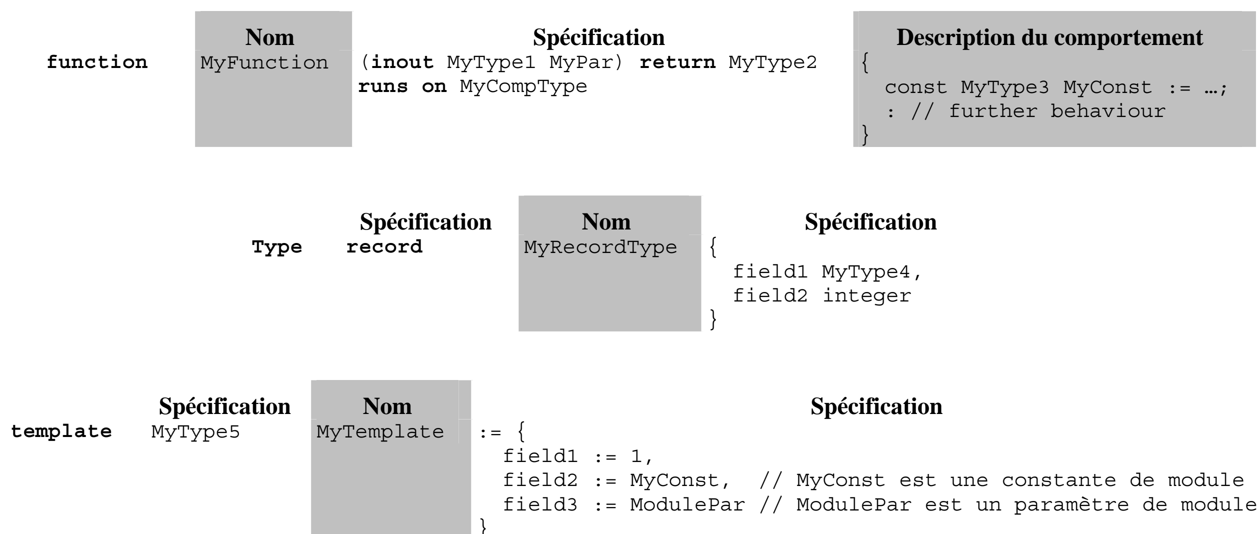
EXEMPLE:

```
module MyModuleA
{       // Ce module contient des définitions et des définitions importées
        :
        const integer MyConstant := 1;
        import from MyModuleB all;    // La portée des définitions importées est
                                      // globale pour MyModuleA
        import from MyModuleC {
            type MyType1, MyType2;
            template all
        }
        type record MyMessageType { … }
        :
        function MyBehaviourC()
        {
            const integer MyConstant := 2;
            // L'importation ne peut pas être utilisée ici
              :
        }
        :
        control
        {   // L'importation ne peut pas être utilisée ici
            :
        }
}
```

7.5.1 Structure des définitions importables

La notation TTCN-3 prend en charge l'importation des définitions suivantes: paramètres de module, types définis par l'utilisateur, signatures, constantes, constantes externes, modèles de données, modèles de signature, fonctions, fonctions externes, variantes et tests élémentaires. Chaque définition possède un *nom* (qui indique l'identificateur de la définition, par exemple un nom de fonction), une *spécification* (par exemple une spécification de type ou une signature de fonction) et, dans le cas de fonctions, de variantes et de tests élémentaires, une *description de comportement* associée.

EXEMPLE:



Les descriptions de comportement n'ont aucun effet sur le mécanisme d'importation parce que leurs composants internes sont considérés comme invisibles par l'importateur quand les fonctions, variantes ou tests élémentaires correspondants sont importés. Ceux-ci ne sont donc pas pris en considération dans les descriptions ci-après.

La partie spécification d'une définition importable contient des *définitions locales* (par exemple des noms de champ de définitions de types structurés ou des valeurs de types énumérés) et des *définitions référencées* (par exemple par des références à des définitions de type, à des modèles, à des constantes ou à des paramètres de module). Pour les exemples ci-dessus, cela implique ce qui suit:

	Nom	Définitions locales	Définitions référencées
function	MyFunction	MyPar	MyType1, MyType2, MyCompType
type	MyRecordType	field1, field2	MyType4, integer
template	MyTemplate		MyType5, field1, field2, field3, MyConst, ModulePar

NOTE 1 – La colonne des définitions locales ne se rapporte qu'aux identificateurs récemment définis dans la définition importable. Les valeurs affectées à des champs individuels de définitions importables, par exemple dans des définitions de modèle, peuvent également être considérées comme des définitions locales mais ne sont pas importantes pour l'explication du mécanisme d'importation.

NOTE 2 – Les définitions référencées "field1", "field2" et "field3" du modèle "MyTemplate" sont les noms des champ de "MyType5", c'est-à-dire qu'elles sont référencées via "MyType5".

Les définitions référencées sont également des définitions importables, c'est-à-dire que la source d'une définition référencée peut de nouveau être structurée en nom ou en partie spécification, celle-ci contenant donc des définitions locales et des définitions référencées. En d'autres termes, une définition importable peut être construite de façon récurrente à partir d'autres définitions importables.

Le mécanisme d'importation de la notation TTCN-3 est associé aux définitions locales et référencées qui sont utilisées dans la partie spécification des définitions importables. Le Tableau 5 spécifie donc les définitions locales et référencées pouvant provenir de définitions importables.

Tableau 5/Z.140 – Définitions locales et référencées pouvant provenir de définitions importables

Définition importable	Définitions locales possibles	Définitions référencées possibles
Paramètre de module		Type de paramètre de module
Type défini par l'utilisateur (pour tous les sous-types)	Noms de paramètre	Type de paramètre
• type énuméré	Valeurs concrètes	
• type structuré	Noms de champ	Types de champ
• type de port		Types de message, signatures

Tableau 5/Z.140 – Définitions locales et référencées pouvant provenir de définitions importables

Définition importable	Définitions locales possibles	Définitions référencées possibles
• type de composant	Noms de constante, noms de variable, noms de temporisateur et noms de port	Types de constante, types de variable, types de port
Signature	Noms de paramètre	Types de paramètre, type de retour, types d'exception
Constante		Type de constante
Constante externe		Type de constante
Modèle de données	Noms de paramètre	Type de modèle, types de paramètre, constantes, paramètres de module, fonctions
Modèle de signature		Définition de signature, constantes, paramètres de module, fonctions
Fonction	Noms de paramètre	Types de paramètre, type de retour, type de composant (clause **`runs on`**)
Fonction externe	Noms de paramètre	Types de paramètre, type de retour
Variante	Noms de paramètre	Types de paramètre, type de composant (clause **`runs on`**)
Test élémentaire	Noms de paramètre	Types de paramètre, types de composant (clause **`runs on`** et **`system`**)

Le mécanisme d'importation TTCN-3 établit une distinction entre *identificateur d'une définition référencée* et *informations nécessaires pour l'usage d'une définition référencée* dans le cadre de la définition importée. L'identificateur d'une définition référencée n'est pas requis pour l'usage de celle-ci et n'est donc pas importé automatiquement.

7.5.2 Règles d'utilisation des importations

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

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

b) seule l'importation directe à partir du module source d'une définition (c'est-à-dire le module où se trouve la définition effective pour l'identificateur désigné dans l'instruction d'importation **`import`**) est autorisée;

c) une définition est importée en même temps que son nom et toutes les définitions locales;

NOTE 1 – Une définition locale, par exemple un nom de champ d'un type d'enregistrement défini par l'utilisateur, n'a de signification que dans le contexte des définitions dans lesquelles elle est définie; par exemple un nom de champ d'un type d'enregistrement ne peut servir qu'à accéder à un champ du type d'enregistrement et non à l'extérieur de ce contexte;

d) une définition est importée en même temps que toutes les informations de définitions référencées qui sont nécessaires pour l'usage de la définition référencée;

NOTE 2 – Les instructions d'importation sont transitives; par exemple, si un module A importe une définition à partir d'un module B qui utilise une référence de type définie dans module C, les informations correspondantes qui sont nécessaires pour l'usage de ce type sont automatiquement importées dans le module A;

e) les identificateurs de définitions référencées ne sont pas automatiquement importés;

NOTE 3 – Si l'on souhaite utiliser les définitions référencées dans le module, ces définitions doivent être explicitement importées à partir de leur module source;

f) lors de l'importation d'une fonction, d'une variante ou d'un test élémentaire, les spécifications de comportement correspondantes et toutes les définitions utilisées à l'intérieur de ces spécifications de comportement restent invisibles pour le module d'importation;

g) les importations cycliques sont interdites.

EXEMPLE:

```
module ModuleONE {

  modulepar integer ModPar1, ModPar2 := 7

  type record RecordType_T1 {
     integer  Field1_T1,
     boolean  Field2_T1
```

```
   }

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

   const integer MyConst := 13;

   template RecordType_T2 Template_T2 (RecordType_T1 TempPar_T2):= { // modèle paramétré
      Field1_T2 := TempPar_T2,      // Référence à un paramètre de modèle
      Field2_T2 := {MyConst, true}, // Référence à une constante de module
      Field3_T2 := ModPar1          // Référence à un paramètre de module
   }

} // fin du module ModuleONE


module ModuleTWO {

   import from ModuleONE {
      template Template_T2
   }

   // Seuls les noms Template_T2 et TempPar_T2 seront visibles dans
   // ModuleTWO. Noter que l'identificateur TempPar_T2 ne peut être utilisé
   // que dans le contexte de Template_T2, par exemple lors de la fourniture
   // d'une valeur de paramètre effective. Toutes les informations
   // nécessaires pour l'usage de Template_T2, par exemple aux fins de la
   // vérification du type, sont importées pour les définitions
   // référencées RecordType_T2, RecordType_T1, Field1_T2,
   // Field2_T2,Field3_T3, MyConst et ModPar1, mais leurs identificateurs
   // sont invisibles dans ModuleTWO. Autrement dit, par exemple il n'est
   // pas possible d'utiliser la constante MyConst ou de déclarer une
   // variable de type RecordType_T1 ou RecordType_T2 dans ModuleTWO sans
   // importer explicitement ces types

   import from ModuleONE {
      modulepar ModPar2
   }

   // Le paramètre de module ModPar2 de ModuleONE est importé à partir de
   // ModuleONE et peut être utilisé comme une constante entière

} // fin du module ModuleTWO

module ModuleTHREE {

   import from ModuleONE all; // importe toutes les définitions à partir de ModuleONE

   type port MyPortType {
      inout RecordType_T2
   }

   type component MyCompType {
      var integer MyComponentVar := ModPar2;   // Référence à un paramètre
                                               // de module de ModuleONE
      port MyPortType MyPort
   }

   function MyFunction () return integer {
      return MyConst    // Retourne une constante de module définie dans ModuleONE
   }

   testcase MyTestCase (out RecordType_T2 MyPar) runs on MyCompType {

      var integer MyTCVar := ModPar2;   // Référence à un paramètre
                                        // de module de ModuleONE

      MyPort.send(Template_T2); // Envoi d'un modèle défini dans ModuleONE
      MyPort.receive(RecordType_T2 : ?) -> value MyPar; // La valeur reçue est affectée au
                                                        // paramètre out MyPar.

   } // fin du test élémentaire MyTestCase

} // fin du moduleTHREE
```

```
module ModuleFOUR {

  import from ModuleTHREE {
     testcase MyTestCase
  }

  // Seuls les noms MyTestCase et MyPar seront visibles et utilisables dans
  // ModuleFOUR. Les informations de type pour RecordType_T2 sont importées
  // via ModuleTHREE à partir de ModuleONE et les informations de type pour
  // MyCompType sont importées à partir de ModuleTHREE. Toutes les
  // définitions utilisées dans la partie comportement de MyTestCase
  // restent cachées pour l'utilisateur de ModuleFOUR.

} // fin du moduleFOUR
```

7.5.3 Vide

7.5.4 Importation de définitions isolées

Des définitions isolées peuvent être importées.

EXEMPLE:

```
import from MyModuleA {
  type MyType1          // importe une seule définition de type à partir de MyModuleA
}

import from MyModuleB {
  type MyType2, Mytype3, MyType4;  // importe trois types
  template MyTemplate1;            // importe un seul modèle
  const MyConst1, MyConst2         // importe deux constantes
}
```

7.5.5 Importation de toutes les définitions d'un module

Toutes les définitions d'une partie de module relative aux définitions peuvent être importées au moyen du mot clé **all** immédiatement après le nom de module. Si toutes les définitions d'un module sont importées au moyen du mot clé **all**, aucune autre forme d'importation (importation de définitions isolées, importation de la même sorte, etc.) ne doit servir à la même instruction **import**.

EXEMPLE 1:

```
import from MyModule all;
```

Si certaines déclarations sont choisies comme ne devant pas être importées, leur sorte et leur identificateur doivent être énumérés dans la liste d'exceptions entre deux accolades après le mot clé **except**.

EXEMPLE 2:

```
import from MyModule all except {
  type MyType3, MyType5
     // les déclarations de type MyType3 et MyType5 sont exclues de
     // l'instruction d'importation mais toutes les autres
     // déclarations de MyModule sont importées
}
```

Le mot clé **all** peut également être utilisé dans la liste d'exceptions; cela exclura de l'instruction d'importation toutes les déclarations de la même sorte.

EXEMPLE 3:

```
import from MyModule all except {
  type MyType3, MyType5;   // les deux types sont exclus de
                           // l'instruction d'importation
  template all             // tous les modèles déclarés dans MyModule
                           // sont exclus de l'instruction d'importation
}
```

7.5.6 Importation de groupes

Des groupes de définitions peuvent être importés.

EXEMPLE 1:

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

L'effet de l'importation d'un groupe est identique à une instruction **import** qui énumère toutes les définitions importables (y compris les sous-groupes) de ce groupe.

En notation TTCN-3, les groupes ne sont utilisés qu'aux fins de la structuration et ne sont pas des unités de portée. Il est donc permis d'importer des sous-groupes (c'est-à-dire un groupe qui est défini dans un autre groupe) directement, c'est-à-dire sans les groupes dans lesquels le sous-groupe est imbriqué. Si le nom d'un sous-groupe qui devrait être importé est identique au nom d'un autre sous-groupe dans le même module (voir § 7.3.1), la notation à points doit servir à identifier de façon univoque le sous-groupe à importer.

Si certaines définitions d'un groupe sont choisies comme ne devant pas être importées, leur sorte et leur identificateur doivent être énumérés dans la liste d'exceptions entre deux accolades après le mot clé **except**.

EXEMPLE 2:

```
import from MyModule {
  group MyGroup except {
      type MyType3, MyType5
           // la définition des types MyType3 et MyType5 est exclue
           // de l'instruction d'importation mais toutes
           // les autres définitions de MyGroup sont importées
  }
}
```

Le mot clé **all** peut également être utilisé dans la liste d'exceptions; ce qui exclura de l'instruction d'importation toutes les définitions de la même sorte.

EXEMPLE 3:

```
import from MyModule {
  group MyGroup except {
      type MyType3, MyType5;    // les deux types sont exclus de
                                // l'instruction d'importation et
                                // tous les modèles définis dans
      template all              // MyGroup sont exclus
                                // de l'instruction d'importation
  }
}
```

7.5.7 Importation de définitions de la même sorte

Le mot clé **all** peut servir à importer toutes les définitions de la même sorte d'un module. Utilisé avec le mot clé **constant**, le mot clé **all** identifie toutes les constantes ainsi que toutes les constantes externes déclarées dans la partie définitions du module auquel se rapporte l'instruction d'importation. De même, utilisé avec le mot clé **function**, le mot clé **all** identifie toutes les fonctions et toutes les fonctions externes définies dans le module indiqué dans l'instruction d'importation.

EXEMPLE 1:

```
import from MyModule {
  type all;           // importe tous les types de MyModule
  template all        // importe tous les modèles de MyModule
}
```

Si on souhaite que certaines déclarations d'une sorte soient exclues de l'instruction d'importation, leur identificateur doit être énuméré après le mot clé **except**.

EXEMPLE 2:

```
import from MyModule {
  type all except MyType3, MyType5;     // importe tous les types sauf MyType3 et MyType5
  template all                          // importe tous modèles définis dans Mymodule
}
```

7.5.8 Traitement des conflits de noms lors d'une importation

Tous les modules TTCN-3 doivent avoir leur propre espace de noms dans lequel toutes les définitions doivent être identifiées de façon univoque. Des conflits de noms peuvent apparaître en raison d'importations, par exemple une importation à partir de différents modules. Les conflits de noms doivent être résolus par adjonction d'un préfixe à la définition importée (qui provoque le conflit de noms) avant l'identificateur du module à partir duquel elle est importée. Le préfixe et l'identificateur doivent être séparés par un point (.).

S'il n'y a aucune ambiguïté, le préfixe peut (au besoin) être présent quand la définition importée est utilisée. Quand la définition est référencée dans le module où elle est justement définie, l'identificateur du module (actuel) peut également servir de préfixe ajouté à l'identificateur de la définition.

EXEMPLE:

```
module MyModuleA  {
    :
   type bitstring MyTypeA;
   import from SomeModuleC {
      type MyTypeA,     // Où MyTypeA est de type chaîne de caractères
           MyTypeB      // Où MyTypeB est de type chaîne de caractères
   }
    :
   control {
      :
      var  SomeModuleC.MyTypeA MyVar1 := "Test String";  // un préfixe doit
                                                        // impérativement être utilisé
      var MyTypeA MyVar2 := '10110011'B;  // c'est le type original MyTypeA
      :
      var MyTypeB MyVar3 := "Test String"; // un préfixe n'a pas besoin d'être utilisé …
      var SomeModuleC.MyTypeB MyVar3 := "Test String";   // … mais peut l'être si tel
                                                        // est le souhait
      :
   }
}
```

NOTE – Les définitions où le même nom est défini dans différents modules sont toujours censées être différentes, même si les définitions effectives sont identiques dans les différents modules. Par exemple, l'importation d'un type qui est déjà défini localement, même avec le même nom, conduira à la disponibilité de deux types différents dans le même module.

7.5.9 Traitement de multiples références à la même définition

L'utilisation de l'instruction **`import`** avec des définitions isolées, des groupes de définitions, des définitions de la même sorte, etc., peut conduire à des situations où la même définition est désignée plus d'une seule fois. De tels cas doivent être résolus par le système et les définitions ne doivent être importées qu'une seule fois.

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

Toutes les instructions **`import`** et toutes les définitions contenues dans des instructions d'importation sont considérées comme étant traitées indépendamment l'une après l'autre, dans l'ordre de leur apparition. Il est important de souligner qu'en général, l'instruction **`except`** n'empêche pas d'une manière générale l'importation des définitions énumérées; toutes les instructions important des définitions de la même sorte peuvent être considérées comme une notation abrégée pour une liste équivalente d'identificateurs de définitions isolées. Avec l'instruction **`except`**, les définitions sont uniquement exclues de cette liste.

EXEMPLE:

```
import from MyModule {
     type all except MyType3;      // importe tous les types de MyModule sauf MyType3
     type MyType3                  // importe MyType3 explicitement
}
```

7.5.10 Importation de définitions à partir de modules non TTCN-3

Si des définitions sont importées à partir d'autres sources que des modules TTCN-3, la spécification du langage doit servir à indiquer le langage (qui peut être associé à un numéro de version) de la source (par exemple module, paquetage, bibliothèque ou même fichier) à partir de laquelle ces définitions sont importées. Cette spécification se compose du mot clé **`language`** suivi d'une déclaration textuelle du langage indiqué. L'utilisation de la spécification de langage est facultative lors de l'importation à partir d'un module TTCN-3 de même édition que le module d'importation. En cas de constatation d'une incompatibilité entre l'identification du langage (y compris l'identification implicite par omission de la spécification du langage) et la syntaxe du module à partir duquel les définitions sont importées, des utilitaires doivent permettre de remédier sans trop de difficulté à cette incompatibilité.

Les identificateurs de langage TTCN-3 suivants sont définis:

- 'TTCN-3:2001' – à utiliser avec des modules conformes à la version 2001 de la présente Recommandation (voir la Bibliographie).
- 'TTCN-3:2003' – à utiliser avec des modules conformes à la version 2003 de la présente Recommandation (voir la Bibliographie).

- 'TTCN-3:2005' – à utiliser avec des modules conformes à la présente Recommandation.

EXEMPLE:

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

NOTE – Le mécanisme d'importation est conçu de façon à permettre de réutiliser des définitions à partir d'autres modules du langage TTCN-3 ou d'autres langages. Les règles applicables à l'importation de définitions à partir de spécifications écrites dans d'autres langages, par exemple les paquetages SDL, peuvent suivre les règles TTCN-3 ou être définies séparément.

8 Configurations de test

8.0 Généralités

La notation TTCN-3 permet la spécification (dynamique) de configurations de test (appelées *configuration* par concision) simultanées. Une configuration se compose d'un ensemble de composants de test interconnectés avec des ports de communication bien définis et d'une interface explicite du système de test qui définit les frontières du système de test.

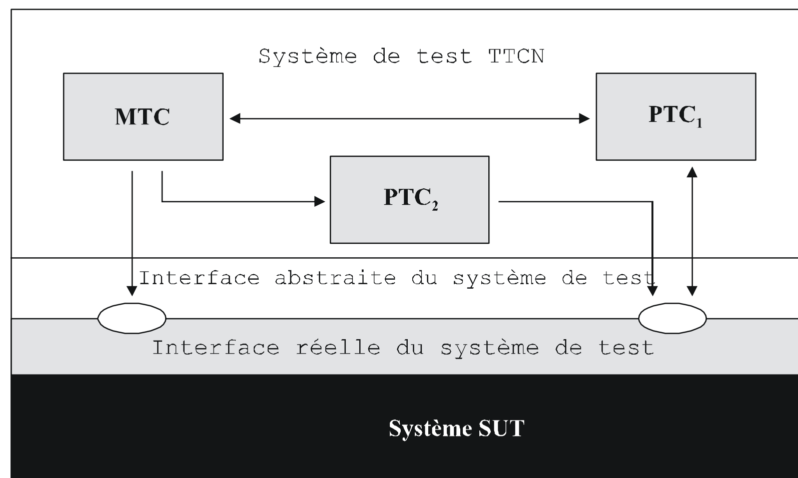


Figure 3/Z.140 – Vue théorique d'une configuration de test TTCN-3 typique

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

L'exécution d'un test élémentaire doit se terminer quand le composant MTC s'achève. Tous les autres composants PTC sont traités également; c'est-à-dire qu'il n'y a pas de relation hiérarchique explicite entre eux et la terminaison d'un seul composant PTC ne met fin ni à d'autres composants ni au composant MTC. Quand le composant MTC s'achève, le système de test doit arrêter tous les composants PTC non achevés au moment où l'exécution des tests élémentaires est arrêtée.

La communication entre les composants de test et entre ces composants et l'interface du système de test est réalisée au moyen de ports de communication (voir § 8.1).

Les types de composant de test et les types de port, indiqués par les mots clés **`component`** et **`port`**, doivent être définis dans la partie du module relative aux définitions. La configuration réelle des composants et des connexions entre eux est réalisée par l'exécution des opérations **`create`** et **`connect`** opérations dans le cadre du comportement de test élémentaire. Les ports de composant sont connectés aux ports de l'interface du système de test au moyen de l'opération **`map`** (voir § 22.2).

8.1 Modèle de communication entre ports

Les composants de test sont connectés au moyen de leurs ports, c'est-à-dire que les connexions entre les composants, et entre ceux-ci et l'interface du système de test, sont fondées sur des ports. Chaque port est modélisé comme une file infinie du type premier entré, premier sorti (FIFO, *first in, first out*) qui mémorise les messages ou appels de procédure entrants jusqu'à ce qu'ils soient traités par le composant possédant ce port.

NOTE – Bien que les ports TTCN-3 soient théoriquement infinis, ces ports peuvent déborder dans un système de test réel. Cet événement devrait être traité comme une erreur de test élémentaire (voir § 25.2.1).

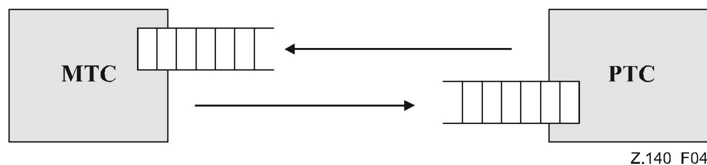


Figure 4/Z.140 – Le modèle des ports de communication TTCN-3

8.2 Restrictions relatives aux connexions

En notation TTCN-3, les connexions s'effectuent de port à port et de port à interface de système de test (voir Figure 5). Il n'y a aucune restriction relative au nombre de connexions qu'un composant peut entretenir. Des connexions point à multipoint sont également autorisées (par exemple Figure 5 (g) ou Figure 5 (h)).

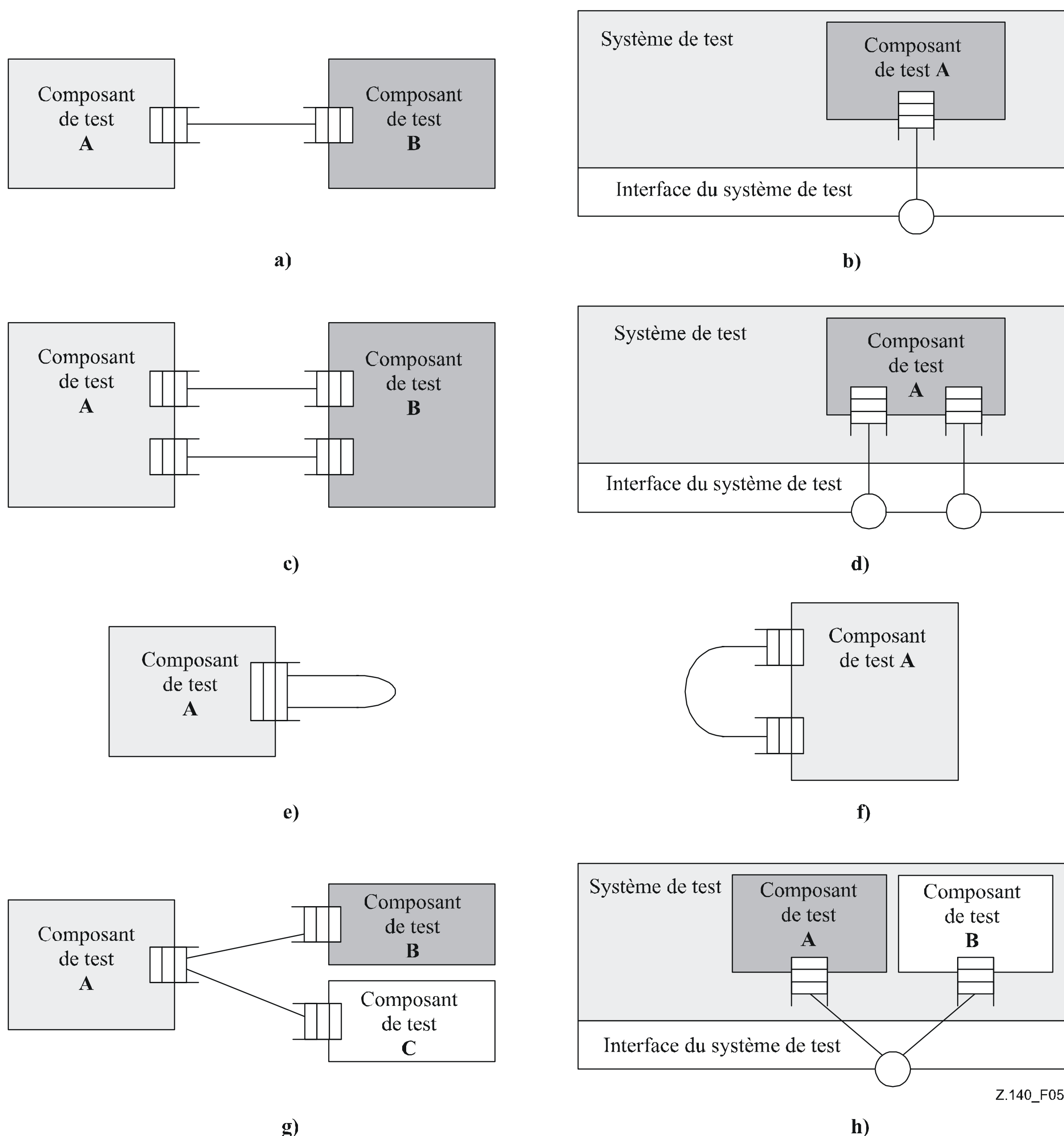


Figure 5/Z.140 – Connexions autorisées

Les connexions suivantes ne sont pas autorisées:

- un port détenu par un composant A ne doit pas être connecté avec deux ou plus de deux ports détenus par le même composant (Figure 6(a) et Figure 6(e));
- un port détenu par un composant A ne doit pas être connecté avec deux ou plus de deux ports détenus par un composant B (voir Figure 6(c));
- un port détenu par un composant A ne peut avoir une connexion biunivoque avec l'interface du système de test. Autrement dit, les connexions représentées dans la Figure 6(b) et dans la Figure 6(d) ne sont pas autorisées;
- les connexions à l'intérieur de l'interface du système de test ne sont pas autorisées (voir Figure 6(f));
- un port qui est connecté ne doit pas être mappé et un port qui est mappé ne doit pas être connecté (voir Figure 6(g)).

Etant donné que la notation TTCN-3 permet des configurations et adresses dynamiques, les restrictions relatives aux connexions ne peuvent pas toujours être vérifiées au moment de la compilation. Les vérifications doivent être effectuées à l'exécution et doivent conduire à une erreur de test élémentaire en cas d'échec.

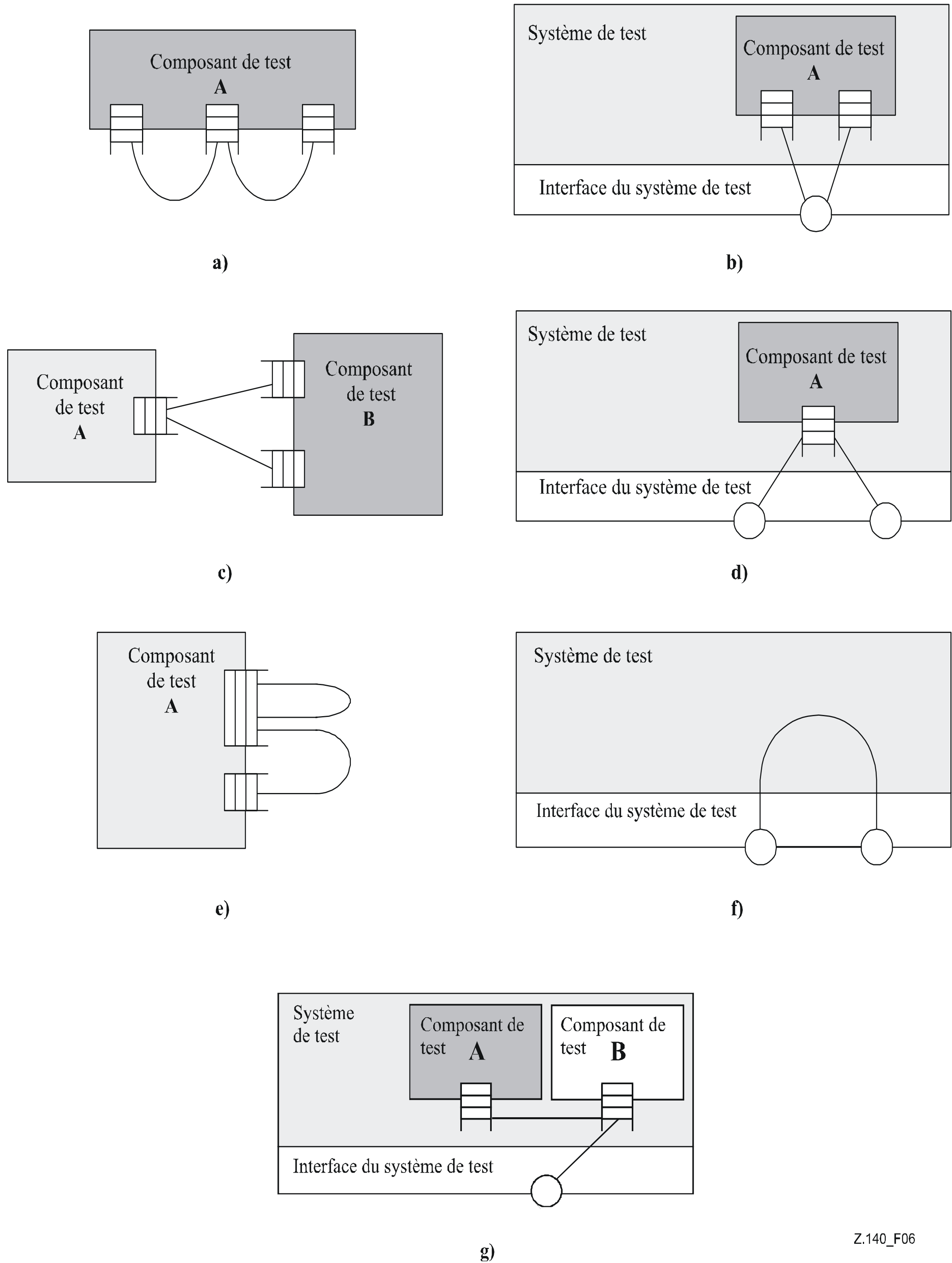


Figure 6/Z.140 – Connexions non autorisées

8.3 Interface du système de test abstrait

La notation TTCN-3 sert à tester des implémentations. L'objet soumis au test est appelé *implémentation sous test* ou *implémentation IUT*. L'implémentation IUT peut offrir des interfaces directes pour les tests ou peut faire partie du système auquel cas l'objet soumis au test est appelé *système sous test* ou *système SUT*. Dans le cas minimal, l'instance IUT et le système SUT sont équivalents. Dans la présente Recommandation, le terme "SUT" est utilisé de façon générale dans le sens d'un système SUT ou d'une instance IUT.

Dans un environnement d'essai réel, les tests élémentaires ont besoin de communiquer avec le système SUT. Cependant, la spécification de la connexion physique réelle est hors du domaine d'application de la notation TTCN-3. Une interface bien définie (mais abstraite) du système de test doit être par contre associée à chaque test élémentaire. Une définition de l'interface du système de test est identique à une définition de composant; c'est-à-dire qu'il s'agit d'une liste de tous les ports de communication possibles au moyen desquels le test élémentaire est connecté au système SUT.

L'interface du système de test définit statiquement le nombre et le type des connexions par port au système SUT pendant un processus d'exécution de test. Cependant, les connexions entre l'interface du système de test et les composants de test TTCN-3 sont de nature dynamique et peuvent être modifiées pendant un processus d'exécution de test au moyen des opérations **map** et **unmap** (voir § 22.2 et 22.3).

8.4 Définition des types de port de communication

8.4.0 Généralités

Les ports facilitent la communication entre les composants de test ainsi qu'entre ceux-ci et l'interface du système de test.

La notation TTCN-3 prend en charge les ports en mode message ou en mode procédure. Chaque port doit être défini comme étant en mode message ou en mode procédure (ou les deux simultanément comme décrit dans le § 8.4.1). Les ports en mode message doivent être identifiés par le mot clé **message** et les ports en mode procédure doivent être identifiés par le mot clé **procedure** dans la définition associée du type de port.

Les ports sont bidirectionnels. Le sens est spécifié par les mots clés **in** (pour le sens entrant), **out** (pour le sens sortant) et **inout** (pour les deux sens). Chaque définition de type de port doit contenir une ou plusieurs listes indiquant le recueil autorisé des types (de message) et/ou des procédures en même temps que le sens de communication autorisé.

Chaque fois qu'une signature (voir aussi § 13) est définie dans le sens "sortant" pour un port en mode procédure, les types de tous ses paramètres **inout** et **out**, son type de retour et ses types d'exception font automatiquement partie du sens entrant de ce port. Chaque fois qu'une signature est définie dans le sens entrant pour un port en mode procédure, les types de tous ses paramètres **inout** et **out**, son type de retour et ses types d'exception font automatiquement partie du sens "sortant" de ce port.

EXEMPLE:

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

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

NOTE – Le terme message désigne aussi bien les messages définis par des modèles que les valeurs effectives résultant d'expressions. La liste limitant les possibilités d'utilisation d'un accès en mode message n'est donc qu'une liste des noms de type.

8.4.1 Ports mixtes

Il est possible de définir un port comme permettant deux sortes de communication, ce qui est indiqué par le mot clé **mixed**. Autrement dit, les listes relatives aux ports mixtes seront également mixtes et comprendront à la fois des signatures et des types. Aucune séparation n'est opérée dans la définition.

```
  // Port mixte définissant un port en mode message et un port en
  // mode procédure avec le même nom. Les listes in, out et inout sont
```

```
// également mixtes: les types MsgType1, MsgType2, MsgType3 et integer se
// rapportent à la partie en mode message du port mixte tandis
// que Proc1, Proc2, Proc3, Proc4 et Proc5 se rapportent au port en mode
// procédure.
type port MyMixedPortType mixed
{
  in  MsgType1, MsgType2, Proc1, Proc2;
  out MsgType3, Proc3, Proc4;
  inout   integer, Proc5;
}
```

En notation TTCN-3, un port mixte est défini comme une abréviation pour deux ports; c'est-à-dire: un port en mode message et un port en mode procédure, avec le même nom. Lors de l'exécution, la distinction entre les deux ports est assurée par les opérations de communication.

Les opérations servant à commander les ports (voir § 23.5), c'est-à-dire **start**, **stop** et **clear**, doivent s'appliquer aux deux files (dans un ordre arbitraire) si elles sont appelées avec un identificateur de port mixte.

8.5 Définition des types de composant

8.5.0 Généralités

Le type **component** définit les ports qui sont associés à un composant. Ces définitions doivent être effectuées dans la partie du module relative aux définitions. Les noms de port contenus dans une définition de composant sont locaux par rapport à ce composant c'est-à-dire qu'un autre composant peut avoir des ports portant les mêmes noms. Les ports du même composant doivent tous avoir des noms uniques. La seule définition d'un composant n'implique pas qu'il y ait une connexion entre les composants à ces ports.

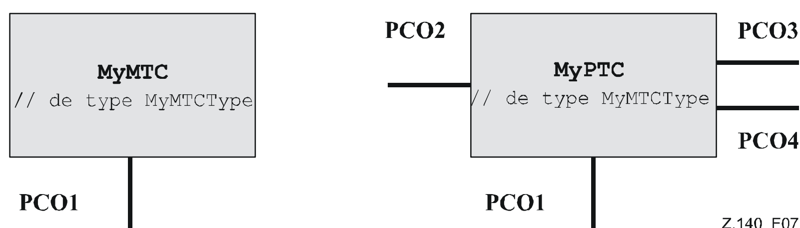


Figure 7/Z.140 – Composants typiques

EXEMPLE:

```
type component MyMTCType
{
  port MyMessagePortType          PCO1
}

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

8.5.1 Déclaration de variables, de constantes et de temporisations localisées dans un composant

Il est possible de déclarer des constantes, des variables et des temporisations localisées dans un composant particulier.

EXEMPLE:

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

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

Les variables et temporisations d'un composant sont associées à l'instance de celui-ci et suivent les règles de portée définies au § 5.3. Chaque nouvelle instance d'un composant va donc avoir son propre ensemble de variables et de temporisations comme spécifié dans la définition du composant (y compris d'éventuelles valeurs initiales, si déclarées).

NOTE – Quand ils sont utilisés comme interface du système de test (voir § 8.8), les composants ne peuvent pas faire usage d'éventuelles constantes, variables et temporisations déclarées dans le composant.

8.5.2 Définition de composants avec matrices de ports

Il est possible de définir des matrices de ports dans les définitions de type de composant (voir également § 22.12).

EXEMPLE:

```
type component My3pcoCompType
{
   port MyMessageInterfaceType PCO[3]
   port MyProcedureInterfaceType PCOm[3][3]
   // Définit un type de composant qui a une matrice de 3 ports
   // en mode message et une matrice bidimensionnelle
   // de 9 ports en mode procédure.
}
```

8.5.3 Extension de types de composant

Il est possible de définir des types de composant comme l'extension d'autres types de composant, au moyen du mot clé **extends**.

EXEMPLE 1:

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

Dans une telle définition, le nouveau type est défini comme étant le *type étendu*, et le type qui suit le mot clé **extends** est défini comme étant le *type parent*.

Cette définition a pour effet que le type étendu contiendra aussi implicitement toutes les définitions extraites du type parent. Ainsi, la définition ci-dessus est équivalente à la définition suivante appelée la *définition de type effective*:

EXEMPLE 2:

```
// effectivement, la définition de l'Exemple 1 équivaut à celle-ci:
type component MyExtendedMTCType
{
  /* les définitions de MyMTCType */
  var integer MyLocalInteger;
  timer MyLocalTimer;
  port MyMessagePortType PCO1

  /* les définitions additionnelles */
  var float MyLocalFloat;
  timer MyOtherLocalTimer;
  port MyMessagePortType PCO2;
}
```

Il est permis d'étendre des types de composant qui sont définis par extension, pour autant que cela ne crée aucune chaîne cyclique de définition.

EXEMPLE 3:

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

Lors de la définition de types de composant par extension, il ne doit y avoir aucun conflit de noms entre les définitions extraites du type parent et les définitions qui sont ajoutées dans le type étendu, c'est-à-dire qu'aucun identificateur de port, de variable, de constante, de temporisation ou de modèle ne doit être déclaré à la fois dans le type parent (directement ou par extension) et dans le type étendu.

EXEMPLE 4:

```
type component MyExtendedMTCType extends MyMTCType
{
  var integer MyLocalInteger;       // ERROR - déjà défini dans MyMTCType (voir exemple 2)
  var float MyLocalTimer;           // ERROR - un temporisateur de ce nom existe dans MyMTCType
  port MyOtherMessagePortType PCO1; // ERROR - un port de ce nom existe dans MyMTCType
}
```

```
type component MyBaseComponent { timer MyLocalTimer };
type component MyInterimComponent extends MyBaseComponent { timer MyOtherTimer };
type component MyExtendedComponent extends MyInterimComponent
{
  timer MyLocalTimer; // ERROR - déjà défini dans MyInterimComponent par extension
}
```

Il est permis d'avoir un type de composant étendant plusieurs types parent dans une même définition, auquel cas les types parent doivent être spécifiés sous forme d'une liste de types séparés par une virgule:

EXEMPLE 5:

```
type component MyCompA extends MyCompB, MyCompC, MyCompD {
  /* définitions additionnelles pour MyCompA */
}
```

L'un quelconque des types parent peut aussi être défini par extension.

La définition effective du type de composant étendu reprend l'ensemble de toutes les définitions de constante, variable, temporisation, port et modèle extraites des types parent (déterminées récursivement si un type parent est également défini par extension) et les définitions déclarées directement dans le type étendu. La définition effective du type de composant doit être exempte de conflit de noms. Pour satisfaire à cette condition, dans le cadre de l'ensemble des types parent utilisés dans la définition du type étendu, toutes les définitions doivent avoir des noms uniques et ces noms doivent être différents des noms des définitions déclarées directement dans le type étendu.

NOTE 1 – N'est pas considérée comme une déclaration différente, et par conséquent ne provoque aucune erreur, une même définition ajoutée au type étendu par des types parent différents (via des chemins d'extension différents).

EXEMPLE 6:

```
type component MyCompB { timer T };
type component MyCompC { var integer T };
type component MyCompD extends MyCompB, MyCompC {}
  // ERROR - conflit de noms entre MyCompB et MyCompC

// MyCompB est défini ci-dessus
type component MyCompE extends MyCompB {
  var integer MyVar1 := 10;
}

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

type component MyCompG extends MyCompB, MyCompE, MyCompF {
  // Pas de conflit de noms.
  // Les trois types parent de MyCompG ont un temporisateur T,
  // soit directement soit par extension de MyCompB; du fait qu'ils
  // résultent tous les trois (directement ou par extension)
  // du temporisateur T déclaré dans MyCompB,
  // cette forme de collision est légale.
  /* définitions supplémentaires ici */
}
```

La sémantique des types de composant avec extensions est définie simplement en remplaçant la définition de chaque type de composant par sa définition effective dans une phase de traitement préalable, avant de l'utiliser.

NOTE 2 – Aux fins de la compatibilité des types de composant, cela signifie qu'une référence de composant "c" du type CT1, qui étend CT2, est compatible avec CT2, et que les tests élémentaires, fonctions et variantes qui spécifient CT2 dans leurs clauses **runs on** peuvent être exécutés sur "c" (voir § 6.7.3).

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

Un système SUT peut se composer de plusieurs entités qui doivent être adressées individuellement. Le type des données d'adressage est à utiliser avec les opérations de port afin de s'adresser à des entités SUT. Quand il est utilisé avec les mots clés **to**, **from** et **sender** le type des données d'adressage ne doit être utilisé que dans les opérations de réception et d'émission des ports mappés à l'interface avec le système de test. La représentation effective des données de type **address** est résolue soit par une définition de type explicite dans la suite de tests ou par le système de test externe (c'est-à-dire que le type **address** reste dans la spécification TTCN-3 sous la forme d'un type ouvert). Cela permet de spécifier des tests élémentaires abstraits indépendamment de tout mécanisme réel d'adressage propre au système SUT.

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

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

EXEMPLE:

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

8.7 Références de composant

Les références de composant sont uniques par rapport aux composants de test créés pendant l'exécution d'un test élémentaire. Cette unique référence de composant est produite par le système de test au moment où un composant est créé, c'est-à-dire qu'une référence de composant est le résultat d'une opération de type **create** (voir § 22.1). En outre, les références de composant sont retournées par les opérations prédéfinies du type **system** (qui retourne la référence de composant afin d'identifier les accès de l'interface avec le système de test), par les opérations du type **mtc** (qui retourne la référence de composant du composant MTC) et par les opérations du type **self** (qui retourne la référence du composant dans lequel le type **self** est appelé).

Les références de composant sont utilisées dans les opérations de configuration **connect**, **map** et **start** (voir § 22) afin d'établir des configurations de test. Elles sont également utilisées dans les parties **from**, **to** et **sender** des opérations de communication aux ports connectés à des composants de test autres que l'interface du système de test, aux fins de l'adressage (voir § 23 et Figure 5).

En outre, la valeur spéciale **null** permet d'indiquer une référence de composant indéfinie, par exemple pour l'initialisation de variables pour le traitement des références de composant.

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

NOTE – Une référence de composant comprend les informations relatives au type de composant. Autrement dit, par exemple, une variable pour le traitement des références de composant doit impérativement utiliser le nom du type de composant correspondant dans sa déclaration.

EXEMPLE:

```
// Une définition de type de composant
type component MyCompType {
  port PortTypeOne PCO1;
  port PortTypeTwo PCO2
}

// Déclaration d'une variable pour le traitement de références à des
// composants de type MyCompType et pour la création d'un composant de ce
// de ce type
var MyCompType MyCompInst := MyCompType.create;

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

// Utilisation de références de composant dans des clauses from et to
MyPCO1.receive from MyCompInst;
 :
MyPCO2.receive(integer:?) -> sender MyCompInst;
 :
MyPCO1.receive(MyTemplate) from MyCompInst;
```

```
  :
MPCO2.send(integer:5) to MyCompInst;

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

8.8 Définition de l'interface du système de test

Une définition de type de composant sert à définir l'interface du système de test parce que, théoriquement, les définitions de type de composant et d'interface du système de test ont la même forme (ce sont toutes les deux des recueils de ports définissant des points de connexion possibles).

NOTE – Les variables, les temporisations et les constantes déclarées dans des types de composant, qui sont utilisées comme interfaces de système de test, seront sans effet.

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

Généralement, une référence de type de composant définissant l'interface du système de test doit être associée à chaque test élémentaire utilisant plus d'un seul composant de test. Les ports de l'interface du système de test doivent automatiquement être instanciés par le système en même temps que le composant MTC quand l'exécution du test élémentaire commence.

L'opération retournant la référence de composant de l'interface du système de test est de type **system**. Elle doit servir pour l'adressage des ports du système de test.

EXEMPLE:

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

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

9 Déclaration des constantes

Les constantes peuvent être déclarées et utilisées dans la partie d'un module relative aux définitions, dans les définitions de type de composant, dans la partie d'un module relative à la commande, dans les tests élémentaires, dans les fonctions et dans les variantes. Les définitions des constantes sont indiquées par le mot clé **const**. Les constantes ne doivent pas être du type port. La valeur de la constante doit être affectée au point de déclaration.

NOTE – La seule valeur qui peut être affectée à des constantes du type par défaut et du type composant est la valeur spéciale **null**.

EXEMPLE 1:

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

L'affectation de la valeur à la constante peut être effectuée dans le module ou à l'extérieur du module. Dans ce dernier cas, il s'agit d'une déclaration de constante externe, indiquée par le mot clé **external**.

EXEMPLE 2:

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

Une constante externe peut être d'un type quelconque sauf du type port, du type par défaut ou du type composant et ce type doit être connu dans le module, c'est-à-dire qu'il doit être un type racine ou un type défini par l'utilisateur dans le module, ou un type importé à partir d'un autre module. Le mappage du type vers la représentation externe d'une constante externe et le mécanisme de transmission de la valeur d'une constante externe dans un module ne relèvent pas du domaine d'application de la présente Recommandation.

10 Déclaration de variables

10.0 Généralités

Les variables peuvent être de simples types de base, des types de chaîne de base, des types structurés, des types de données spéciaux (y compris des sous-types issus de ces types) ainsi que des types adresse, composant et par défaut.

NOTE – Les variables des types structurés et composant ne peuvent être déclarées qu'à partir de types définis par l'utilisateur.

Les variables peuvent être déclarées et utilisées dans la partie d'un module relative à la commande, dans les tests élémentaires, dans les fonctions et dans les variantes. En outre, des variables peuvent être déclarées dans les définitions de type de composant. Ces variables peuvent être utilisées dans les tests élémentaires, dans les variantes et dans les fonctions qui exploitent le type de composant indiqué. Les variables ne doivent pas être déclarées ou utilisées dans la partie d'un module relative aux définitions (c'est-à-dire que les variables globales ne sont pas prises en charge en notation TTCN-3).

L'utilisation de variables non initialisées ou pas complètement initialisées en d'autres emplacements que dans la partie gauche des affectations ou comme paramètres effectifs transmis aux paramètres "out" formels doit provoquer une erreur.

10.1 Variables de valeur

Une variable de valeur est déclarée par le mot clé **var** suivi par un identificateur de type et un identificateur de variable. Une valeur initiale peut être affectée lors de la déclaration. Les variables de valeur doivent mémoriser uniquement des valeurs et peuvent être utilisées aussi bien dans la partie droite que dans la partie gauche d'affectations, dans des expressions, après le mot clé **return** dans des corps de fonctions comportant une clause retour dans leurs en-têtes et peuvent être transmises à des paramètres formels de valeur et de type modèle.

EXEMPLE:

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

10.2 Variables de modèle

Les variables de modèle sont déclarées par le mot clé **var template** suivi par un identificateur de type et un identificateur de variable. Un contenu initial peut être affecté lors de la déclaration. En plus de valeurs, les variables de modèle peuvent aussi mémoriser des mécanismes d'appariement (voir § 14.3). Elles peuvent être utilisées aussi bien dans la partie droite que dans la partie gauche d'affectations, après le mot clé **return** dans des corps de fonctions définissant une valeur de retour de type modèle dans leurs en-têtes et peuvent être transmises comme paramètres effectifs à des paramètres formels de type modèle. Lorsqu'elles sont utilisées dans la partie droite d'affectations, elles ne doivent pas être des opérandes d'opérateurs TTCN-3 (voir § 15) et la variable située dans la partie gauche doit être une variable de modèle également. Il est aussi permis d'affecter une instance de modèle à une variable de modèle ou à un champ de variable de modèle.

NOTE – Les variables de modèle, de même que les modèles globaux et locaux, doivent être entièrement spécifiées pour pouvoir être utilisées dans des opérations d'émission et de réception.

EXEMPLE:

```
template MyRecord MyTempl ( template boolean par_bool ) :=
   { field1 := par_bool, field2 := * }
:
function Myfunc () return template MyRecord {
```

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

Il n'est pas permis d'appliquer directement des opérations TTCN-3 à des variables de modèle, mais il est permis d'utiliser la notation à points et la notation indexée pour inspecter et modifier des champs de variable de modèle. Les règles à appliquer lorsque ces notations tentent d'atteindre des champs en dehors d'un mécanisme d'appariement sont indiquées dans le § 14.3.1.

11 Déclaration des temporisations

11.0 Généralités

Les temporisations peuvent être déclarées et utilisées dans la partie d'un module relative à la commande, dans les tests élémentaires, dans les fonctions et dans les variantes. En outre, des temporisations peuvent être déclarées dans les définitions de type de composant. Ces temporisations peuvent être utilisées dans les tests élémentaires, dans les fonctions et dans les variantes qui exploitent le type de composant indiqué. Une déclaration de temporisation peut avoir une valeur facultative de durée par défaut qui lui est affectée. Le temporisateur doit être déclenché avec cette valeur si aucune autre valeur n'est spécifiée. Cette valeur doit être du type **`float`** non négatif (c'est-à-dire supérieure ou égale à 0,0), l'unité de base étant la seconde.

EXEMPLE 1:

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

timer MyTimer2;             // déclaration de MyTimer2 sans valeur par défaut de
                            // temporisation, c'est-à-dire qu'une valeur doit être
                            // affectée quand le temporisateur est déclenché
```

En plus des instances de temporisation isolées, des matrices de temporisations peuvent également être déclarées. La ou les durées par défaut des éléments d'une matrice de temporisations doivent être affectées au moyen d'une matrice de valeurs. L'affectation d'une ou de plusieurs durées par défaut doit utiliser la notation de valeurs de matrice spécifiée au § 6.5. Si l'on souhaite omettre l'affectation de durée par défaut pour certains éléments de la matrice de temporisations, cela doit être déclaré explicitement au moyen du symbole de non-utilisation ("-").

EXEMPLE 2:

```
timer t_Mytimer1[5] := { 1.0, 2.0, 3.0, 4.0, 5.0 }
  // tous les éléments de la matrice de temporisations reçoivent
  // une durée par défaut.

timer t_Mytimer2[5] := { 1.0, -, 3.0, 4.0, 5.0 }
  // la deuxième temporisation (t_Mytimer2[1]) est laissée sans durée
  // par défaut.
```

11.1 Temporisations utilisées comme paramètres

Les temporisations ne peuvent être transmises que par référence à des fonctions et à des variantes. Les temporisations transmises dans une fonction ou dans une variante sont connues à l'intérieur de la définition de comportement de la fonction ou variante.

Les temporisations transmises en tant que paramètres par référence peuvent être utilisées comme toute autre temporisation, c'est-à-dire qu'elles n'ont pas besoin d'être déclarées. Un temporisateur déclenché peut également être transmis dans une fonction ou variante. La temporisation continue à fonctionner, c'est-à-dire qu'elle n'est pas arrêtée implicitement. D'éventuels événements d'expiration de temporisateur peuvent donc être traités à l'intérieur de la fonction ou variante à laquelle la temporisation est transmise.

EXEMPLE:

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

12 Déclaration des messages

Un des éléments clés de la notation TTCN-3 est la capacité d'émettre et de recevoir des messages complexes par les ports de communication définis dans la configuration de test. Ces messages peuvent être ceux qui concernent explicitement les tests du système SUT ou les messages internes de coordination et de commande propres à la configuration de test applicable.

NOTE – En notation TTCN-2, ces messages sont les primitives de service abstraites (ASP, *abstract service primitive*), les unités de données protocolaires (PDU, *protocol data unit*) et messages de coordination. Le langage noyau de la notation TTCN-3 est générique en ce sens qu'il ne formule aucune distinction syntaxique ou sémantique de cette sorte.

13 Déclaration des signatures de procédure

13.0 Généralités

Les signatures de procédure (abrégées en *signatures*) sont requises pour la communication en mode procédure. La communication en mode procédure peut servir à la communication dans le système de test, c'est-à-dire entre les composants de test, ou pour la communication entre le système de test et le système SUT. Dans ce dernier cas, une procédure peut soit être invoquée dans le système SUT (c'est-à-dire que c'est le système de test qui effectue l'appel) ou être invoquée dans le système de test (c'est-à-dire que c'est le système SUT qui effectue l'appel). Pour toutes les procédures utilisées, c'est-à-dire les procédures utilisées pour la communication entre les composants de test, les procédures appelées à partir du système SUT et les procédures appelées à partir du système de test, le type **signature** de procédure complète doit être défini dans le module TTCN-3.

13.1 Signatures pour communications bloquantes et non bloquantes

La notation TTCN-3 prend en charge les communications *bloquantes* et *non bloquantes* en mode procédure. Les définitions de signature pour communications non bloquantes doivent utiliser le mot clé **noblock**, ne doivent avoir que des paramètres de type **in** (voir § 13.2) et ne doivent avoir aucune valeur de retour (voir § 13.3). Mais ces définitions peuvent déclencher des exceptions (voir § 13.4). Par défaut, les définitions de signature sans le mot clé **noblock** sont censées servir aux communications bloquantes en mode procédure.

EXEMPLE:

```
signature MyRemoteProcOne ();          // La procédure MyRemoteProcOne sera
                                        // utilisée pour les communications
                                        // bloquantes en mode procédure. Elle n'a
                                        // n'a ni paramètres ni valeur de retour

signature MyRemoteProcTwo () noblock;   // La procédure MyRemoteProcTwo sera
                                        // utilisée pour une communication
                                        // non bloquante en mode
                                        // procédure. Elle n'a ni
                                        // paramètres ni valeur de retour.
```

13.2 Paramètres des signatures de procédure

Les définitions de signature peuvent avoir des paramètres. A l'intérieur d'une définition de type **signature,** la liste des paramètres peut comprendre des identificateurs de paramètre et des types de paramètre avec le sens correspondant, c'est-à-dire **in**, **out**, ou **inout**. Les sens **inout** et **out** indiquent que ces paramètres servent à extraire des informations à partir de la procédure distante. Noter que le sens des paramètres est celui qui est perçu par l'*appelé* plutôt que par l'*appelant*.

EXEMPLE:

```
signature MyRemoteProcThree (in integer Par1, out float Par2, inout integer Par3);
// La procédure MyRemoteProcThree sera utilisée pour la communication
// bloquante en mode procédure. Cette procédure a trois
// paramètres: Par1 qui est un paramètre entrant de type integer, Par2 qui
// est un paramètre sortant de type float et Par3 qui est un paramètre
// bilatéral de type integer.
```

13.3 Procédures distantes retournant une valeur

Une procédure distante peut retourner une valeur après sa terminaison. Le type de la valeur de retour doit être spécifié au moyen d'une clause `return` dans la définition de signature correspondante.

EXEMPLE:

```
signature MyRemoteProcFour (in integer Par1) return integer;
// La procédure MyRemoteProcFour sera utilisée pour une communication
// bloquante en mode procédure. La procédure a le paramètre entrant Par1 de
// type integer et retourne une valeur de type integer après sa terminaison
```

13.4 Spécification des exceptions

Les exceptions qui peuvent être déclenchées par des procédures distantes sont représentées en notation TTCN-3 sous la forme de valeurs d'un type spécifique. Des modèles et des mécanismes d'appariement peuvent donc servir à spécifier ou à vérifier des valeurs de retour de procédure distante.

NOTE – La conversion d'exceptions produites par le système SUT ou envoyées à celui-ci afin d'obtenir la représentation TTCN-3 correspondante du type ou du système SUT est propre à l'utilitaire et au système utilisé et est donc hors du domaine d'application de la présente Recommandation.

Les exceptions sont définies sous la forme d'une liste d'exceptions incluse dans la définition de type **signature**. Cette liste définit tous les différents types qui peuvent être associés à l'ensemble des exceptions possibles (dont la signification ne sera habituellement distinguée qu'au moyen de valeurs spécifiques de ces types).

EXEMPLE:

```
signature MyRemoteProcFive (inout float Par1) return integer
      exception (ExceptionType1, ExceptionType2);
// La procédure MyRemoteProcFive sera utilisée pour la communication
// bloquante en mode procédure.Elle retourne une valeur en virgule
// flottante dans le paramètre bilatéral Par1 et une valeur d'entier, ou
// peut déclencher des exceptions de type ExceptionType1 ou ExceptionType2

signature MyRemoteProcSix (in integer Par1) noblock
      exception (integer, float);
// La procédure MyRemoteProcSix sera utilisée pour la communication
// non bloquante en mode procédure. Dans le cas d'un échec de terminaison,
// la procédure MyRemoteProcSix déclenche des exceptions de type integer
// ou float.
```

14 Déclaration des modèles

14.0 Généralités

Les modèles servent soit à transmettre un ensemble de valeurs distinctes ou à vérifier si un ensemble de valeur reçue correspond au modèle spécifié. Les modèles peuvent être définis globalement dans la partie définitions d'un module, localement dans un test élémentaire, une fonction, une variante ou un bloc d'instructions ou en ligne sous forme d'arguments d'une opération de communication ou de paramètre effectif d'un appel de test élémentaire, de fonction ou de variante.

Les modèles offrent les possibilités suivantes:

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

b) ils peuvent être paramétrés;

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

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

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

Une déclaration de modèle doit impérativement spécifier un ensemble valeurs de base ou de symboles correspondants pour chacun des champs définis dans le type approprié ou dans la définition de signature appropriée; c'est-à-dire qu'il s'agit d'une spécification totale. Une déclaration de modèle modifié (voir § 14.6) spécifie seulement les champs à modifier à partir du modèle de base; c'est-à-dire qu'il s'agit d'une spécification partielle. Le symbole de non-utilisation doit uniquement être utilisé dans les modèles de signature pour des paramètres qui ne sont pas applicables et dans les déclarations de modèle modifié et les modèles en ligne modifiés afin d'indiquer l'absence de modification dans le champ ou élément spécifié.

Il existe un certain nombre de restrictions applicables aux fonctions utilisées dans des expressions lors de la spécification de modèles ou de champs de modèle; ces restrictions sont spécifiées dans le § 16.1.4.

14.1 Déclaration des modèles de message

14.1.0 Généralités

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

Les modèles peuvent être spécifiés pour tout type TTCN-3 défini dans le Tableau 3 sauf pour les types **`port`** et **`default`**.

EXEMPLE:

```
// Quand il est utilisé dans une opération de réception, ce modèle va
// correspondre à une valeur d'entier quelconque
template integer Mytemplate := ?;
// Ce modèle va correspondre seulement à la valeur d'entier 1, 2 ou 3
template integer Mytemplate := (1, 2, 3);
```

14.1.1 Modèles pour envoi de messages

Un modèle utilisé dans une opération de type **`send`** définit un ensemble complet de valeurs de champ constituant le message à transmettre par un port de test. Au moment de l'opération de type **`send`**, le modèle doit être totalement défini, c'est-à-dire que tous les champs doivent correspondre à des valeurs effectives et aucun mécanisme d'appariement ne doit être utilisé dans les champs de modèle, ni directement ni indirectement.

NOTE – Pour les modèles d'envoi, l'omission d'un champ facultatif est considérée comme une notation de valeur plutôt que comme un mécanisme d'appariement.

EXEMPLE:

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

// un modèle de message pourrait être
template MyMessageType MyTemplate:=
{
  field1 := omit,
  field2 := "My string",
  field3 := true
}

// et une opération d'envoi correspondante pourrait être
MyPCO.send(MyTemplate);
```

14.1.2 Modèles pour la réception de messages

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

EXEMPLE:

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

// un modèle de message pourrait être
template MyMessageType MyTemplate:=
{
  field1 := ?,
  field2 := pattern "abc*xyz",
```

```
  field3 := true
}

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

14.2 Déclaration des modèles de signature

14.2.0 Généralités

Les instances des listes de paramètres de procédure contenant des valeurs effectives peuvent être spécifiées au moyen de modèles. Ceux-ci peuvent être définis pour toute procédure au moyen d'une référence à la définition de signature associée. Un modèle de signature définit uniquement les valeurs et les mécanismes d'appariement des paramètres de prodécure, mais pas la valeur de retour. Les valeurs ou les mécanismes d'appariement pour un retour doivent être définis dans le cadre de l'opération **`reply`** ou **`getreply`** (voir § 23.3.3 et 23.3.4, respectivement).

EXEMPLE:

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

// Exemple de modèles associés à une signature de procédure définie
template RemoteProc Template1:=
{
  Par1 := 1,
  Par2 := 2,
  Par3 := 3
}

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

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

14.2.1 Modèles d'invocation de procédures

Un modèle utilisé dans une opération d'appel **`call`** ou de réponse **`reply`** définit un ensemble complet de valeurs de champ pour tous les paramètres **`in`** et **`inout`**. Au moment de l'opération d'appel **`call,`** tous les paramètres **`in`** et **`inout`** contenus dans le modèle doivent correspondre à des valeurs effectives et aucun mécanisme d'appariement ne doit être utilisé dans ces champs, directement ou indirectement. Toute spécification de modèle pour paramètres **`out`** est simplement ignorée; il est donc permis de spécifier des mécanismes d'appariement pour ces champs, ou de les omettre (voir Annexe B).

EXEMPLE:

```
// Dans les exemples du § 14.2.0

 // Invocation valide étant donné que tous les paramètres in et inout ont
 // une valeur distincte
MyPCO.call(RemoteProc:Template1);

 // Invocation valide étant donné que tous les paramètres in et inout ont
 // une valeur distincte
MyPCO.call(RemoteProc:Template2);

 // Invocation non valide parce que le paramètre inout Par3 a un
 // attribut d'appariement et non une valeur
MyPCO.call(RemoteProc:Template3);

 // Les modèles ne retournent jamais de valeurs. Dans le cas de Par2 et
 // Par3, les valeurs retournées par l'opération d'appel doivent
 // impérativement être extraites au moyen d'une clause d'affectation située
 // à la fin de l'instruction d'appel
```

14.2.2 Modèles d'acceptation des invocations de procédure

Un modèle utilisé dans une opération de type **getcall** définit un modèle de données auquel les champs du paramètre entrant sont appariés. Les mécanismes d'appariement, définis dans l'Annexe B, peuvent être utilisés dans tout modèle utilisé par cette opération. Aucune association de valeurs entrantes avec le modèle ne doit se produire. Les éventuels paramètres de type **out** doivent être ignorés dans le processus d'appariement.

EXEMPLE:

```
// Dans les exemples du § 14.2.0

// Opération getcall valide, qui va correspondre si Par1 == 1 et Par3 == 3
MyPCO.getcall(RemoteProc:Template1);

// Opération getcall valide, qui va correspondre si Par1 == 1 et
// Par3 == 3
MyPCO.getcall(RemoteProc:Template2);

// Opération getcall valide, qui va correspondre si Par1 == 1 et
// si Par3 a une valeur de type Any
MyPCO.getcall(RemoteProc:Template3);
```

14.3 Mécanismes d'appariement de modèles

14.3.0 Généralités

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

Les mécanismes d'appariement et les structures génériques ne peuvent donc être utilisés en ligne que dans les événements reçus (c'est-à-dire lors des opérations de type **receive**, **trigger**, **getcall**, **getreply** et **catch**). Ils peuvent apparaître dans des valeurs explicites.

EXEMPLE 1:

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

L'identificateur de type peut être omis quand la valeur identifie le type sans ambiguïté.

EXEMPLE 2:

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

NOTE – Les types suivants peuvent être omis: integer, float, boolean, bitstring, hexstring, octetstring.

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

Les mécanismes d'appariement sont répartis en quatre groupes:

a) valeurs spécifiques:
   - une expression dont l'évaluation donne une valeur spécifique;
   - **omit**: la valeur est omise;
b) symboles spéciaux qui peuvent être utilisés *au lieu* de valeurs:
   - **(…)**: une liste de valeurs;
   - **complement (…)**: complément d'une liste de valeurs;
   - **?**: structure générique représentant une valeur quelconque;
   - *****: structure générique représentant une valeur quelconque ou aucune valeur du tout (c'est-à-dire une valeur omise);
   - (*lowerBound* .. *upperBound*): une étendue de valeurs d'entier ou en virgule flottante entre et y compris les limites inférieure et supérieure;
   - **superset**: au moins tous les éléments énumérés, voire plus;
   - **subset**: au plus les éléments énumérés, voire moins;

c) symboles spéciaux qui peuvent être utilisés *à l'intérieur de* valeurs:

– **?**: structure générique représentant un élément particulier dans une chaîne, dans une matrice, dans un type **`record of`** ou **`set of`**;

– *****: structure générique représentant un nombre quelconque d'éléments consécutifs contenus dans une chaîne, dans une matrice, dans un type **`record of`** ou **`set of`**, ou ne représentant aucun élément que ce soit (c'est-à-dire un élément omis);

– **permutation**: tous les éléments énumérés mais dans un ordre arbitraire (noter que les structures génériques ? et * sont également permises comme éléments de la liste de permutation);

d) symboles spéciaux qui décrivent des *attributs* de valeurs:

– **length**: restrictions de longueur de chaîne pour les types de chaîne; restriction du nombre d'éléments dans les types **`record of, set of`** et dans les matrices;

– **ifpresent**: appariement de valeurs facultatives de champ (si elles ne sont pas omises).

Les mécanismes d'appariement pris en charge, ainsi que leurs (éventuels) symboles associés et leur portée d'application sont représentés dans le Tableau 6. La colonne de gauche de ce tableau énumère tous les types TTCN-3 auxquels ces mécanismes d'appariement s'appliquent. Une description complète de chaque mécanisme d'appariement figure dans l'Annexe B.

Tableau 6/Z.140 – Mécanismes d'appariement TTCN-3

Mécanisme utilisé avec valeurs de	Valeur		Au lieu des valeurs							A l'intérieur de valeurs			Attributs	
	Specific value	Omit value	Complemented list	Value list	Any value(?)	Any value Or None(*)	Range	Superset	Subset	Any element(?)	Any elements Or None(*)	Per mutation	Length restriction	If present
`boolean`	Oui	Oui	Oui	Oui	Oui	Oui[a)]								Oui[b)]
`integer`	Oui	Oui	Oui	Oui	Oui	Oui[a)]	Oui							Oui[b)]
`float`	Oui	Oui	Oui	Oui	Oui	Oui[a)]	Oui							Oui[b)]
`bitstring`	Oui	Oui	Oui	Oui	Oui	Oui[a)]				Oui	Oui		Oui	Oui[b)]
`octetstring`	Oui	Oui	Oui	Oui	Oui	Oui[a)]				Oui	Oui		Oui	Oui[b)]
`hexstring`	Oui	Oui	Oui	Oui	Oui	Oui[a)]				Oui	Oui		Oui	Oui[b)]
`character strings`	Oui	Oui	Oui	Oui	Oui	Oui[a)]	Oui			Oui	Oui		Oui	Oui[b)]
`record`	Oui	Oui	Oui	Oui	Oui	Oui[a)]								Oui[b)]
`record of`	Oui	Oui	Oui	Oui	Oui	Oui[a)]				Oui	Oui	Oui	Oui	Oui[b)]
`array`	Oui	Oui	Oui	Oui	Oui	Oui[a)]				Oui	Oui		Oui	Oui[b)]
`set`	Oui	Oui	Oui	Oui	Oui	Oui[a)]								Oui[b)]
`set of`	Oui	Oui	Oui	Oui	Oui	Oui		Oui	Oui	Oui	Oui		Oui	Oui[b)]
`enumerated`	Oui	Oui	Oui	Oui	Oui	Oui[a)]								Oui[b)]
`union`	Oui	Oui	Oui	Oui	Oui	Oui[a)]								Oui[b)]
`anytype`	Oui	Oui	Oui	Oui	Oui	Oui[a)]								Oui[b)]

a) Lorsqu'il est utilisé, ce mécanisme doit être appliqué à des champs facultatifs de types record et set uniquement (sans restriction imposée au type de ce champ).

b) Lorsqu'il est utilisé, ce mécanisme doit être appliqué aux champs record et set uniquement (sans restriction imposée au type de ce champ).

14.3.1 Référence à des éléments de modèles ou de champs de modèle

14.3.1.1 Référence à des éléments individuels de chaîne

Il n'est pas permis de faire référence à des éléments individuels de chaîne à l'intérieur de modèles ou de champs de modèle.

EXEMPLE:

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

t_Char2 := t_Char1[1];
// doit provoquer une erreur car la référence à des éléments individuels
// de chaîne n'est pas autorisée.
```

14.3.1.2 Référence à des champs `record` et `set`

Les modèles et les variables de modèle permettent de faire référence à des sous-champs à l'intérieur d'une définition de modèle au moyen de la notation à points. Cependant, le champ auquel il est fait référence peut-être un sous-champ d'un champ structuré auquel un mécanisme d'appariement est affecté. Le présent paragraphe indique les règles applicables à ces cas.

- omit, AnyValueOrNone, listes de valeurs et listes de valeurs complémentées: la référence à un sous-champ contenu dans un champ structuré auquel **omit**, AnyValueOrNone (*), une liste de valeurs ou une liste de valeurs complémentées est affecté, doit provoquer une erreur.

EXEMPLE 1:

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

:
var template R1 t_R1 := {
  f1 := 5,
  f2 := omit
}
var template R2 t_R2 := t_R1.f2.g2;
  // provoque une erreur car omit est affecté à t_R1.f2
t_R1. f2 := *
t_R2 := t_R1.f2.g2;
  // provoque une erreur car * est affecté à t_R1.f2

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

t_R2 := t_R1.f2;
t_R2 := t_R1.f2.g2;
t_R2 := t_R1.f2.g2.g2;
  // toutes ces affectations provoquent une erreur car une liste de
  // valeurs est affectée à t_R1

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

t_R2 := t_R1.f2;
t_R2 := t_R1.f2.g2;
t_R2 := t_R1.f2.g2.g2;
  // toutes ces affectations provoquent une erreur car une liste de
  // valeurs complémentées est affectée à t_R1
```

- AnyValue: en cas de référence à un sous-champ contenu dans un champ structuré auquel AnyValue (?) est affecté, dans la partie droite d'une affectation, AnyValue (?) doit être retourné pour les sous-champs obligatoires et AnyValueOrNone doit être retourné pour les sous-champs facultatifs.

  En cas de référence à un sous-champ contenu dans un champ structuré auquel AnyValue (?) est affecté, dans la partie gauche d'une affectation, le champ structuré doit être développé récursivement jusqu'à la profondeur du sous-champ référencé. Durant ce développement, AnyValue (?) doit être affecté aux sous-champs obligatoires et AnyValueOrNone doit être affecté aux sous-champs facultatifs. Après ce développement, la valeur ou le mécanisme d'appariement situé dans la partie droite de l'affectation doit être affecté au sous-champ référencé.

EXEMPLE 2:

```
t_R1 := {f1:=0, f2:=?}
t_R2 := t_R1.f2.g2;
  // après l'affectation t_R2 deviendra:
  // {g1:=?,g2:={g1:=?,g2:=*}}
t_R1.f2.g2.g2 := ({g1:=1, g2:=omit},{g1:=2, g2:=omit});
```

```
// en premier lieu le champ t_R1.f2 a été hypothétiquement développé à
// {g1:=?,g2:={g1:=?,g2:=*}}
// en conséquence, après l'affectation t_R1 deviendra:
// {f1:=0, f2:={g1:=?,g2:={g1:=?,g2:=({g1:=1, g2:=omit},{g1:=2, g2:=omit})}}}
```

- Attribut ifpresent: la référence à un sous-champ contenu dans un champ structuré auquel l'attribut **`ifpresent`** est associé doit provoquer une erreur (quelle que soit la valeur ou le mécanisme d'appariement auquel **`ifpresent`** est ajouté).

14.3.1.3 Référence à des éléments `record of` et `set of`

Les modèles et les variables de modèle permettent la référence à des éléments d'un modèle ou d'un champ **`record of`** ou **`set of`** au moyen de la notation indexée. Cependant, un mécanisme d'appariement peut être affecté au modèle ou au champ dans lequel l'élément est référencé. Le présent paragraphe indique les règles applicables au traitement de ces cas.

- Omit, AnyValueOrNone, listes de valeurs, listes de valeurs complémentées, sous-ensemble et superensemble: la référence à un élément contenu dans un champ record of ou set of auquel **`omit`**, AnyValueOrNone (*) avec ou sans attribut de longueur, une liste de valeurs, une liste de valeurs complémentées, un sous-ensemble ou un superensemble est affecté, doit provoquer une erreur.

EXEMPLE 1:

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

:
var template RoI t_RoI;
var template RoRoI t_RoRoI;
var template integer t_Int;
:
t_RoRoI := ({},{0},{0,0},{0,0,0});
t_RoI := t_RoRoI[0];
  // doit provoquer une erreur car une liste de valeurs est affectée à t_RoRoI;
```

- AnyValue: en cas de référence à un élément d'un modèle ou d'un champ **`record of`** ou **`set of`** auquel AnyValue (?) est affecté (sans attribut de longueur), dans la partie droite d'une affectation, AnyValue (?) doit être retournée. Si un attribut de longueur est associé à AnyValue (?), l'indice de la référence ne doit pas violer l'attribut de longueur.

  En cas de référence à un élément contenu dans un modèle ou un champ **`record of`** ou **`set of`** auquel AnyValue (?) est affecté (sans attribut de longueur), dans la partie gauche d'une affectation, la valeur ou le mécanisme d'appariement situé dans la partie droite de l'affectation doit être affecté à l'élément référencé, AnyElement(?) doit être affecté à tous les éléments précédant l'élément référence (s'il y en a un) et un seul AnyElementsOrNone(*) doit être ajouté à la fin. Lorsqu'un attribut de longueur est associé à AnyValue(?), cet attribut doit être transmis au nouveau modèle ou champ de manière transparente. L'indice ne doit violer les restrictions de type dans aucun des cas susmentionnés.

EXEMPLE 2:

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

:
var template RoI t_RoI;
var template RoRoI t_RoRoI;
var template integer t_Int;
:
t_RoI := ?;
t_Int := t_RoI[5];
  // après l'affectation t_Int prendra la valeur AnyValue(?);

t_RoRoI := ?;
t_RoI := t_RoRoI[5];
  // après l'affectation t_RoI prendra la valeur AnyValue(?);
t_Int := t_RoRoI[5].[3];
  // après l'affectation t_Int prendra la valeur AnyValue(?);

t_RoI := ? length (2..5);
t_Int := t_RoI[3];
  // après l'affectation t_Int prendra la valeur AnyValue(?);
t_Int := t_RoI[5];
  // doit provoquer une erreur car l'indice référencé est extérieur à
  // l'attribut de longueur
```

```
  // (il est à noter que l'indice 5 se référerait au 6e élément);

t_RoRoI[2] := {0,0};
  // après l'affectation t_RoRoI prendra la valeur {?,?,{0,0},*};
t_RoRoI[4] := {1,1};
  // après l'affectation t_RoRoI prendra la valeur {?,?,{0,0},?,{1,1},*};
t_RoI[0] := -5;
  // après l'affectation t_RoI prendra la valeur {-5,*}length(2..5);
t_RoI := ? length (2..5);
t_RoI[1] := 1;
  // après l'affectation t_RoI prendra la valeur {?,1,*}length(2..5);
t_RoI[3] := ?
  // après l'affectation t_RoI prendra la valeur {?,1,?,?,*}length(2..5);
t_RoI[5] := 5
  // après l'affectation t_RoI prendra la valeur {?,1,?,?,?,5,*}length(2..5);
  // il est à noter que t_RoI devient un ensemble vide mais qu'il ne
  // doit provoquer aucune erreur;
```

- Permutation: la référence à un élément d'un modèle ou d'un champ **record of**, qui est situé à l'intérieur d'une permutation (d'après son indice), doit provoquer une erreur. Les indices des éléments hébergés par une permutation doivent être déterminés d'après le nombre d'éléments de la permutation. L'inclusion de la structure AnyValueOrNone comme élément de permutation fait que la permutation héberge tous les indices des éléments **record of.**

EXEMPLE 3:

```
t_RoI := {permutation(0,1,3,?),2,?}
t_Int := t_RoI[5];
  // après l'affectation t_Int prendra la valeur AnyValue(?)

t_RoI := {permutation(0,1,3,?),2,*}
t_Int := t_RoI[5];
  // après l'affectation t_Int correspondra au symbole * (AnyValueOrNone)
t_Int := t_RoI[2];
  // provoque une erreur car le troisième élément (avec l'indice 2) est
  // à l'intérieur de la permutation

t_RoI := {permutation(0,1,3,*),2,?}
t_Int := t_RoI[5];
  // provoque une erreur car la permutation contient AnyValueOrNone(*)
  // qui peut s'appliquer à tout indice record of
```

- Attribut ifpresent: la référence à un élément contenu dans le champ **record of** ou **set of** auquel l'attribut **ifpresent** est associé doit provoquer une erreur (quels que soient la valeur ou le mécanisme d'appariement auxquels **ifpresent** est ajouté).

14.4 Paramétrage de modèles

14.4.0 Généralités

Les modèles relatifs aux opérations d'émission comme de réception peuvent être paramétrés. Les paramètres effectifs d'un modèle peuvent comprendre des valeurs et des modèles, des fonctions et les symboles spéciaux correspondants. Les règles applicables aux listes de paramètres formels et de paramètres effectifs doivent être suivies comme défini dans le § 5.2.

EXEMPLE:

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

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

14.5 Vide

14.6 Modèles modifiés

14.6.0 Généralités

Normalement, un modèle spécifie un ensemble de valeurs de base ou de valeurs par défaut, ou un ensemble de symboles correspondants pour chacun des champs définis dans la définition de type ou de signature appropriée. Si de légères modifications sont requises afin de spécifier un nouveau modèle, il est possible de spécifier un modèle modifié. Un modèle modifié spécifie des modifications relatives à des champs particuliers du modèle original, soit directement ou indirectement.

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

Les modifications apparaissent de façon liée avec retour final au modèle original. Si un champ de modèle est spécifié avec sa valeur correspondante ou son symbole d'appariement correspondant dans le modèle modifié, alors la valeur spécifiée où le symbole d'appariement spécifié remplace celui qui a été spécifié dans le modèle parent. Si un champ de modèle avec sa valeur correspondante ou son symbole d'appariement correspondant n'est pas spécifié dans le modèle modifié, alors la valeur ou le symbole d'appariement se trouvant dans le modèle parent doit être utilisé. Quand le champ à modifier est imbriqué dans un champ de modèle qui est déjà un champ structuré, aucun autre champ de ce champ structuré n'est modifié en dehors de celui (ceux) qui est (sont) explicitement indiqué(s).

Un modèle modifié ne doit pas se rapporter à lui-même, soit directement ou indirectement; c'est-à-dire que les dérivations récursives ne sont pas autorisées.

EXEMPLE 1:

```
// Si l'on a
type record MyRecordType
{
  integer field,
  charstring field2,
  boolean field3
}
template MyRecordType MyTemplate1 :=
{
  field1 := 123,
  field2 := "A string",
  field3 := true
}
// alors le fait d'écrire
template MyRecordType MyTemplate2 modifies MyTemplate1 :=
{
  field1 := omit,            // le champ field1 est facultatif mais présent
                             // dans MyTemplate1
  field2 := "A modified string"
                           // le champ field3 est inchangé
}
// revient à écrire
template MyRecordType MyTemplate2 :=
{
  field1 := omit,
  field2 := "A modified string",
  field3 := true
}
```

Dans le cas où l'on souhaite modifier certaines valeurs d'un modèle modifié ou d'un champ de modèle modifié de type **`record of`**, et uniquement dans ces cas, on peut aussi utiliser la notation par affectation de valeurs lorsque l'indice de l'élément à modifier est situé à gauche de l'affectation.

EXEMPLE 2:

```
template MyRecordOfType MyBaseTemplate := { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
template MyRecordOfType MyModifTemplate modifies MyBaseTemplate := { [2] := 3, [3] := 2 };
// MyModifTemplate doit correspondre à la séquence de valeurs { 0, 1, 3, 2, 4, 5, 6, 7, 8, 9 }
```

14.6.1 Paramétrage de modèles modifiés

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

a) le modèle dérivé ne doit pas omettre de paramètres définis pendant l'une quelconque des étapes de modification entre le modèle de base et le modèle effectivement modifié;

b) un modèle dérivé peut, au besoin, avoir des paramètres supplémentaires (ajoutés);

c) la liste des paramètres formels doit suivre le nom de chaque modèle modifié.

EXEMPLE:

```
// si l'on a
template MyRecordType MyTemplate1(integer MyPar):=
{
   field1 := MyPar,
   field2 := "A string",
   field3 := true
}

// alors une modification pourrait être
   template MyRecordType MyTemplate2(integer MyPar) modifies MyTemplate1 :=
{ // le champ field1 est paramétré dans Template1 et reste également
  // paramétré dans Template2
   field2 := "A modified string",
}
```

14.6.2 Modèles en ligne modifiés

En plus de la création de modèles modifiés explicitement nommés, la notation TTCN-3 permet la définition de modèles en ligne modifiés.

EXEMPLE:

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

// pourrait servir à définir un modèle en ligne modifié d'opération Setup
pco1.send (modifies Setup := {field1:= 76});
```

14.7 Modification des champs de modèle

Dans les opérations de communication (par exemple **send**, **receive**, **call**, **getcall**, etc.) il n'est permis de modifier des champs de modèle que par paramétrage ou par modèles dérivés en ligne. Les effets de ces modifications sur la valeur du champ de modèle ne persistent pas dans le modèle qui fait suite à l'événement de communication correspondant.

La notation à points *MyTemplateId.FieldId* ne doit pas servir à insérer ou à extraire des valeurs contenues dans des modèles lors d'événements de communication. Le symbole "->" doit être utilisé à cette fin (voir § 23).

14.8 Opération d'appariement

L'opération **match** permet de comparer la valeur d'une variable ou d'un paramètre à un modèle. Cette opération retourne une valeur booléenne. Si le type du modèle et celui de la variable ne sont pas compatibles (voir § 6.7) l'opération retourne la valeur "false". Si les types sont compatibles, la valeur de retour de l'opération indique si la valeur de la variable est conforme au modèle spécifié.

EXEMPLE:

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

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

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

```
// Vrai si la valeur effective de Rxvalue est inférieure à 10 et Faux sinon
   :
}
```

14.9 Opération de valuation

L'opération de type **valueof** permet d'affecter la valeur spécifiée dans un modèle à une variable. La variable et le modèle doivent être de types compatibles (voir § 6.7) et chaque champ du modèle doit correspondre à une valeur isolée.

EXEMPLE:

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

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

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

15 Opérateurs

15.0 Généralités

La notation TTCN-3 prend en charge un certain nombre d'opérateurs prédéfinis qui peuvent être utilisés dans les termes des expressions TTCN-3. Les opérateurs prédéfinis s'inscrivent dans sept catégories:

a) opérateurs arithmétiques;

b) opérateurs de concaténation;

c) opérateurs relationnels;

d) opérateurs logiques;

e) opérateurs binaires;

f) opérateurs de décalage;

g) opérateurs de rotation.

Ces opérateurs sont énumérés dans le Tableau 7.

Tableau 7/Z.140 – Liste des opérateurs TTCN-3

Catégorie	Opérateur	Symbole ou mot clé
Opérateurs arithmétiques	addition	**+**
	soustraction	**-**
	multiplication	*****
	division	**/**
	modulo	**mod**
	reste	**rem**
Opérateurs de concaténation	concaténation	**&**
Opérateurs relationnels	égal à	**==**
	inférieur à	**<**
	supérieur à	**>**
	inégal à	**!=**
	supérieur ou égal à	**>=**
	inférieur ou égal à	**<=**

Tableau 7/Z.140 – Liste des opérateurs TTCN-3

Catégorie	Opérateur	Symbole ou mot clé
Opérateurs logiques	non logique	**not**
	et logique	**and**
	ou logique	**or**
	ou exclusif logique	**xor**
Opérateurs binaires	non binaire	**not4b**
	et binaire	**and4b**
	ou binaire	**or4b**
	ou exclusif binaire	**xor4b**
Opérateurs de décalage	décalage à gauche	<<
	décalage à droite	>>
Opérateurs de rotation	rotation à gauche	<@
	rotation à droite	@>

La priorité de ces opérateurs est représentée dans le Tableau 8. A l'intérieur de chaque rangée de ce tableau, les opérateurs énumérés ont une priorité égale. Si plusieurs opérateurs de priorité égale apparaissent dans une expression, les opérations sont évaluées de gauche à droite. Les parenthèses peuvent servir à grouper des opérandes dans des expressions, auquel cas une expression entre parenthèses a la priorité la plus élevée lors de l'évaluation.

Tableau 8/Z.140 – Priorité des opérateurs

Priorité	Type d'opérateur	Opérateur
La plus élevée		(…)
	Unaire	+, -
	Binaire	*, /, mod, rem
	Binaire	+, -, &
	Unaire	**not4b**
	Binaire	**and4b**
	Binaire	**xor4b**
	Binaire	**or4b**
	Binaire	<<, >>, <@, @>
	Binaire	<, >, <=, >=
	Binaire	==, !=
	Unaire	**not**
	Binaire	**and**
	Binaire	**xor**
La moins élevée	Binaire	**or**

15.1 Opérateurs arithmétiques

Les opérateurs arithmétiques représentent les opérations d'addition, de soustraction, de multiplication, de division, de modulo et de reste. Les opérandes de ces opérateurs doivent être de type **integer** (y compris les dérivés de ce type) ou **float** (y compris les dérivés de ce type) sauf pour **mod** et **rem** qui ne doivent être utilisés qu'avec le type **integer** (y compris les dérivés de ce type).

Avec les types **integer** le type résultant des opérations arithmétiques est **integer**. Avec les types float, le type résultant des opérations arithmétiques est **float**.

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

L'exécution de l'opération de division (/) sur:

a) deux valeurs de type **integer** donne comme résultat la partie totale de type **integer** de la valeur résultant de la division du premier nombre de type **integer** par le second (c'est-à-dire que les valeurs fractionnaires sont rejetées);

b) deux valeurs de type **float** donne comme résultat la valeur de type **float** résultant de la division du premier nombre de type **float** par le second (c'est-à-dire que les valeurs fractionnaires ne sont pas rejetées).

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

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

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

Le Tableau 9 décrit la différence entre les opérateurs **mod** et **rem.**

Tableau 9/Z.140 – Effet des opérateurs mod et rem

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

15.2 Opérateurs de concaténation

Les opérateurs de concaténation prédéfinis effectuent la concaténation de valeurs de types concaténés compatibles. L'opération est une simple concaténation de gauche à droite. Aucune forme d'addition arithmétique n'est impliquée. Le type résultant est le type radical de l'opérande.

EXEMPLE:

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

15.3 Opérateurs relationnels

Les opérateurs relationnels prédéfinis représentent les relations d'égalité (==), d'infériorité (<), de supériorité (>), d'inégalité (!=), de supériorité ou égalité (>=) et d'infériorité ou égalité (<=). Les opérandes d'égalité et d'inégalité peuvent être de types arbitraires mais compatibles, à l'exception du type **enumerated**, auquel cas les opérandes doivent être des instances du même type. Tous les autres opérateurs relationnels ne doivent avoir que des opérandes du type **integer** (y compris les types dérivés de celui-ci), du type **float** (y compris les dérivés de ce type) ou des instances des mêmes types **enumerated**. Le type résultant de ces opérations est: **boolean**.

Deux valeurs de type **charstring** ou **universal charstring** ne sont égales que si elles ont une longueur égale et si les caractères sont les mêmes à toutes les positions. Pour les valeurs des types **bitstring, hexstring** ou **octetstring**, la même règle d'égalité s'applique, sauf que les fractions qui doivent être égales à toutes les positions sont respectivement: des bits, des chiffres hexadécimaux ou des paires de chiffres hexadécimaux.

Deux valeurs de type **record**, **set**, **record of** ou **set of** ne sont égales que si et seulement si leurs structures de valeur effective sont compatibles (voir § 6.7) et si les valeurs de tous les champs correspondants sont égales. Les valeurs de type **record** peuvent également être comparées aux valeurs de type **record of** et les valeurs de type **set** peuvent être comparées aux valeurs de type **set of**. Dans ces cas, la règle qui s'applique est la même que pour comparer deux valeurs de type **record** ou **set**.

NOTE – L'expression "tous les champs" signifie que les champs facultatifs non présents dans la valeur effective d'un type **record** doivent être considérés comme étant une valeur indéfinie. Un tel champ ne peut être égal qu'à un champ facultatif manquant (également considéré comme étant une valeur indéfinie) quand il est comparé à la valeur d'un autre type **record** ou ne peut être égal qu'à un élément de valeur indéfinie quand il est comparé à une valeur de type **record of**. Ce principe s'applique également à la comparaison des valeurs de deux types **set** ou à celle d'un type **set** et d'un type **set of**.

Deux valeurs de type **union** sont égales si et seulement si, dans ces deux valeurs, les types des champs choisis sont compatibles et si les valeurs effectives des champs choisis sont égales.

EXEMPLE:

```
// si l'on a
type  set  SetA{
      integer    a1   optional,
      integer    a2   optional,
      integer    a3   optional
      };

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

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

type  set of integer   SetOf;

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

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

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

// et
const      Set A  conSetA1  :={ a1 := 0, a2 := omit, a3 := 2 };
      // Noter que l'ordre de définition des valeurs des champs n'a
      // pas d'importance
const      SetB   conSetB1  :={ b1 := 0, b3 := 2, b2 := omit };
const      SetB   conSetB2  :={ b2 := 0, b3 := 2, b1 := omit };
const      SetC   conSetC1  :={ c1 := 0, c2 :=2 };
const      SetOf  conSetOf1 :={ 0, omit, 2 };
const      SetOf  conSetOf2 :={ 0, 2 };
const      UniD   conUniD1  :={ d1:= 0 };
const      UniE   conUniE1  :={ e1:= 0 };
const      UniE   conUniE2; :={ e2:= 0 };
const      UniF   conUniF1; :={ f1:= 0 };

// Alors
conSetA1 == conSetB1;
  // retourne la valeur true
conSetA1 == conSetB2;
  // retourne la valeur "false" parce que ni le champ a1 ni le champ a2
  // n'est égal à sa contrepartie
  // (l'élément correspondant n'est pas omis)
conSetA1 == conSetC1;
  // retourne la valeur "false", parce que les structures de valeur
  // effective de SetA et SetC ne sont pas compatibles
conSetA1 == conSetOf1;
  // retourne la valeur true
conSetA1 == conSetOf2;
  // retourne la valeur "false", car la contrepartie du champ omis a2
  // est 2, mais la contrepartie de a3 est indéfinie
conSetC1 == conSetOf2;
  // retourne la valeur true
conUniD1 == conUniE1;
  // retourne la valeur true
conUniD1 == conUniE2;
  // retourne la valeur "false", car le champ choisi e2 n'est pas la
  // contrepartie du champ d1 de UniD1
```

```
conUniD1 == conUniF1;
  // retourne la valeur "false", car les structures de valeur effective de
  // UniD1 et UniF ne sont pas compatibles
```

15.4 Opérateurs logiques

Les opérateurs prédéfinis de type **boolean** effectuent les opérations de **non** logique, de **et** logique, de **ou** logique et de **oux** (ou exclusif) logique. Leurs opérandes doivent être de type **boolean**. Le type résultant des opérations logiques est: **boolean**.

Le **non** logique est l'opérateur unaire qui retourne la valeur **true** si son opérande était de valeur **false** et qui retourne la valeur **false** si l'opérande était de valeur **true**.

Le **et** logique retourne la valeur **true** si la valeur de ses deux opérandes sont **true**; sinon, il retourne la valeur **false**.

Le **ou** logique retourne la valeur **true** si au moins un de ses opérandes a la valeur **true**; il ne retourne la valeur **false** que si ses deux opérandes ont la valeur **false**.

Le **oux** logique retourne la valeur **true** si un seul de ses opérandes a la valeur **true**; il retourne la valeur **false** si ses deux opérandes ont la valeur **false** ou si ses deux opérandes ont la valeur **true**.

L'évaluation court-circuit est appliquée aux expressions booléennes, c'est-à-dire que l'évaluation des opérandes des opérateurs logiques prend fin une fois que le résultat est connu: dans le cas de l'opérateur et (**and**), si l'argument de gauche s'évalue comme ayant la valeur **false**, alors l'argument de droite n'est pas évalué et l'expression tout entière s'évalue comme ayant la valeur **false**. Dans le cas de l'opérateur ou (**or**), si l'argument de gauche s'évalue comme ayant la valeur **true**, alors l'argument de droite n'est pas évalué et l'expression tout entière s'évalue comme ayant la valeur **true**.

15.5 Opérateurs binaires

Les opérateurs binaires prédéfinis effectuent les opérations de non binaire, de et binaire, de ou binaire et de oux binaire. Ces opérateurs sont respectivement désignés par: **not4b**, **and4b**, **or4b** et **xor4b**.

NOTE – Ces termes signifient "not for bit" (non en mode binaire)", "and for bit" (et en mode binaire), etc.

Leurs opérandes doivent être de type **bitstring, hexstring,** ou **octetstring**. Dans le cas des opérateurs **and4b, or4b et xor4b**, les opérandes doivent être de types compatibles. Le type résultant des opérateurs binaires doit être le type radical de l'opérande.

L'opérateur unaire **not4b** inverse les valeurs binaires individuelles de son opérande. Chaque bit 1 de l'opérande est réglé à 0 et chaque bit 0 est réglé à 1. Autrement dit:

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

EXEMPLE 1:

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

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

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

EXEMPLE 2:

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

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

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

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

EXEMPLE 3:

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

L'opérateur binaire **xor4b** accepte deux opérandes d'égale longueur. Aux positions binaires correspondantes, la valeur résultante est 0 si les deux bits sont réglés l'un et l'autre à 0 ou à 1; sinon, la valeur du bit résultant est 1. Autrement dit:

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

EXEMPLE 4:

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

15.6 Opérateurs de décalage

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

Les opérateurs de décalage ont un comportement différent selon le type de leur opérande de gauche. Si le type de l'opérande de gauche est:

a) **bitstring**, alors l'unité de décalage appliquée est 1 bit;

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

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

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

EXEMPLE 1:

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

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

EXEMPLE 2:

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

15.7 Opérateurs de rotation

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

Les opérateurs de rotation ont un comportement différent selon le type de leur opérande de gauche. Si le type de l'opérande de gauche est:

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

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

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

d) **charstring** ou **universal charstring**, alors l'unité de rotation appliquée est un seul caractère.

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

EXEMPLE 1:

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

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

EXEMPLE 2:

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

16 Fonctions et variantes

En notation TTCN-3, les fonctions et variantes servent à spécifier et à structurer un comportement de test, à définir un comportement par défaut et à structurer des calculs dans un module, etc., comme décrit dans les paragraphes suivants.

16.1 Fonctions

16.1.0 Généralités

Les fonctions sont utilisées en notation TTCN-3 afin d'exprimer un comportement de test, d'organiser l'exécution d'un test ou de structurer des calculs dans un module, par exemple afin de calculer une valeur isolée, à initialiser un ensemble de variables ou à vérifier certaines conditions. Les fonctions peuvent retourner une valeur ou un modèle. Le retour d'une valeur par le mot clé **return** suivi par un identificateur de type. Le retour d'un modèle est indiqué par les mots clés **return template** suivis par un identificateur de type.

Le mot clé **return**, quand il est utilisé dans le corps de la fonction retour d'une valeur définie dans son en-tête, doit toujours être suivi par une expression représentant la valeur de retour. Le type de la valeur de retour doit être compatible avec le type de retour. Le mot clé **return**, quand il est utilisé dans le corps de la fonction avec retour d'un modèle défini dans son en-tête, doit toujours être suivi par une expression ou une instance de modèle représentant le modèle de retour. Le type du modèle de retour doit être compatible avec le type de modèle de retour.

L'instruction de retour contenue dans le corps de la fonction incite celle-ci à terminer et à retourner la valeur de retour à l'emplacement de l'appel de fonction.

EXEMPLE 1:

```
// Définition de MyFunction qui n'a aucun paramètre
function MyFunction() return integer
{

  return 7;    // retourne la valeur d'entier 7 quand la fonction
               // s'achève

}

// Définition des fonctions qui peuvent retourner des symboles
// d'appariement ou des modèles function MyFunction2() return template
function MyFunction2() return template integer
{
:
  return ?;    // retourne le mécanisme d'appariement AnyValue
}
```

```
function MyFunction3() return template octetstring
{
:
  return "FF??FF"O;    // retourne une chaîne d'octets comportant AnyElement
}
```

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

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

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

Dans un module, le comportement d'une fonction peut être défini au moyen des instructions de programmation et des opérations décrites dans le § 18. Si une fonction utilise des variables, des constantes, des temporisations et des ports qui sont déclarés dans une définition de type de composant, le type de composant doit être désigné au moyen du mot clé **runs on** dans l'en-tête de la fonction. La seule exception à cette règle est le cas où toutes les informations nécessaires au sujet du composant sont transmises dans la fonction sous forme de paramètres.

EXEMPLE 2:

```
function MyFunction3() runs on MyPTCType {
                                   // MyFunction3 ne retourne pas de valeur
                                   // mais fait bien usage de l'opération
  var integer MyVar := 5;          // de port d'envoi. Cette fonction
  PCO1.send(MyVar);                // exige donc une clause "runs on"
                                   // afin de résoudre les identificateurs
                                   // de port au moyen d'une
}                                  // référence à un type de composant
```

Une fonction sans clause **runs on** ne doit jamais invoquer une fonction ou une variante, ni activer localement une variante par défaut avec une clause **runs on**.

Les fonctions lancées au moyen de l'opération de lancement **start** de composant de test doivent toujours avoir une clause **runs on** (voir § 22.5) et sont considérées comme étant invoquées dans le composant à lancer, c'est-à-dire non localement. Cependant, l'opération de lancement **start** d'un composant de test peut être invoquée dans des fonctions sans clause **runs on**.

NOTE – Les restrictions concernant la clause **runs on** ne sont associées qu'aux fonctions et aux variantes et non aux tests élémentaires.

Les fonctions utilisées dans la partie commande d'un module TTCN-3 ne doivent avoir aucune clause **runs on**. Néanmoins, elles sont autorisées à exécuter des tests élémentaires.

16.1.1 Paramétrage de fonctions

Les fonctions peuvent être paramétrées. Les règles applicables aux listes de paramètres formels doivent être suivies comme défini dans le § 5.2.

EXEMPLE:

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

16.1.2 Invocation des fonctions

Une fonction est invoquée par référence à son nom et par fourniture de la liste de paramètres effectifs. Les fonctions qui ne retournent pas de valeurs doivent être invoquées directement. Les fonctions qui retournent des valeurs peuvent être invoquées directement ou à l'intérieur d'expressions. Les règles applicables aux listes de paramètres effectifs doivent être suivies comme défini dans le § 5.2.

EXEMPLE:

```
MyVar := MyFunction4();       // La valeur retournée par MyFunction4 est affectée
                              // à MyVar. Les types de la valeur retournée et de
                              // MyVar doivent être compatibles
```

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

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

Des restrictions particulières s'appliquent aux fonctions associées à des composants de test au moyen de l'opération de lancement de composant de test **`start`**. Ces restrictions sont décrites dans le § 22.5.

16.1.3 Fonctions prédéfinies

La notation TTCN-3 contient un certain nombre de fonctions prédéfinies (intégrées) qui n'ont pas besoin d'être déclarées avant usage.

Tableau 10/Z.140 – Liste des fonctions prédéfinies en notation TTCN-3

Catégorie	Fonction	Mot clé
Fonctions de conversion	Convertit une valeur **integer** en valeur de type **charstring**	`int2char`
	Convertit une valeur **integer** en valeur **universalcharstring**	`int2unichar`
	Convertit une valeur **integer** en valeur **bitstring**	`int2bit`
	Convertit une valeur **integer** en valeur **hexstring**	`int2hex`
	Convertit une valeur **integer** en valeur **octetstring**	`int2oct`
	Convertit une valeur **integer** en valeur **charstring**	`int2str`
	Convertit une valeur **integer** en valeur **float**	`int2float`
	Convertit une valeur **float** en valeur **integer**	`float2int`
	Convertit une valeur **charstring** en valeur **integer**	`char2int`
	Convertit une valeur **charstring** en valeur **octetstring**	`char2oct`
	Convertit une valeur universal **charstring** en valeur **integer**	`unichar2int`
	Convertit une valeur **bitstring** en valeur **integer**	`bit2int`
	Convertit une valeur **bitstring** en valeur **hexstring**	`bit2hex`
	Convertit une valeur **bitstring** en valeur **octetstring**	`bit2oct`
	Convertit une valeur **bitstring** en valeur **charstring**	`bit2str`
	Convertit une valeur **hexstring** en valeur **integer**	`hex2int`
	Convertit une valeur **hexstring** en valeur **bitstring**	`hex2bit`
	Convertit une valeur **hexstring** en valeur **octetstring**	`hex2oct`
	Convertit une valeur **hexstring** en valeur **charstring**	`hex2str`
	Convertit une valeur **octetstring** en valeur **integer**	`oct2int`
	Convertit une valeur **octetstring** en valeur **bitstring**	`oct2bit`
	Convertit une valeur **octetstring** en valeur **hexstring**	`oct2hex`
	Convertit une valeur **octetstring** en valeur **charstring**	`oct2str`
	Convertit une valeur **octetstring** en valeur **charstring**	`oct2char`
	Convertit une valeur **charstring** en valeur **integer**	`str2int`
	Convertit une valeur **charstring** en valeur **octetstring**	`str2oct`
	Convertit une valeur **charstring** en valeur **float**	`str2float`
Fonctions de longueur/taille	Retourne la longueur d'une valeur de tout type chaîne	`lengthof`
	Retourne le nombre d'éléments contenus dans un type **record**, **record of**, **template**, **set**, **set of** ou **array**	`sizeof`
	Retourne le nombre d'éléments contenus dans un type structuré	`sizeoftype`

Tableau 10/Z.140 – Liste des fonctions prédéfinies en notation TTCN-3

Catégorie	Fonction	Mot clé
Fonctions de présence/choix	Détermine si un champ facultatif est présent dans un type **record**, **record of**, **template**, **set** ou **set of**	`ispresent`
	Détermine quel choix a été effectué dans un type **union**	`ischosen`
Fonctions de traitement de chaîne	Retourne une partie de la chaîne d'entrée correspondant à la description de structure spécifiée	`regexp`
	Retourne la portion spécifiée de la chaîne d'entrée	`substr`
	Remplace une sous-chaîne d'une chaîne par la chaîne d'entrée ou insère celle-ci dans une chaîne	`replace`
Autres fonctions	Produit un nombre aléatoire à virgule flottante	`rnd`

Quand une fonction prédéfinie est invoquée:

1) le nombre des paramètres effectifs doit être le même que le nombre des paramètres formels;
2) chaque paramètre effectif doit prendre la valeur d'un élément de son type correspondant de paramètre formel;
3) toutes les variables apparaissant dans la liste des paramètres effectifs doivent être associées.

La description complète des fonctions prédéfinies est reproduite dans l'Annexe C.

16.1.4 Restrictions applicables aux fonctions appelées à partir d'emplacements spécifiques

Des fonctions retournant des valeurs peuvent être appelées durant des opérations de communication (dans des modèles, des champs de modèle ou des modèles en ligne) ou durant l'évaluation d'instantanés (dans des sentinels Booléens d'instructions **altstep** ou dans des variantes (voir § 20.1.1) et lors de l'initialisation de définitions locales dans des variantes (voir § 16.2.2)). Pour éviter des effets secondaires qui modifient l'état du composant ou de l'instantané effectif et pour empêcher que les évaluations ultérieures sur un instantané inchangé ne donnent des résultats différents, les opérations suivantes ne doivent pas être utilisées dans les fonctions appelées dans les cas spécifiés ci-dessus:

- toutes les opérations sur composant, c'est-à-dire les opérations **create**, **start** (lancement de composant), **stop** (arrêt de composant), **kill**, **running** (exécution de composant), **alive**, **done** (fin d'exécution) et **killed** (voir les Notes 1, 3, 4 et 6);
- toutes les opérations sur les ports, c'est-à-dire les opérations **start** (lancement de port), **stop** (arrêt de port), **halt**, **clear**, **send**, **receive**, **trigger**, **call**, **getcall**, **reply**, **getreply**, **raise**, **catch**, **check**, **connect**, **map** (voir les Notes 1, 2, 3 et 6);
- l'opération **action** (voir les Notes 2 et 6);
- toutes les opérations de temporisation, c'est-à-dire les opérations **start** (déclenchement de temporisateur), **stop** (arrêt de temporisateur), **running** (temporisateur en phase d'exécution), **read**, **timeout** (voir les Notes 4 et 6);
- appel de fonctions externes (voir les Notes 4 et 6);
- appel de la fonction prédéfinie **rnd** (voir les Notes 4 et 6);
- modification de variables de composant, c'est-à-dire utilisation de variables de composant dans la partie droite d'affectations, et lors de l'instanciation des paramètres **out** et **inout** (voir les Notes 4 et 6);
- appel de l'opération **setverdict** (voir les Notes 4 et 6);
- activation et désactivation de valeurs par défaut, c'est-à-dire les instructions **activate** et **deactivate** (voir les Notes 5 et 6);
- appel de fonctions avec les paramètres **out** ou **inout** (voir les Notes 7 et 8).

NOTE 1 – L'exécution des opérations **start**, **stop**, **done**, **killed**, **halt**, **clear**, **receive**, **trigger**, **getcall**, **getreply**, **catch** et **check** peut entraîner des modifications de l'instantané actuel.

NOTE 2 – Les opérations **send**, **call**, **reply**, **raise** et **action** doivent être évitées pour des raisons de lisibilité, c'est-à-dire que toute communication doit être rendue explicite et non pas constituer un effet secondaire d'une autre opération de communication ou de l'évaluation d'un instantané.

NOTE 3 – Les opérations **map**, **unmap**, **connect**, **disconnect**, **create** doivent être évitées pour des raisons de lisibilité, c'est-à-dire que toutes les opérations de configuration doivent être rendues explicites et non pas constituer un effet secondaire d'une opération de communication ou de l'évaluation d'un instantané.

NOTE 4 – L'appel des fonctions externes **rnd**, **running**, **alive**, **read**, **setverdict**, et l'écriture à des variables de composant doivent être évités car cela pourrait avoir pour effet que des évaluations ultérieures du même instantané donnent des résultats différents, avec pour conséquence, par exemple, de rendre impossible la détection d'un blocage.

NOTE 5 – Les opérations **activate** et **deactivate** doivent être évitées car elles modifient l'ensemble des valeurs par défaut qui est considéré durant l'évaluation de l'instantané actuel.

NOTE 6 – Les restrictions à l'exception de la limitation de l'utilisation du paramétrage **out** ou **inout** doivent être appliquées récursivement, c'est-à-dire qu'il est interdit de les utiliser directement, ou via une longue chaîne arbitraire d'invocations de fonctions.

NOTE 7 – La restriction d'appel de fonctions avec paramètres **out** ou **inout** ne doit pas être appliquée récursivement, c'est-à-dire que l'appel de fonctions qui elles-mêmes appellent des fonctions avec paramètres **out** ou **inout** est légal.

NOTE 8 – L'utilisation des paramètres **out** ou **inout** doit être évitée car elle peut aussi avoir pour effet que des évaluations ultérieures du même instantané donnent des résultats différents.

16.2 Variantes

16.2.0 Généralités

La notation TTCN-3 utilise les variantes afin de spécifier un comportement par défaut ou afin de structurer les options d'une instruction **alt**. Les variantes sont des unités de portée similaires aux fonctions. Le corps de la variante définit un ensemble facultatif de définitions locales et un ensemble d'options, dites *options sommitales*, qui forment le corps de la variante. Les règles syntaxiques des options sommitales sont identiques aux règles syntaxiques des options des instructions de type **alt**.

Le comportement d'une variante peut être défini au moyen des instructions de programmation et au moyen des opérations résumées dans le § 18. Si une variante comprend des opérations d'accès ou utilise des variables de composant, des constantes ou des temporisations, le type de composant associé doit être désigné au moyen du mot clé **runs on** dans l'en-tête de la variante. La seule exception à cette règle est le cas où tous les accès, toutes les variables, toutes les constantes et toutes les temporisations utilisés dans la variante sont transmis sous forme de paramètres.

EXEMPLE:

```
// si l'on a
type component MyComponentType {
  var integer MyIntVar := 0;
  timer MyTimer;
  port MyPortTypeOne PCO1, PCO2;
  port MyPortTypeTwo PCO3;
}

// Définition de variante utilisant PCO1, PCO2, MyIntVar et MyTimer de
// MyComponentType

altstep AltSet_A(in integer MyPar1) runs on MyComponentType {
  [] PCO1.receive(MyTemplate(MyPar1, MyIntVar) {
       setverdict(inconc);
     }
  [] PCO2.receive {
          repeat
     }
  [] MyTimer.timeout {
       setverdict(fail);
          stop
     }
}
```

Les variantes peuvent invoquer des fonctions et des variantes ou activer des variantes par défaut. Une variante sans clause **runs on** ne doit jamais invoquer une fonction ou variante ni activer localement une variante par défaut avec clause **runs on**.

16.2.1 Paramétrage des variantes

Les variantes peuvent être paramétrées. Une variante qui est activée par défaut ne doit avoir que des paramètres de type **in**, des paramètres de port et des paramètres de temporisation. Une variante qui n'est invoquée que comme une option dans une instruction **alt** ou que comme une instruction autonome dans une description de comportement TTCN-3 peut avoir des paramètres **in**, **out** et **inout**. Les règles applicables aux listes de paramètres formels doivent être suivies comme défini dans le § 5.2.

16.2.2 Définitions locales dans des variantes

16.2.2.0 Généralités

Les variantes peuvent définir des définitions locales de constantes, de variables et de temporisations. Les définitions locales doivent être précisées avant l'ensemble des options.

EXEMPLE:

```
altstep AnotherAltStep(in integer MyPar1) runs on MyComponentType {
   var integer MyLocalVar := MyFunction();       // variable locale
   const float MyFloat := 3.41;                 // constante locale
   [] PCO1.receive(MyTemplate(MyPar1, MyLocalVar) {
        setverdict(inconc);
        }
   [] PCO2.receive {
        repeat
        }
}
```

16.2.2.1 Restrictions pour l'initialisation de définitions locales dans des variantes

L'initialisation de définitions locales par appel de fonctions retournant des valeurs peut avoir des effets secondaires. Afin d'éviter des effets secondaires qui provoquent une incohérence entre l'instantané actuel et l'état du composant, et pour éviter que des évaluations ultérieures d'un instantané inchangé donnent des résultats différents, les restrictions indiquées dans le § 16.1.4 doivent s'appliquer à l'initialisation de définitions locales.

16.2.3 Invocation de variantes

L'invocation d'une variante est toujours associée à une instruction **alt**. L'invocation peut être effectuée soit implicitement par le mécanisme de valeurs par défaut (voir § 21) ou explicitement par un appel direct dans une instruction **alt** (voir § 20.1.6). L'invocation d'une variante ne provoque aucun nouvel instantané et l'évaluation des options sommitales d'une variante est assurée au moyen de l'instantané effectif de l'instruction **alt** à partir de laquelle la variante a été appelée.

NOTE – Un nouvel instantané dans une variante va évidemment être pris si, dans une option sommitale sélectionnée, une nouvelle instruction **alt** est spécifiée et introduite.

Dans le cas d'une invocation implicite d'une variante au moyen du mécanisme de valeurs par défaut, cette variante doit être activée par défaut au moyen d'une instruction **activate** avant que l'emplacement de l'invocation soit atteint.

EXEMPLE 1:

```
  :
var default MyDefVarTwo := activate(MySecondAltStep()); // Activation d'une variante
                                                        // par défaut
  :
```

Un appel explicite de variante dans une instruction **alt** se présente comme une option d'appel de fonction.

EXEMPLE 2:

```
  :
alt {
   [] PCO3.receive {
        …
       }
   [] AnotherAltStep(); // appel explicite de la variante AnotherAltStep
                        // présenté comme une option d'une instruction "alt"
   [] MyTimer.timeout {}
}
```

Quand une variante est appelée explicitement dans une instruction **alt**, la prochaine option à vérifier est la première option de la variante **altstep**. Les options de la variante **altstep** sont vérifiées et exécutées de la même façon que les options d'une instruction **alt** (voir § 20.1) sauf qu'aucun nouvel instantané n'est pris lors de l'entrée dans la variante **altstep**. Un échec de terminaison de la variante (c'est-à-dire que toutes les options sommitales de la variante **altstep** ont été vérifiées et qu'aucune branche correspondante n'a été trouvée) provoque l'évaluation de la prochaine option ou l'invocation du mécanisme de valeurs par défaut (si l'appel explicite est la dernière option de l'instruction **alt**). Un succès de terminaison peut soit provoquer la terminaison du composant de test, c'est-à-dire que la variante **altstep** se termine par une instruction d'arrêt **stop**, ou provoquer un nouvel instantané et la réévaluation de l'instruction **alt**, c'est-à-dire que la variante **altstep** se termine par l'opération répéter (voir § 20.2) ou par une continuation immédiatement après l'instruction **alt**, c'est-à-dire que l'option sommitale choisie de la variante se termine sans répétition explicite **repeat** ou **stop**.

Une variante **altstep** peut également être appelée comme instruction autonome dans une description de comportement TTCN-3. Dans ce cas, l'appel de la variante **altstep** peut être interprété comme un abrégé d'instruction **alt** ayant seulement une seule option décrivant l'appel explicite de la variante **altstep**.

EXEMPLE 3:

```
// L'instruction
AnotherAltStep(); // AnotherAltStep est censée être une variante correctement définie

// est un abrégé pour

alt {
   [] AnotherAltStep();
}
```

16.3 Fonctions et variantes pour des types différents de composant

Voir § 6.7.3.

17 Tests élémentaires

17.0 Généralités

Les tests élémentaires sont une sorte spéciale de fonction. Dans la partie d'un module relative à la commande, l'instruction **execute** sert à lancer des tests élémentaires (voir § 27.1). Le résultat de l'exécution d'un test élémentaire est toujours une valeur de type **verdicttype**. Chaque test élémentaire doit contenir un composant MTC et un seul, du type qui est indiqué dans l'en-tête de la définition du test élémentaire. Le comportement défini dans le corps d'un test élémentaire est le comportement du composant MTC.

Quand un test élémentaire est invoqué, le composant MTC est créé, les ports du composant MTC et l'interface avec le système de test sont instanciés et le comportement spécifié dans les définitions du test élémentaire est lancé dans le composant MTC. Toutes ces actions doivent être effectuées implicitement, c'est-à-dire sans les opérations explicites **create** et **start.**

Afin de fournir les informations permettant que ces opérations implicites se produisent, un en-tête de test élémentaire a deux parties:

a) partie interface (obligatoire): indiquée par le mot clé **runs on** qui fait référence au type de composant requis pour le composant MTC et qui rend les noms des ports associés visibles dans le comportement du composant MTC;

b) partie système de test (facultative): indiquée par le mot clé **system** qui fait référence au type de composant définissant les ports requis par l'interface du système de test. La partie système de test ne doit être omise que si, pendant l'exécution du test, seul le composant MTC est instancié. Dans ce cas, le type de composant MTC définit implicitement les ports de l'interface du système de test.

EXEMPLE:

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

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

17.1 Paramétrage de tests élémentaires

Les tests élémentaires peuvent être paramétrés. Les règles applicables aux listes de paramètres formels doivent être suivies comme défini dans le § 5.2.

18 Aperçu général des instructions de programmation et des opérations

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

Les instructions peuvent être soit isolées (sans inclure d'autres instructions de programmation) ou composites (pouvant inclure d'autres instructions et des blocs d'instructions et de déclarations).

Les instructions doivent être exécutées dans l'ordre de leur apparition, c'est-à-dire séquentiellement, comme illustré dans la Figure 8.

Figure 8/Z.140 – Illustration du comportement séquentiel

Les instructions individuelles contenues dans la séquence doivent être séparées par le délimiteur ";".

EXEMPLE:

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

La spécification d'un bloc vide d'instructions et de déclarations, c'est-à-dire { }, peut être trouvée dans des instructions composites, par exemple une branche dans une instruction **`alt`**. Elle implique qu'aucune action n'est entreprise.

Tableau 11/Z.140 – Aperçu général des expressions, instructions et opérations TTCN-3

Instruction	Mot clé ou symbole associé	Instruction pouvant être utilisée dans la partie commande du module	Instruction pouvant être utilisée dans des fonctions, tests élémentaires et variantes	Instruction pouvant être utilisée dans des fonctions appelées à partir de modèles, sentinelles booléennes ou à partir de définitions locales de variante
Expressions	(…)	Oui	Oui	Oui
Instructions de programmation de base				
Affectations	**`:=`**	Oui	Oui	Oui (voir Note 3)
Journalisation	**`log`**	Oui	Oui	Oui
Etiqueter et Saut	**`label / goto`**	Oui	Oui	Oui
Condition "si-sinon"	**`if (…) {…} else {…}`**	Oui	Oui	Oui
Boucle "pour"	**`for (…) {…}`**	Oui	Oui	Oui
Boucle "tant que"	**`while (…) {…}`**	Oui	Oui	Oui
Boucle "exécuter tant que"	**`do {…} while (…)`**	Oui	Oui	Oui
Arrêt d'exécution	**`stop`**	Oui	Oui	
Sélection de test élémentaire	**`select case (…) { case (…) {…} case else {…}}`**	Oui	Oui	Oui
Instructions de programmation comportementales				
Comportement à options	**`alt {…}`**	Oui (voir Note 1)	Oui	
Réévaluation de comportement à options	**`repeat`**	Oui (voir Note 1)	Oui	

Tableau 11/Z.140 – Aperçu général des expressions, instructions et opérations TTCN-3

Instruction	Mot clé ou symbole associé	Instruction pouvant être utilisée dans la partie commande du module	Instruction pouvant être utilisée dans des fonctions, tests élémentaires et variantes	Instruction pouvant être utilisée dans des fonctions appelées à partir de modèles, sentinelles booléennes ou à partir de définitions locales de variante
Comportement entrelacé	`interleave {…}`	Oui (voir Note 1)	Oui	
Commande de retour	`return`		Oui (voir Note 4)	Oui
Instructions pour manipulation des valeurs par défaut				
Activer une valeur par défaut	`activate`	Oui (voir Note 1)	Oui	
Désactiver une valeur par défaut	`deactivate`	Oui (voir Note 1)	Oui	
Opérations de configuration				
Création d'un composant de test parallèle	`create`		Oui	
Connexion d'un port de composant à un autre port de composant	`connect`		Oui	
Déconnexion de deux ports de composants	`disconnect`		Oui	
Mappage d'un port vers une interface du système de test	`map`		Oui	
Démappage d'un port de l'interface du système de test	`unmap`		Oui	
Obtention de la valeur de référence de composant MTC	`mtc`		Oui	Oui
Obtention de valeur de référence de composant d'interface du système de test	`system`		Oui	Oui
Obtention de valeur de référence de composant propre	`self`		Oui	Oui
Début d'exécution de comportement de composant de test	`start`		Oui	
Arrêt d'exécution de comportement de composant de test	`stop`		Oui	
Suppression d'un composant de test du système	`kill`		Oui	
Vérification de terminaison de comportement d'un PTC	`running`		Oui	
Vérification de l'existence d'un PTC dans le système de test	`alive`		Oui	
Attente de terminaison de comportement d'un PTC	`done`		Oui	
Attendre qu'un PTC cesse d'exister	`killed`		Oui	
Opérations de communication				
Envoi de message	`send`		Oui	
Invocation d'appel de procédure	`call`		Oui	
Réponse à un appel de procédure à partir d'une entité distante	`reply`		Oui	
Propagation d'une exception (à un appel accepté)	`raise`		Oui	
Réception de message	`receive`		Oui	
Déclenchement sur message	`trigger`		Oui	
Acceptation d'un appel de procédure provenant d'une entité distante	`getcall`		Oui	

Tableau 11/Z.140 – Aperçu général des expressions, instructions et opérations TTCN-3

Instruction	Mot clé ou symbole associé	Instruction pouvant être utilisée dans la partie commande du module	Instruction pouvant être utilisée dans des fonctions, tests élémentaires et variantes	Instruction pouvant être utilisée dans des fonctions appelées à partir de modèles, sentinelles booléennes ou à partir de définitions locales de variante
Traitement de la réponse provenant d'un appel précédent	**getreply**		Oui	
Acquisition d'une exception (provenant d'une entité appelée)	**catch**		Oui	
Vérification du message/appel reçu (actuellement)	**check**		Oui	
Suppression de file d'attente à un port	**clear**		Oui	
Suppression de file d'attente et activation de l'émission et de la réception à un port	**start**		Oui	
Désactivation de l'émission et interdiction des opérations de réception à apparier à un port	**stop**		Oui	
Désactivation de l'émission et interdiction des opérations de réception à apparier à de nouveaux messages/appels	**halt**		Oui	
Opérations de temporisation				
Armement de temporisateur	**start**	Oui	Oui	
Arrêt de temporisateur	**stop**	Oui	Oui	
Lecture de la durée écoulée	**read**	Oui	Oui	
Vérification d'armement de temporisateur	**running**	Oui	Oui	
Evénement d'expiration de temporisateur	**timeout**	Oui	Oui	
Opérations de verdict				
Mise à jour de verdict local	**setverdict**		Oui	
Requête de verdict local	**getverdict**		Oui	Oui
Actions externes				
Commande d'action externe (SUT)	**action**	Oui	Oui	
Exécution de test élémentaire				
Exécution d'un test élémentaire	**execute**	Oui	Oui (voir Note 2)	
NOTE 1 – Cette instruction ne peut servir qu'à commander des opérations de temporisation. NOTE 2 – Cette instruction ne peut servir que dans des fonctions ou variantes utilisées dans la partie commande du module. NOTE 3 – La modification de variables de composant est interdite. NOTE 4 – Cette instruction peut servir dans des fonctions et variantes mais pas dans des tests élémentaires.				

19 Expressions et instructions de programmation de base

19.0 Généralités

Les expressions et les instructions de programmation de base peuvent être utilisées dans la partie commande d'un module et dans les fonctions, variantes et tests élémentaires de la notation TTCN-3.

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

Instructions de programmation de base	
Instruction	**Mot clé ou symbole associé**
Affectations	`:=`
Journalisation	`log`
Etiqueter et "saut"	`label / goto`
Condition "si-sinon"	`if (…) { … } else { … }`
Boucle "pour"	`for (…) { … }`
Boucle "tant que"	`while (…) { … }`
Boucle "exécuter tant que"	`do { … } while (…)`
Arrêt d'exécution	`stop`
Sélection de test élémentaire	`select case (…) { case (…) {…} case else {…} }`

19.1 Expressions

19.1.0 Généralités

La notation TTCN-3 permet la spécification d'expressions au moyen des opérateurs définis dans le § 15. Les expressions sont construites à partir d'autres expressions (simples). Les expressions ne peuvent utiliser que des fonctions retournant des valeurs. Le résultat d'une expression doit être la valeur d'un type spécifique et les opérateurs utilisés doivent être compatibles avec le type de l'opérande.

EXEMPLE:

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

19.1.1 Expressions booléennes

Une expression de type **`boolean`** ne doit contenir que des valeurs et des opérateurs et/ou opérateurs relationnels de type **`boolean`**. Une telle expression doit prendre la valeur booléenne **`true`** ou **`false`**.

EXEMPLE:

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

19.2 Affectations

Des valeurs peuvent être affectées à des variables, ce qui est indiqué par le symbole ":=". Pendant l'exécution d'une affectation, le côté droit de celle-ci doit prendre une valeur qui soit compatible avec le type de son côté gauche. L'effet d'une affectation est d'associer la variable à la valeur de l'expression. L'expression ne doit contenir aucune variable illimitée. Toutes les affectations se suivent dans l'ordre de leur apparition, c'est-à-dire celui d'un traitement de gauche à droite.

EXEMPLE:

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

19.3 L'instruction "Log"

L'instruction **`log`** permet d'écrire un ou plusieurs items de journalisation dans un dispositif de journalisation associé à la commande de test ou au composant de test dans lequel l'instruction est utilisée. Les items à journaliser doivent être identifiés par une liste d'éléments séparés par des virgules dans l'argument de l'instruction log. Les items de journalisation peuvent être constitués des différents éléments de langage spécifiés dans le Tableau 12b ou d'expressions composées de ces items de journalisation.

Il est vivement recommandé que l'exécution de l'instruction log n'ait aucun effet sur le comportement du test. En particulier, les fonctions utilisées dans une instruction log ne devraient modifier ni explicitement ni implicitement les valeurs de variable de composant, l'état des ports ou des temporisateurs, et ne devraient pas modifier la valeur de l'un quelconque de leurs paramètres **`inout`** ou **`out`**.

EXEMPLE:

```
var integer myVar:= 1;
log("Line 248 in PTC_A: ", myVar, " (actual value of myVar)");
// La chaîne "Line 248 in PTC_A:1(valeur effective de myVar" est écrite
// dans un dispositif de journalisation du système de test
```

NOTE 1 – Les fonctions utilisées dans des instructions log ne devraient pas utiliser directement ou indirectement des instructions autres que **if…else**, **for**, **while**, **do…while**, **label**, **goto**, **return**, **mtc**, **system**, **self**, **running** (PTC ou temporisateur), **read** et **getverdict**.

NOTE 2 – La définition des capacités complexes de journalisation et de suivi du cheminement, qui peut dépendre d'un utilitaire, est hors du domaine d'application de la présente Recommandation.

Tableau 12b/Z.140 – Eléments de langage TTCN-3 qui peuvent être journalisés

Elément utilisé dans une instruction log	Elément journalisé	Commentaire
Identificateur de paramètre de module	Valeur effective	
Valeur de littéral	Valeur	Cette valeur comporte également des commentaires en langage courant
Identificateur de constante de données	Valeur effective	
Identificateur de constante externe	Valeur effective	
Instance de modèle	Valeurs effectives de modèle ou de champ et symboles d'appariement	
Identificateur de variable de type de données	Valeur effective ou chaîne "UNINITIALIZED"	Voir Notes 3 et 4
Identificateurs de variables de type **self**, **mtc**, **system** ou composant	Valeur effective et, s'il est affecté, le nom de l'instance de composant ou chaîne "UNINITIALIZED"	Pour les valeurs effectives de journalisation, voir les Notes 2 à 4. Les états de composant effectifs doivent être journalisés conformément à la Note 5.
Opération d'exécution (composant ou temporisateur)	Valeur de retour	**true** ou **false**. Dans le cas de matrices de composants ou de temporisations, la spécification des éléments matriciels doit être incluse.
Opération `alive` (composant)	Valeur de retour	**true** ou **false**. Dans le cas de matrices, les spécifications des éléments matriciels doivent être incluses.
Instance de port	Etat effectif	Les états du port doivent être journalisés conformément à la Note 6.
Identificateur de variable de type par défaut	Etat effectif ou chaîne "UNINITIALIZED"	Les états par défaut doivent être journalisés conformément à la Note 7. Voir aussi les Notes 2 à 4.
Nom de temporisateur	Etat effectif	Les états du temporisateur doivent être journalisés conformément à la Note 8.
Opération read	Valeur de retour	Voir § 24.3.
Fonctions prédéfinies	Valeur de retour	Voir l'Annexe C.
Instance de fonction	Valeur de retour	Seules les fonctions avec une clause return sont autorisées
Instance de fonction externe	Valeur de retour	Seules les fonctions externes avec une clause return sont autorisées.
Identificateur de paramètres formels	Voir la colonne commentaire	La journalisation des paramètres effectifs doit suivre les règles spécifiées pour les éléments de langage qu'ils remplacent. Les éléments à journaliser sont: dans le cas de paramètres de valeur, la valeur des paramètres effectifs; dans le cas de paramètres de type modèle, les valeurs de modèle ou de champ effectives et les symboles d'appariement; dans le cas de paramètres de type composant, la référence de composant effective, etc. En outre, pour les paramètres de temporisation, l'utilisation de l'opération read est autorisée, ainsi que l'utilisation de l'opération running pour les paramètres de type composant et de temporisation.

Tableau 12b/Z.140 – Eléments de langage TTCN-3 qui peuvent être journalisés

NOTE 1 – On entend par valeur effective/modèle effectif la valeur ou le modèle au moment de l'exécution de l'instruction log.
NOTE 2 – Le type de la valeur journalisée dépend de l'utilitaire utilisé.
NOTE 3 – Dans le cas d'identificateurs de matrices sans spécification d'éléments matriciels, les valeurs effectives doivent être journalisées, ainsi que pour les noms des références de composant de tous les éléments matriciels.
NOTE 4 – La chaîne "UNINITIALIZED" ne doit être journalisée que si l'item log est illimité (non initialisé).
NOTE 5 – Les états de composant qui peuvent être journalisés sont: Inactive, Running, Stopped et Killed (pour plus de détails voir l'Annexe F).
NOTE 6 – Les états de port qui peuvent être journalisés sont: Started et Stopped (pour plus de détails voir l'Annexe F).
NOTE 7 – Les états par défaut qui peuvent être journalisés sont: Activated et Deactivated.
NOTE 8 – Les états de temporisation qui peuvent être journalisés sont: Inactive, Running et Expired (pour plus de détails voir l'Annexe F).

19.4 L'instruction "Label"

L'instruction **label** permet de spécifier des étiquettes dans des tests élémentaires, dans des fonctions, dans des variantes et dans la partie commande d'un module. Une instruction **label** peut être utilisée librement comme d'autres instructions de programmation comportementales TTCN-3 conformément aux règles syntaxiques définies dans l'Annexe A. Elle peut être utilisée avant ou après une instruction TTCN-3 mais non comme la première instruction d'une option normale ou supérieure dans une instruction **alt**, **interleave** ou **altstep**. Les étiquettes utilisées après le mot clé **label** doivent être uniques parmi toutes les étiquettes définies dans le même test élémentaire, dans la même fonction, dans la même variante ou dans la même partie commande.

EXEMPLE:

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

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

19.5 L'instruction "Goto" (saut)

L'instruction **goto** peut être utilisée dans les fonctions, tests élémentaires, variantes et dans la partie commande d'un module TTCN-3. L'instruction **goto** effectue un saut jusqu'à une étiquette de type **label.**

L'instruction **goto** permet de sauter librement (c'est-à-dire en avant et en arrière) dans une séquence d'instructions, d'effectuer un saut à partir d'une unique instruction composite (par exemple à partir d'une boucle de type **while**) et de sauter plusieurs niveaux à partir d'instructions composites imbriquées (par exemple à partir d'options imbriquées). Cependant, l'utilisation de l'instruction **goto** doit être limitée par les règles suivantes:

a) il n'est pas autorisé de sauter à destination ou en provenance de fonctions, de tests élémentaires, de variantes et de la partie commande d'un module TTCN-3;

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

c) il n'est pas autorisé d'utiliser l'instruction **goto** dans une instruction de type **interleave**.

EXEMPLE:

```
// Le fragment de code TTCN-3 suivant comprend
 :
label L1;                          // … la définition d'étiquette L1,
MyVar := 2 * MyVar;
if (MyVar < 2000) { goto L1; }     // … un saut en arrière vers L1,
MyVar2 := Myfunction(MyVar);
if (MyVar2 > MyVar) { goto L2; }   // … un saut en avant vers L2,
PCO1.send(MyVar);
PCO1.receive -> value MyVar2;
label L2;                          // … la définition d'étiquette L2,
PCO2.send(integer: 21);
alt {
  [] PCO1.receive { }
  [] PCO2.receive(integer: 67) {
      label L3;                    // … la définition d'étiquette L3,
      PCO2.send(MyVar);
      alt {
        [] PCO1.receive { }
        [] PCO2.receive(integer: 90) {
           PCO2.send(integer: 33);
           PCO2.receive(integer: 13);
           goto L4;                // … un saut en avant à partir de deux
                                   // instructions alt imbriquées,
           }
        [] PCO2.receive(MyError) {
           goto L3;                // … un saut en arrière à
                                   // partir d'une instruction actuelle alt,
           }
        [] any port.receive {
           goto L2;                // … un saut en arrière à partir
                                   // de deux instructions alt imbriquées,
           }
       }
      }
  [] any port.receive {
       goto L2;                    // … et un long saut en arrière
                                   // à partir d'une instruction alt.
      }
}
label L4;
 :
```

19.6 L'instruction "If-else" (échappement conditionnel)

L'instruction **`if-else`**, également appelée *instruction conditionnelle*, sert à indiquer un branchement logique dans le flux de commande en raison d'expressions de type **`boolean`**. Schématiquement, l'instruction conditionnelle se présente comme suit:

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

Où le terme `statementBlock`$_x$ se rapporte à un bloc d'instructions.

EXEMPLE:

```
if (date == "1.1.2005") { return ( fail ); }

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

Une proposition plus complexe pourrait être:

```
if (expression1)
  statementblock1
```

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

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

19.7 L'instruction for (pour)

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

L'instruction **`for`** contient deux affectations et une expression de type **`boolean`**. La première affectation est nécessaire afin d'initialiser la variable d'indice (ou de compteur) de la boucle. L'expression de type **`boolean`** achève la boucle et la seconde affectation sert à manipuler la variable d'indice.

EXEMPLE 1:

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

Le critère de terminaison de la boucle doit être exprimé par l'expression de type **`boolean`**. Il est vérifié au début de chaque nouvelle itération de la boucle. S'il s'évalue à **`true`**, l'exécution continue avec le bloc d'instructions contenu dans l'instruction **`for`**. S'il s'évalue à **`false`**, l'exécution continue avec l'instruction qui suit immédiatement la boucle **`for`**.

La variable d'indice d'une boucle de type **`for`** peut être déclarée avant d'être utilisée dans l'instruction **`for`** ou peut être déclarée et initialisée dans l'en-tête de l'instruction **`for`**. Si la variable d'indice est déclarée et initialisée dans l'en-tête de l'instruction **`for`**, la portée de la variable d'indice est limitée au corps de boucle, c'est-à-dire qu'il n'est visible qu'à l'intérieur de ce corps de boucle.

EXEMPLE 2:

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

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

19.8 L'instruction "While" (tant que)

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

EXEMPLE:

```
while (j<10){ … }
```

19.9 L'instruction "Do-while" (exécution tant que)

La boucle de type **`do-while`** est identique à une boucle de type **`while`** sauf que la condition de la boucle doit être vérifiée à la *fin* de chaque itération de la boucle. Autrement dit, lors de l'utilisation d'une boucle **`do-while`**, le comportement est exécuté au moins une fois avant que la condition de la boucle soit évaluée pour la première fois.

EXEMPLE:

```
do { … } while (j<10);
```

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

L'instruction **stop** donne à une exécution une fin différente selon le contexte dans lequel elle est utilisée. Quand elle est utilisée dans la partie commande d'un module ou dans une fonction employée par la partie commande d'un module, cette instruction achève l'exécution de la partie commande du module. Quand elle est utilisée dans un test élémentaire, dans une variante ou dans une fonction qui s'exécute sur un composant de test, cette instruction met fin au composant de test correspondant.

EXEMPLE:

```
module MyModule {
   : // Module définitions
  testcase MyTestCase() runs on MyMTCType system MySystemType{
      var MyPTCType ptc:= MyPTCType.create;    // PTC création de composant PTC
      ptc.start(MyFunction());                // début d'exécution de composant PTC
       :                                      // maintien du comportement de test
                                              // élémentaire
     stop     // arrête le composant MTC, tous les composants PTC et
              // le test élémentaire tout entier
  }
  function MyFunction() runs on MyPTCType {
      :
     stop     // n'arrête que le composant PTC; le test élémentaire
              // continue
  }
  control {
      :   // exécution du test
     stop // arrête la campagne de tests
  } // fin de la commande
} // fin du module test
```

NOTE – La sémantique d'une instruction **stop** qui arrête un composant de test est identique à l'opération d'arrêt de composant **self.stop** (voir § 22.6).

19.11 L'instruction "Select Case"

L'instruction **select case** peut être utilisée à la place des instructions **if .. else** lorsqu'on compare une valeur à une ou plusieurs autres valeurs. L'instruction contient une partie en-tête et zéro ou plusieurs branches. Jamais plusieurs branches ne sont exécutées en même temps. Schématiquement, l'instruction **select case** se présente comme suit:

```
select (expression)
   {
  case (templateInstance1a, templateInstance1b,…)
     statementblock1
  case (templateInstance2a, templateInstance2b,…)
     statementblock2
  case else
     statementblock3
   }
```

Où templateInstance fait référence à un modèle défini ou en ligne et $statementblock_x$ fait référence à un bloc d'instructions.

NOTE – La présentation schématique ci-dessus est équivalente à la présentation schématique suivante qui utilise les instructions if-else:

```
if (match(expression, templateInstance1a or match(expression, templateInstance1b
or …)
  statementblock1
else if (match(expression, templateInstance2a or match(expression, templateInstance2b or …)
  statementblock2
else
  statementblock3
```

Dans la partie en-tête de l'instruction **select case,** une expression doit être indiquée. Chaque branche commence par le mot-clé **case** suivi par une liste de productions templateInstance (une branche d'énumération list, qui peut aussi contenir un élément isolé) ou le mot-clé **else** (une branche d'échappement else) et un bloc d'instructions.

Toutes les productions templateInstance contenues dans toutes les branches list doivent être d'un type compatible avec le type de l'expression figurant dans l'en-tête. Une branche list est sélectionnée et le bloc d'instructions de la branche sélectionnée n'est exécuté que si l'une quelconque des productions templateInstance s'apparie avec la valeur de l'expression figurant dans l'en-tête de l'instruction. Une fois exécuté le bloc d'instructions de la branche sélectionnée

(c'est-à-dire sans effectuer un saut à partir d'une instruction go to), l'exécution continue avec l'instruction qui suit l'instruction select case.

Le bloc d'instructions d'une branche else est toujours exécuté si aucune autre branche précédant textuellement la branche else n'a été sélectionnée.

Les branches sont évaluées dans leur ordre textuel. Si aucune des productions templateInstance ne s'apparie avec la valeur de l'expression figurant dans l'en-tête et que l'instruction ne contienne aucune branche else, l'exécution continue sans qu'aucune des branches **select case** ne soit exécutée.

EXEMPLE:

```
select (MyModulePar)  // où MyModulePar est de type chaîne de caractères
{
  case ("firstValue")
      {
       log ("The first branch is selected");
      }
  case (MyCharVar, MyCharConst)
      {
       log ("The second branch is selected");
      }
  case else
      {
       log ("The value of the module parameter MyModulePar is selected");
      }
}
```

20 Instructions de programmation comportementales

20.0 Généralités

Les instructions de programmation comportementales peuvent être utilisées dans les tests élémentaires, dans les fonctions et les variantes, sauf pour:

a) l'instruction **return**, qui ne doit être utilisée que dans des fonctions;

b) l'instruction **alt**, l'instruction **interleave** et l'instruction **repeat,** qui peuvent également être utilisées dans la partie commande du module.

Les instructions de programmation comportementales spécifient le comportement dynamique des composants de test aux ports de communication. Le comportement de test peut être exprimé séquentiellement, comme un ensemble d'options ou comme une combinaison de séquences et d'options. Un opérateur d'entrelacement permet la spécification de séquences ou d'options entrelacées.

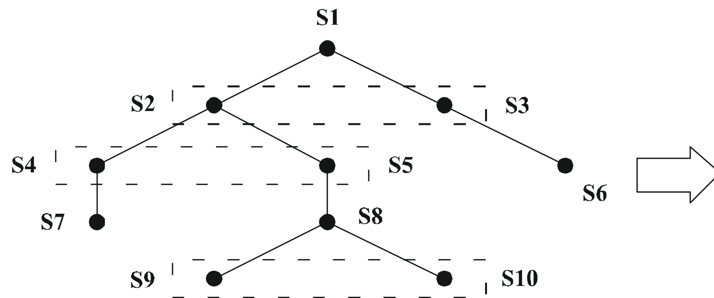
Tableau 13/Z.140 – Aperçu général des instructions de programmation comportementales TTCN-3

Instructions de programmation comportementales	
Instructions	**Mot clé ou symbole associé**
Comportement à options	**alt { … }**
Réévaluation d'instructions "alt"	**repeat**
Comportement entrelacé	**interleave { … }**
Commande de retour	**return**

20.1 Comportement à options

20.1.0 Généralités

Une forme de comportement plus complexe est celui où des séquences d'instructions sont exprimées comme des ensembles d'options offertes de façon à former une arborescence de chemins d'exécution, comme illustré dans la Figure 9.




```
S1;
alt  {
   [] S2 {
          alt  {
            []  S4  { S7 }
            []  S5  {
                    S8;
                    alt  {
                         []  S9  {}
                         []  S10 {}
                       }
                   }
             }
       }
   []  S3   { S6 }
}
```

Z.140_F09

Figure 9/Z.140 – Illustration du comportement à options

L'instruction **`alt`** indique un branchement logique de comportement de test dû à la réception et à la manipulation d'événements de communication et/ou de temporisation et/ou à la terminaison de composants de test parallèles, c'est-à-dire que cette instruction est associée à l'utilisation des opérations TTCN-3 **`receive`**, **`trigger`**, **`getcall`**, **`getreply`**, **`catch`**, **`check`**, **`timeout`**, **`done`** et **`killed`**. L'instruction **`alt`** indique un ensemble d'événements possibles qui doivent être appariés en fonction d'un instantané particulier (voir § 20.1.1).

EXEMPLE:

```
// Utilisation d'instructions "alt" imbriquées
 :
alt {
  [] L1.receive(DL_REL_CO:*) {
      setverdict(pass);
      TAC.stop;
      TNOAC.start;
      alt {
        [] L1.receive(t_DL_EST_IN) {
              TNOAC.stop;
              setverdict(pass);
           }
        [] TNOAC.timeout {
           L1.send(t_DEL_EST_RQ);
           TAC.start;
           alt {
             [] L1.receive(DL_EST_CO:*) {
                   TAC.stop;
                   setverdict(pass)
              }
              [] TAC.timeout {
                   setverdict(inconc);
                 }
             [] L1.receive {
                   setverdict(inconc)
                 }
           }
       }
        [] L1.receive {
              setverdict(inconc)
           }
      }
     }
  [] TAC.timeout {
      setverdict(inconc)
     }
  [] L1.receive {
      setverdict(inconc)
     }
}
 :
```

20.1.1 Exécution d'un comportement à options

Lors de l'entrée dans une instruction de type **alt**, un instantané est pris. Un instantané est considéré comme étant un état partiel d'un composant de test qui comprend toutes les informations nécessaires afin d'évaluer les conditions booléennes de garde des branches facultatives, tous les composants de test arrêtés qui sont applicables, tous les événements applicables d'expiration de temporisation ainsi que les messages, appels, réponses et exceptions de niveau supérieur contenus dans les files applicables d'attente d'entrée dans un port. Tout composant de test, toute temporisation et tout port qui est indiqué dans au moins une option de l'instruction **alt**, ou dans l'option sommitale d'une variante invoquée en tant qu'option dans l'instruction **alt** ou activée par défaut, est considéré comme étant applicable. Une description détaillée de l'instantané sémantique est reproduite dans la sémantique opérationnelle de la notation TTCN-3 (Rec. UIT-T Z.143 [3]).

NOTE 1 – Les instantanés d'analyse ne sont qu'un moyen théorique permettant de décrire le comportement de l'instruction **alt**. Les algorithmes concrets permettant la manipulation de l'instantané peuvent être trouvés dans la Rec. UIT-T Z.143 [3].

NOTE 2 – La sémantique TTCN-3 part du principe que la prise d'un instantané est immédiate, c'est-à-dire sans aucune durée. Dans une implémentation réelle, la prise d'un instantané peut prendre un certain temps et des conditions critiques peuvent se produire. La manipulation de telles conditions critiques est hors du domaine d'application de la présente Recommandation.

Les branches facultatives contenues dans l'instruction **alt** et les options sommitales des variantes invoquées et des variantes activées par défaut sont traitées dans l'ordre de leur apparition. Si plusieurs valeurs par défaut sont actives, l'ordre inverse de leur activation détermine l'ordre d'évaluation des options sommitales contenues dans les valeurs par défaut. Les branches facultatives contenues dans valeurs par défaut actives sont atteintes au moyen du mécanisme de valeurs par défaut décrit dans le § 21.

Chaque branche facultative est soit une branche qui peut être gardée par une expression booléenne ou des branches d'échappement conditionnel, c'est-à-dire des branches facultatives commençant par [**else**].

Les branches d'échappement conditionnel sont toujours choisies et exécutées quand elles sont atteintes (voir § 20.1.3).

Les branches qui peuvent être gardées par des expressions booléennes invoquent soit une variante (*branche de variante*), ou commencent par une opération de fin d'exécution **done** (*branche de fin d'exécution*), par une opération **killed** (*branche supprimée*), par une opération de fin de temporisation de type **timeout** (*branche d'expiration de temporisation*) et par une opération de réception (*branche de réception*), c'est-à-dire de type **receive**, **trigger**, **getcall**, **getreply**, **catch** ou **check**. L'évaluation des sentinelles booléennes doit être fondée sur l'instantané. La sentinelle booléenne est considérée comme étant *satisfaite* si aucune sentinelle booléenne est définie, ou si la sentinelle booléenne s'évalue comme étant de type **true**. Les branches sont traitées et exécutées de la manière suivante:

Une *branche de variante* est choisie si la sentinelle booléenne est satisfaite. La sélection d'une *branche de variante* provoque l'invocation de la variante désignée, c'est-à-dire que la variante est invoquée et que l'évaluation de l'instantané se poursuit à l'intérieur de la variante.

Une *branche de fin d'exécution* est choisie si la sentinelle booléenne est satisfaite et si le composant de test spécifié est contenu dans la liste des composants arrêtés de l'instantané. La sélection provoque l'exécution du bloc d'instructions qui suit l'opération de fin d'exécution. L'opération de fin d'exécution **done** n'a par elle-même aucun autre effet.

Une *branche supprimée* est choisie si la sentinelle booléenne est satisfaite et si le composant de test spécifié est contenu dans la liste des composants supprimés de l'instantané. La sélection provoque l'exécution du bloc d'instructions qui suit l'opération **killed**. L'opération **killed** n'a par elle-même aucun autre effet.

Une *branche d'expiration de temporisation* est choisie si la sentinelle booléenne est satisfaite et si l'événement spécifié d'expiration de temporisation est contenu dans la liste des temporisations expirées de l'instantané. La sélection provoque l'exécution de l'opération de type **timeout** spécifiée, c'est-à-dire que la suppression de l'événement d'expiration de temporisation à partir de la liste des temporisations expirées et l'exécution du bloc d'instructions faisant suite à l'opération **timeout**.

Une *branche de réception* est choisie si la sentinelle booléenne est satisfaite et si le critère d'opération de réception correspondant est satisfait par un des messages, des appels, des réponses ou des exceptions contenus dans l'instantané. La sélection provoque l'exécution de l'opération de réception, c'est-à-dire la suppression du message, de l'appel, de la réponse ou de l'exception correspondant de la file d'attente à un port, avec éventuellement une affectation des informations reçues à une variable et l'exécution du bloc d'instructions faisant suite à l'opération de réception. Dans le cas de l'opération **trigger**, le message sommital de la file est également supprimé si la sentinelle booléenne est satisfaite mais que les critères de correspondance ne le soient pas. Dans ce cas, le bloc d'instructions de l'option en cause n'est pas exécuté.

NOTE 3 – La sémantique TTCN-3 décrit l'évaluation d'un instantané comme une série actions indivisibles d'un composant de test. La sémantique n'implique pas que l'évaluation d'un instantané n'ait aucune durée. Pendant l'évaluation d'un instantané, des composants de test peuvent s'arrêter, des temporisations peuvent arriver à expiration et de nouveaux messages, appels, réponses

ou exceptions peuvent entrer dans les files d'attente à un port du composant. Cependant, ces événements ne modifient pas l'instantané effectif et ne sont donc pas pris en considération pour l'évaluation de l'instantané.

Si aucune des branches facultatives contenues dans l'instruction **alt** ni aucune des options sommitales contenues dans les variantes invoquées et valeurs par défaut actives ne peut être choisie et exécutée, l'instruction **alt** doit être réexécutée, c'est-à-dire qu'un nouvel instantané est pris et que l'évaluation des branches facultatives est répétée avec le nouvel instantané. Cette procédure répétitive doit continuer jusqu'à ce qu'une branche facultative soit choisie et exécutée, ou que le test élémentaire soit arrêté par un autre composant ou par le système de test (p. ex. parce que le composant MTC est arrêté) ou par une erreur dynamique.

Le test élémentaire doit s'arrêter et indiquer une erreur dynamique si un composant de test est complètement bloqué. Autrement dit, aucune des options ne peut être choisie, aucun composant de test applicable n'est en cours d'exécution, aucune temporisation applicable n'est en cours d'exécution et tous les ports applicables contiennent au moins un message, un appel, une réponse ou une exception qui ne s'apparie pas.

NOTE 4 – La procédure répétitive de prise d'un instantané complet et de réévaluation de toutes les options n'est qu'un moyen théorique de décrire la sémantique de l'instruction **alt**. L'algorithme concret qui met en œuvre cette sémantique est hors du domaine d'application de la présente Recommandation.

20.1.2 Sélection/désélection d'une option

Si nécessaire, il est possible d'activer/désactiver une option au moyen d'une expression booléenne placée entre les '[ ]' crochets de l'option.

L'évaluation d'une expression booléenne gardant une option peut avoir des effets secondaires. Afin d'éviter des effets secondaires qui provoquent une incohérence entre l'instantané effectif et l'état du composant, les mêmes restrictions que pour l'initialisation de définitions locales dans des variantes doivent s'appliquer (voir § 16.2.2.1).

Les crochets d'ouverture et de fermeture '[' ']' doivent être présents au début de chaque option, même s'ils sont vides. Cela non seulement augmente la lisibilité mais est également nécessaire afin de distinguer syntaxiquement une option d'une autre.

EXEMPLE:

```
// Utilisation d'une instruction alt avec des expressions boléennes
// (ou avec une sentinelle)
  :
alt {
   [x>1] L2.receive {            // Expression/sentinelle boléenne
          setverdict(pass);
      }
   [x<=1] L2.receive {           // Expression/sentinelle boléenne
           setverdict(inconc);
      }
}
  :
```

20.1.3 Branche d'échappement dans une instruction "alt"

Toute branche d'une instruction **alt** peut être définie comme une branche d'échappement par insertion du mot clé **else** entre les crochets d'ouverture et de fermeture situés au début de l'option. La branche d'échappement ne doit contenir aucune des actions autorisées dans les branches gardées par une expression booléenne (c'est-à-dire un appel de variante **altstep** ou une opération de fin d'exécution **done**, une opération **killed**, une opération d'expiration de temporisation ou de réception). Le bloc d'instructions de la branche d'échappement est toujours exécuté si aucune autre option n'apparaît avant le texte de cette branche d'échappement.

EXEMPLE:

```
// Utilisation d'une instruction alt avec des expressions
// (ou sentinelles) boléennes et une
// branche d'échappement
  :
alt {
   [x>1] L2.receive {
        setverdict(pass);
      }
   [x<=1] L2.receive {
        setverdict(inconc);
      }
   [else] {              // branche d'échappement
        MyErrorHandling();
        setverdict(fail);
```

```
        stop;
       }
}
  :
```

Il y a lieu de noter que le mécanisme de valeurs par défaut (voir § 21) est toujours invoqué à la fin de chaque instruction "alt". Si une branche d'échappement **`else`** est définie, le mécanisme de valeurs par défaut n'est jamais appelé, c'est-à-dire que des valeurs par défaut actives ne sont jamais introduites.

NOTE 1 – Il est également possible d'utiliser une branche **`else`** dans des variantes.

NOTE 2 – Il est permis d'utiliser une instruction de type **`repeat`** comme dernière instruction d'une branche de type **`else`**.

NOTE 3 – Il est permis de définir plusieurs branches d'échappement dans une instruction "alt" ou dans une variante, mais seule la première branche d'échappement est toujours exécutée.

20.1.4 Vide

20.1.5 Réévaluation d'instruction "alt"

La réévaluation d'une instruction **`alt`** peut être spécifiée au moyen d'une instruction de type **`repeat`** (voir § 20.2).

EXEMPLE:

```
alt {
   [] PCO3.receive {
       count := count + 1;
       repeat                  // utilisation de la répétition
      }
   [] T1.timeout { }
   [] any port.receive {
       setverdict(fail);
       stop;
      }
}
```

20.1.6 Invocation de variantes en tant qu'options

La notation TTCN-3 permet l'invocation de variantes en tant qu'options dans des instructions de type **`alt`** (voir § 16.2.3).

EXEMPLE:

```
  :
alt {
   [] PCO3.receive { }
   [] AnotherAltStep(); // appel explicite de la variante AnotherAltStep
                        // comme option d'une instruction alt
   [] MyTimer.timeout { }
}
  :
```

20.2 L'instruction "Repeat" (répétition)

L'instruction **`repeat`** (répétition), lorsqu'elle est utilisée dans le bloc d'instructions et de déclarations des options des instructions **`alt`**, provoque la réévaluation de l'instruction **`alt`**, c'est-à-dire qu'un nouvel instantané est pris et que les options de l'instruction **`alt`** sont évaluées dans l'ordre de leur spécification. Lorsqu'elle est utilisée dans des blocs d'instructions et des déclarations des parties *réponse* et *traitement d'exception* d'appels bloquants en mode procédure, l'instruction **`repeat`** provoque la réévaluation de la partie réponse et traitement d'exception de l'appel (voir § 23.3.1.5).

EXEMPLE 1:

```
// Utilisation de l'instruction repeat dans une instruction alt
alt {
   [] PCO3.receive {
       count := count + 1;
       repeat           // utilisation de la répétition
      }
   [] T1.timeout { }
   [] any port.receive {
       setverdict(fail);
       stop;
      }
}
```

Si une instruction **repeat** est utilisée dans une option sommitale dans une définition de variante, cette instruction provoque un nouvel instantané et la réévaluation de l'instruction **alt** à partir de laquelle la variante a été appelée. L'appel de la variante peut être effectué soit implicitement par le mécanisme de valeurs par défaut (voir § 21) ou explicitement dans l'instruction **alt** (voir § 20.1.6).

EXEMPLE 2:

```
// utilisation de la répétion dans une variante
altstep AnotherAltStep() runs on MyComponentType {
   [] PCO1.receive{
       setverdict(inconc);
       repeat                 // utilisation de la répétition
       }
   [] PCO2.receive {}
}
```

20.3 Comportement entrelacé

L'instruction **interleave** permet de spécifier l'apparition et le traitement de l'entrelacement des instructions **done**, **killed**, **timeout**, **receive**, **trigger**, **getcall**, **catch** et **check**.

Les instructions de transfert de commande **for**, **while**, **do-while**, **goto**, **activate**, **deactivate**, **stop**, **repeat**, **return**, les appels de variantes en tant qu'options et les appels (directs et indirects) de fonctions définies par l'utilisateur, qui comportent des opérations de communication, ne doivent pas être utilisés dans des instructions de type **interleave**. En outre, il n'est pas autorisé de placer des sentinelles devant les branches d'une instruction **interleave** au moyen d'expressions booléennes (c'est-à-dire que les crochets '[ ]' doivent toujours être vides). Il n'est pas non plus autorisé de spécifier des branches de type **else** dans un comportement entrelacé.

Un comportement entrelacé peut toujours être remplacé par un ensemble équivalent d'options imbriquées. Les procédures de ce remplacement et la sémantique opérationnelle d'entrelacement sont décrites dans la Rec. UIT-T Z.143 [3].

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

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

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

b) l'évaluation se poursuit alors par la prise du prochain instantané.

La sémantique opérationnelle d'entrelacement est entièrement définie dans la Rec. UIT-T Z.143 [3].

EXEMPLE:

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

// est un abrégé pour
:
alt {
[] PCO1.receive(MySig1)
   {   PCO1.send(MySig2);
       alt {
       []    PCO1.receive(MySig3)
        {    PCO2.receive(MySig4);
             PCO2.send(MySig5);
             PCO2.send(MySig6);
```

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

:
```

20.4 L'instruction "Return" (retour)

L'instruction **return** met fin à l'exécution d'une fonction ou d'une variante et fait revenir la commande au point à partir duquel la fonction ou la variante a été appelée. Lorsqu'elle est utilisée dans des fonctions, une instruction **return** peut (à titre d'option) être associée à une valeur de retour.

NOTE – Lorsqu'elle est utilisée dans des variantes, l'instruction **return** a le même effet que si la fin du bloc d'instructions et de déclarations de l'option choisie était atteinte; par exemple, lorsque la variante est appelée à partir de l'instruction **alt**, l'exécution reprend à la première instruction qui suit l'instruction **alt**.

EXEMPLE:

```
function MyFunction() return boolean {
    :
   if (date == "1.1.2005") {
      return false; // l'exécution s'arrête sur le 1.1.2000 et retourne le booléen "false"
   }
    :
   return true; // le booléen "true" est retourné
}

function MyBehaviour() return verdicttype {
    :
   if (MyFunction()) {
      setverdict(pass); // utilisation de MyFunction dans une instruction if
   }
   else {
      setverdict(inconc);
   }
    :
   return getverdict; // retour explicite du verdict
}
```

21 Manipulation des valeurs par défaut

21.0 Généralités

La notation TTCN-3 permet l'activation de variantes (voir § 16.2) par défaut. Pour chaque composant de test, les valeurs par défaut, c'est-à-dire les variantes activées, sont mémorisées sous la forme d'une liste ordonnée. Les valeurs par défaut sont énumérées dans l'ordre inverse de leur activation, c'est-à-dire que la valeur par défaut activée en dernier est le premier élément de la liste des valeurs par défaut actives. Les opérations TTCN-3 **`activate`** (voir § 21.3) et **`deactivate`** (voir § 21.4) fonctionnent d'après la liste de valeurs par défaut. Une opération **`activate`** ajoute une nouvelle valeur par défaut comme premier élément de la liste et une opération **`deactivate`** supprime une valeur par défaut de la liste. Une valeur par défaut dans la liste des valeurs par défaut peut être identifiée au moyen de la référence de valeur par défaut qui est produite à la suite de l'opération **`activate`** correspondante.

Tableau 14/Z.140 – Aperçu général des instructions TTCN-3 pour la manipulation des valeurs par défaut

Instructions pour manipulation des valeurs par défaut	
Instruction	**Mot clé ou symbole associé**
Activer une valeur par défaut	**`Activate`**
Désactiver une valeur par défaut	**`deactivate`**

21.1 Le mécanisme de comportement par défaut

Le mécanisme de comportement par défaut est évoqué à la fin de chaque instruction **`alt`** si, en raison de l'instantané effectif, aucune des options spécifiées n'a pu être exécutée. Un mécanisme évoqué par défaut invoque la première variante contenue dans la liste des valeurs par défaut, c'est-à-dire la valeur par défaut activée en dernier, et attend le résultat de sa terminaison. Celle-ci peut être efficace ou inefficace. Le qualificatif *inefficace* signifie qu'aucune des options sommitales de la variante **`altstep`** (voir § 16.2) définissant le comportement par défaut n'a pu être sélectionnée; le qualificatif *efficace* signifie qu'une seule des options sommitales de la valeur par défaut a été choisie et exécutée.

Dans le cas d'un échec de terminaison, le mécanisme de valeurs par défaut invoque la prochaine valeur par défaut contenue dans la liste. Si la dernière valeur par défaut contenue dans la liste s'est terminée inefficacement, le mécanisme de valeurs par défaut retournera à l'emplacement de l'instruction **`alt`** où il a été invoqué, c'est-à-dire à la fin de l'instruction **`alt`** et va indiquer une exécution inefficace de valeur par défaut. Une exécution inefficace de valeur par défaut va également être indiquée si la liste des valeurs par défaut est vide.

Une exécution inefficace de valeur par défaut peut provoquer un nouvel instantané ou une erreur dynamique si le composant de test est bloqué (voir § 20.1).

Dans le cas d'un succès de terminaison, soit la valeur par défaut peut arrêter le composant de test au moyen d'une instruction **`stop`**, ou le flux de commande principal du composant de test va continuer immédiatement après l'instruction **`alt`** à partir de laquelle le mécanisme de valeurs par défaut a été appelé, ou bien le composant de test va prendre un nouvel instantané et réévaluer l'instruction **`alt`**. Celle-ci doit être spécifiée au moyen d'une instruction **`repeat`** (voir § 20.2). Si l'option sommitale choisie dans la valeur par défaut se termine sans instruction **`repeat`**, le flux de commande du composant de test va continuer immédiatement après l'instruction **`alt`**.

NOTE – La notation TTCN-3 ne limite pas l'implémentation du mécanisme de valeurs par défaut. Celui-ci peut par exemple être implémenté sous la forme d'un processus implicitement appelé à la fin de chaque instruction **`alt`** ou sous la forme d'un fil d'exécution distinct, qui n'est chargé que de la manipulation des valeurs par défaut. La seule exigence est que les valeurs par défaut soient appelées dans l'ordre inverse de leur activation quand le mécanisme de valeurs par défaut a été invoqué.

21.2 Références de valeurs par défaut

Les références de valeurs par défaut sont des références uniques à des valeurs par défaut activées. Une référence unique de valeur par défaut est produite par un composant de test quand une variante est activée par défaut, c'est-à-dire qu'une référence de valeur par défaut est le résultat d'une opération de type **`activate`** (voir § 21.3).

Les références de valeurs par défaut sont du type spécial et prédéfini **`default`**. Les variables de type **`default`** peuvent servir à manipuler des valeurs par défaut activées dans des composants de test. La valeur spéciale **`null`** permet d'indiquer une référence de valeur par défaut indéfinie, p. ex. pour l'initialisation de variables manipulant des références de valeurs par défaut.

Les références de valeurs par défaut sont utilisées dans les opérations de type **deactivate** (voir § 21.4) afin d'identifier la valeur par défaut à désactiver.

La représentation effective des données de type **default** doit être résolue à l'extérieur par le système de test. Cela permet de spécifier des tests élémentaires abstraits indépendamment de tout environnement d'exécution TTCN-3 réel. En d'autres termes, la notation TTCN-3 ne limite pas l'implémentation d'un système de test en ce qui concerne la manipulation et l'identification de valeurs par défaut.

EXEMPLE:

```
// Déclaration d'une variable pour le traitement des valeurs par défaut
// et initialisation avec la valeur null
var default MyDefaultVar := null;
  :
// Utilisation de MyDefaultVar pour mémoriser une valeur par défaut activée
MyDefaultVar := activate(MyDefAltStep()); // La variante MyDefAltStep est activée par défaut
  :
// Utilisation de MyDefaultVar pour la désactivation de la valeur par
// défaut MyDefAltStep
deactivate(MyDefaultVar);
  :
```

21.3 L'opération Activate (activation)

21.3.0 Généralités

L'opération **activate** sert à activer les variantes par défaut. Cette opération va placer la variante désignée comme premier élément de la liste de valeurs par défaut et retournera une référence de valeur par défaut. La référence de valeur par défaut est un identificateur unique de la valeur par défaut qui peut être utilisé dans une opération **deactivate** pour la désactivation de la valeur par défaut.

L'effet d'une opération **activate** est localisé dans le composant de test dans lequel il est appelé. Autrement dit, un composant de test ne peut pas activer une valeur par défaut dans un autre composant de test.

EXEMPLE 1:

```
  :
// Déclaration d'une variable pour le traitement de valeurs par défaut
var default MyDefaultVar := null;
  :
// Déclaration d'une variable de référence de valeur par défaut et
// activation d'une variante
// par défaut
var default MyDefVarTwo := activate(MySecondAltStep());
  :
// Activation de la variante MyAltStep par défaut
MyDefaultVar := activate(MyAltStep()); // MyAltStep est activée par défaut
  :
```

L'opération **activate** peut être appelée sans sauvegarder la référence de valeur par défaut retournée. Cette forme est utile dans les tests élémentaires qui ne nécessitent pas la désactivation de la valeur par défaut activée, c'est-à-dire que la désactivation d'une valeur par défaut est effectuée implicitement lors de la terminaison du composant MTC.

EXEMPLE 2:

```
  :
// Activation d'une variante par défaut, sans affectation d'une référence
// de valeur par défaut
activate(MyCommonDefault());
```

21.3.1 Activation de variantes paramétrées

Les paramètres effectifs d'une variante paramétrée (voir § 16.2.1) qui devrait être activée par défaut doivent être fournis dans l'instruction **activate** correspondante. Autrement dit, les paramètres effectifs sont associés à la valeur par défaut au moment de l'activation de celle-ci (et non pas, par exemple au moment de son invocation par le mécanisme de valeurs par défaut). Toutes les instances de temporisation contenues dans la liste des paramètres effectifs doivent être déclarées en tant que temporisations localisées de type composant (voir § 8.5.1).

EXEMPLE:

```
altstep MyAltStep2 ( integer    par_value1, MyType par_value2,
                     MyPortType par_port,   timer  par_timer )
{
  :
}

function MyFunc () runs on MyCompType
{ :
var default MyDefaultVar := null;

MyDefaultVar := activate(MyAltStep2(5, myVar, myCompPort, myCompTimer);
   // MyAltStep2 est activée par défaut avec les paramètres effectifs 5 et
   // la valeur de myVar. Une modification de myVar avant un appel de MyAltStep2 par
   // le mécanisme de valeurs par défaut ne va pas modifier les
   // paramètres effectifs de l'appel.
  :
}
```

21.4 L'opération Deactivate (désactivation)

L'opération **deactivate** sert à désactiver des valeurs par défaut, c'est-à-dire des variantes déjà activées. L'opération **deactivate** va supprimer la valeur par défaut référencée de la liste de valeurs par défaut.

L'effet d'une opération **deactivate** est localisé dans le composant de test où il est appelé. Autrement dit, un composant de test ne peut pas désactiver une valeur par défaut dans un autre composant de test.

Une opération **deactivate** sans paramètre désactive toutes les valeurs par défaut d'un composant de test.

L'appel d'une opération **deactivate** avec la valeur spéciale **null** n'a aucun effet. L'appel d'une opération **deactivate** avec une référence de valeur par défaut indéfinie, par exemple. une ancienne référence à une valeur par défaut qui a déjà été désactivée ou une variable de référence de valeur par défaut non initialisée, doit provoquer une erreur d'exécution.

EXEMPLE:

```
  :
var default MyDefaultVar := null;
var default MyDefVarTwo := activate(MySecondAltStep());
var default MyDefVarThree := activate(MyThirdAltStep());
  :
MyDefaultVar := activate(MyAltStep());
  :
deactivate(MyDefaultVar); // deactivates MyAltStep
  :
deactivate;  // désactive toutes les autres valeurs par défaut, c'est-à-dire  MySecondAltStep
             // et MyThirdAltStep en l'occurrence
```

22 Opération de configuration

22.0 Généralités

Les opérations de configuration (voir Tableau 15) servent à installer et à commander les composants de test. Ces opérations ne doivent pas être utilisées que dans les tests élémentaires, fonctions et variantes TTCN-3 (c'est-à-dire qu'elles ne pas être employées dans la partie d'un module relative à la commande).

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

Opération	Explication	Exemples de syntaxe
Opérations de connexion		
connect	Connecte le port d'un composant de test au port d'un autre composant de test	**connect**(ptc1:p1, ptc2:p2);
disconnect	Déconnecte deux ports connectés ou plus	**disconnect**(ptc1:p1, ptc2:p2);
map	Mappe le port d'un composant de test avec le port de l'interface avec le système de test	**map**(ptc1:q, **system**:sutPort1);
unmap	Démappe deux ports mappés ou plus	**unmap**(ptc1:q, **system**:sutPort1);

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

Opération	Explication	Exemples de syntaxe
Opérations de composant de test		

create	Création d'un composant de test normal ou sous-tension (alive), la distinction entre composants de test normaux et "alive" étant effectuée pendant la création (le composant MTC se comporte comme un composant de test normal)	Composants de test hors tension (non-alive):
		`var PTCType c := PTCType.create;`
		Composants de test sous-tension (alive):
		`var PTCType c := PTCType.create alive;`

start	Lancement du comportement de test sur un composant de test; le lancement d'un comportement n'affecte pas l'état des variables, temporisations ou ports du composant	`c.start(PTCBehaviour());`
stop	Arrêt d'un comportement de test sur un composant de test	`c.stop;`
kill	Supprime un composant de test	`c.kill;`
alive	Retourne la valeur true si le composant de test a été créé et qu'il est prêt à exécuter, ou exécute déjà, un comportement; sinon, retourne la valeur false	`if (c.alive) …`
running	Retourne la valeur true tant que le composant de test exécute un comportement; sinon, retourne la valeur false	`if (c.running) …`
done	Vérifie si la fonction en cours d'exécution sur le composant de test a pris fin	`c.done;`
killed	Vérifie si un composant de test a bien été supprimé	`c.killed { … }`
Opérations de référence		
mtc	Transmet la référence au composant MTC	`connect(mtc:p, ptc:p);`
system	Transmet la référence à l'interface du système de test	`map(c:p, system:sutPort);`
self	Transmet la référence au composant de test qui exécute cette opération	`self.stop;`

22.1 L'opération "Create" (création)

Le MTC est le seul composant de test qui est automatiquement créé quand un test élémentaire commence. Tous les autres composants de test (les PTC) doivent être créés explicitement pendant l'exécution du test par l'opération **create**. Un composant est créé avec son ensemble complet de ports, dont les files d'entrée sont vides et avec son ensemble complet de constantes, de variables et de temporisations. Par ailleurs, si un port est défini comme étant du type **in** ou **inout**, ce port doit être dans un état de détection, prêt à recevoir du trafic sur la connexion.

Toutes les variables de composant et tous les temporisateurs sont réinitialisés à leur (éventuelle) valeur initiale et toutes les constantes de composant sont réinitialisées à la valeur qui leur avait été affectée lors de la création explicite ou implicite du composant.

On distingue deux types de composant PTC: un PTC qui ne peut exécuter une fonction de comportement qu'une seule fois et un PTC qui est maintenu sous tension (alive) après la fin d'une fonction de comportement et qui peut donc être réutilisé pour exécuter une autre fonction. Le second type de composant PTC est créé au moyen du mot clé supplémentaire **alive**. Un PTC de type sous tension doit être détruit explicitement au moyen de l'opération **kill** (voir § 22.9), tandis qu'un PTC de type hors tension (non-alive) est supprimé implicitement une fois que sa fonction de comportement prend fin. La fin d'un test élémentaire, c'est-à-dire du MTC, met fin à tous les PTC qui existent encore, s'il y en a.

Etant donné que tous les composants et ports sont implicitement supprimés à la terminaison de chaque test élémentaire, celui-ci doit créer entièrement sa configuration requise de composants et de connexions quand il est invoqué.

L'opération **create** doit retourner l'unique référence de composant de l'instance récemment créée. L'unique référence au composant va normalement être mémorisée dans une variable (voir § 8.7). Elle pourra servir à connecter des instances et à effectuer une communication, par exemple une émission ou une réception.

A titre facultatif, un nom peut être associé à l'instance de composant récemment créée. Ce nom doit être une valeur de **charstring** (chaîne de caractères) et, lorsqu'il est affecté, doit apparaître en tant qu'argument de la fonction **create**. Le système de test doit associer le nom "MTC" au composant MTC et le nom "SYSTEM" à l'interface du

système de test automatiquement au moment de la création. Les noms associés aux composants ne doivent pas obligatoirement être uniques.

NOTE – Le nom de l'instance de composant ne doit être utilisé qu'aux fins de la journalisation (voir § 19.3), ne doit pas être utilisé pour faire référence à l'instance de composant (on utilisera à cette fin la référence de composant) et n'a aucun effet sur l'appariement.

EXEMPLE:

```
// Cet exemple déclare des variables de type MyComponentType, qui servent
// à mémoriser
// les références des instances de composant récemment créées de type
// MyComponentType
// qui sont le résulttat des opérations "create". Un nom associé est
// attribué à certaines
// des instances de composant créées.
  :
var MyComponentType MyNewComponent;
var MyComponentType MyNewestComponent;
var MyComponentType MyAliveComponent;
var MyComponentType MyAnotherAliveComponent;
  :
MyNewComponent := MyComponentType.create;
MyNewestComponent := MyComponentType.create("Newest");
MyAliveComponent := MyComponentType.create alive;
MyAnotherAliveComponent := MyComponentType.create("Another Alive") alive;
  :
```

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

22.2 Les opérations "Connect" (connexion) et "Map" (mappage)

22.2.0 Généralités

Les ports d'un composant de test peuvent être connectés à d'autres composants ou aux ports de l'interface du système de test. Dans le cas de connexions entre deux composants de test, l'opération **connect** doit être utilisée. Lors de la connexion d'un composant de test à une interface du système de test, l'opération **map** doit être utilisée. L'opération **connect** relie directement un port à un autre avec le côté **in** connecté au côté **out** et vice versa. L'opération **map** peut par ailleurs être considérée comme étant simplement une conversion de nom définissant la façon dont les flux de communication devraient être désignés.

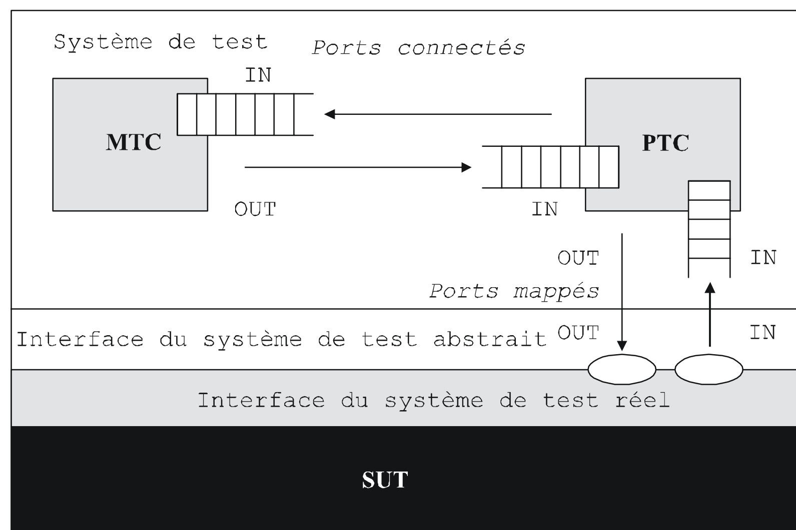


Figure 10/Z.140 – Illustration des opérations "connect" et "map"

Avec les deux opérations **connect** et **map**, les ports à connecter sont identifiés par les références des composants à connecter et par les noms des ports à connecter.

L'opération **mtc** identifie le composant MTC, l'opération **system** identifie l'interface du système de test et l'opération **self** identifie le composant de test dans lequel l'opération **self** a été appelée (voir § 22.4). Ces trois opérations peuvent servir à identifier et à connecter des ports.

Les deux opérations **connect** et **map** peuvent être appelées à partir de chaque définition comportementale sauf pour la partie commande d'un module. Cependant, avant que l'une ou l'autre opération soit appelée, les composants à connecter doivent avoir été créés et leur référence de composant doit être connue en même temps que le nom du port applicable.

Les deux opérations **map** et **connect** permettent la connexion d'un port à plusieurs autres ports. Il n'est pas permis de connecter un composant à un port déjà affecté ni d'affecter un composant à un port déjà connecté.

EXEMPLE:

```
 // L'on part du principe que les ports Port1, Port2, Port3 et PCO1 sont
 // correctement définis et déclarés dans les définitions
 // correspondantes du type de port et du type de composant
 :
var MyComponentType  MyNewPTC;
 :
MyNewPTC := MyComponentType.create;
 :
 :
connect(MyNewPTC:Port1, mtc:Port3);
map(MyNewPTC:Port2, system:PCO1);
 :
 :
 // Dans cet exemple, un nouveau composant de type MyComponentType est créé
 // et sa référence est mémorisée dans le variable MyNewPTC.
 // Ensuite, dans l'opération "connect", le Port1 de ce nouveau composant est
 // connecté au Port3 du composant MTC. Au moyen de l'opération map,
 // le port Port2 du nouveau composant est alors connecté au port PC01 de
 // l'interface du système de test
```

22.2.1 Connexions et mappages cohérents

Dans les deux opérations **connect** et **map**, seules des connexions cohérentes sont autorisées.

Si l'on suppose ce qui suit:

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

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

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

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

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

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

- outlist-PORT1 $\subseteq$ inlist-PORT2 et si outlist-PORT2 $\subseteq$ inlist-PORT1.

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

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

Dans tous les autres cas, les opérations ne sont pas autorisées.

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

22.3 Les opérations "Disconnect" (déconnexion) et "Unmap" (démappage)

Les opérations **disconnect** et **unmap** sont les opérations inverses des opérations **connect** et **map**. Elles effectuent la déconnexion de ports (déjà connectés) de composants de test et le démappage de ports de composants de test et de ports de l'interface du système de test (déjà mappés).

Les deux opérations **disconnect** et **unmap** peuvent être appelées à partir de tout composant si les références applicables de composant ainsi que les noms des ports applicables sont connus. Une opération **disconnect** ou **unmap** n'a d'effet que si la connexion ou le mappage à supprimer a été créée préalablement.

EXEMPLE 1:

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

Pour faciliter les opérations **disconnect** et **unmap** relatives à toutes les connexions et les mappages d'un composant ou d'un port, il est permis d'utiliser lesdites opérations avec un seul argument, lequel spécifie une extrémité des connexions à **disconnect** déconnecter ou à **unmap** démapper. Le mot clé **all port** peut être utilisé pour désigner tous les ports d'un composant.

EXEMPLE 2:

```
  :
disconnect(MyNewComponent:Port1);         // déconnecte toutes les connexions de Port1,
                                          // qui est détenu par le composant MyNewComponent
unmap(MyNewComponent:all port);           // démappe tous les ports du composant MyNewComponent
  :
```

L'utilisation d'une opération **disconnect** ou **unmap** sans paramètres est une forme abrégée de l'utilisation de l'opération avec le paramètre **self:all port**. Elle déconnecte ou démappe tous les ports du composant qui appelle l'opération.

EXEMPLE 3:

```
  :
disconnect;                             // est une forme abrégée pour …
disconnect(self:all port);              // qui déconnecte tous les ports du composant
                                        // composant qui a appelé l'opération
  :
unmap;                                  // est une forme abrégée pour … qui démappe tous les
                                        // ports qui démappe tous les ports du composant qui a
                                        // appelé l'opération
  :
```

Le mot clé **all component** ne doit être utilisé que combiné avec le mot clé **all port**, c'est-à-dire sous la forme **all component:all port**, et ne doit être utilisé que par le composant MTC. De plus, l'argument **all component:all port** doit être utilisé comme le seul et unique argument d'une opération **disconnect** ou **unmap** et il permet de libérer toutes les connexions et tous les mappages de la configuration de test.

EXEMPLE 4:

```
  :
  :
disconnect(all component:all port);      // le composant MTC déconnecte tous les ports de
                                         // tous les composants dans la configuration de test.
  :
  :
unmap(all component:all port);          // le composant MTC démappe tous les ports de tous
                                        // les composants dans la configuration de test.
  :
```

22.4 Les opérations "MTC", "System" et "Self"

La référence de composant (voir § 8.7) comporte trois opérations: **mtc** et **system** qui retournent respectivement la référence du composant de test principal et celle de l'interface du système de test. L'opération **self** peut servir à retourner la référence du composant dans lequel elle est appelée.

EXEMPLE:

```
var MyComponentType MyAddress;
MyAddress := self; // mémoriser la référence actuelle du composant
```

Les seules opérations autorisées sur les références de composant sont l'affectation, l'égalité et l'inégalité.

22.5 L'opération "Start" (lancement de composant de test)

Une fois qu'un composant PTC a été créé et connecté, un comportement doit lui être associé et l'exécution de son comportement doit être lancée. L'on utilise à cette fin l'opération **start** (car la création de composant PTC ne lance pas l'exécution du comportement du composant). La raison de la distinction entre les opérations **create** et **start** est de permettre d'effectuer les opérations de connexion avant d'exécuter réellement le composant de test.

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

Un composant PTC de type sous tension peut exécuter plusieurs fonctions comportementales en ordre séquentiel. Le lancement d'une deuxième fonction comportementale sur un composant PTC hors tension ou le lancement d'une fonction sur un composant PTC qui est encore en cours d'exécution provoque une erreur de test élémentaire. Si une fonction est lancée sur un composant PTC de type sous tension après achèvement d'une fonction précédente, cette fonction utilise les valeurs de variable, les temporisations, les ports et le verdict local en l'état où ils se trouvaient après achèvement de la fonction précédente. En particulier, si une temporisation a été lancée dans la fonction précédente, la fonction subséquente devrait être activée pour traiter un éventuel événement d'expiration de temporisation.

EXEMPLE:

```
function MyFirstBehaviour() runs on MyComponentType { … }
function MySecondBehaviour() runs on MyComponentType { … }
 :
var MyComponentType MyNewPTC;
var MyComponentType MyAlivePTC;
 :
 MyNewPTC := MyComponentType.create;            // Création d'un nouveau composant
                                                // de test hors tension.
 MyAlivePTC := MyComponentType.create alive;    // Création d'un nouveau composant
                                                // de test sous tension
 :
MyNewPTC.start(MyFirstBehaviour());             // Début du nouveau composant hors tension.
MyNewPTC.done;                                  // Attente de terminaison
MyNewPTC.start(MySecondBehaviour());            // Erreur de test élémentaire
 :
MyAlivePTC.start(MyFirstBehaviour());           // Lancement du composant de type sous tension
MyAlivePTC.done;                                // Attente de terminaison
 MyAlivePTC.start(MySecondBehaviour());         // Lancement de la prochaine fonction sur le
                                                // même composant
   :
```

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

- si cette fonction a des paramètres, ceux-ci ne doivent être que des paramètres entrants de type **in**, c'est-à-dire des paramètres par valeur;
- cette fonction doit avoir une définition de type **runs on** faisant référence à un type de composant qui est compatible avec le composant récemment créé (voir § 6.7.3);
- les ports et les temporisations ne doivent pas être transmis vers cette fonction.

NOTE – Etant donné que les ports de type **in** et de type **inout** commencent leur détection quand le composant est créé, il peut y avoir, au moment où ils commencent l'exécution, des messages dans les files entrantes de tels ports, déjà en attente de traitement.

22.6 L'opération comportementale "Stop" (arrêt de composant de test)

L'instruction **stop** permet à un composant de test de mettre fin à l'exécution de son propre comportement de test en cours d'exécution ou à l'exécution du comportement de test en cours sur un autre composant de test. Si un composant n'arrête pas son propre comportement, mais arrête le comportement en cours d'exécution sur un autre composant de test dans le système de test, le composant à arrêter doit être identifié au moyen de sa référence de composant. Un composant peut arrêter son propre comportement au moyen d'une simple instruction **stop** (voir § 19.10) ou en s'adressant à lui-même dans l'opération **stop**, par exemple au moyen de l'opération **self**.

EXEMPLE 1:

NOTE 1 – Alors que les opérations **create**, **start**, **running, done** et **killed** ne peuvent servir qu'à des composants PTC, l'opération **stop** peut également être appliquée au composant MTC.

```
var MyComponentType MyComp := MyComponentType.create;
                                  // Un nouveau composant de test a été créé
MyComp.start(CompBehaviour());    // Le nouveau composant est lancé
  :
```

```
if (date == "1.1.2005") {
  MyComp.stop;                 // Le nouveau composant "MyComp" est arrêté
}

 :
if (a < b ) {
   :
  self.stop;                   // Le composant de test qui est en cours
                               // d'exécution arrête son propre comportement
}
 :
  stop                         // Le composant de test arrête son propre comportement
```

Arrêter un composant de test est le moyen explicite de mettre fin à l'exécution du comportement en cours d'exécution. Un comportement de composant de test prend fin également par l'achèvement de son exécution dès qu'est atteinte la fin du test élémentaire ou de la fonction qui est lancée sur ce composant ou par une instruction **`return`** explicite. Cette terminaison est également appelée arrêt implicite. L'arrêt implicite a les mêmes effets qu'un arrêt explicite, c'est-à-dire que le verdict global est mis à jour avec le verdict local du composant de test arrêté (voir § 25).

Si le composant de test arrêté est le composant MTC, les ressources de tous les composants PTC existants doivent être libérés, les composants PTC doivent être supprimés du système de test et le test élémentaire doit prendre fin (voir § 27.2).

L'arrêt d'un composant de test de type hors-tension (implicitement ou explicitement) doit supprimer celui-ci et toutes les ressources associées à ce composant de test doivent être libérées.

L'arrêt d'un composant de type sous-tension ne doit arrêter que le comportement en cours d'exécution, le composant continuant d'exister et pouvant exécuter un nouveau comportement (lancé sur le composant au moyen de l'opération **`start`**). Le composant doit être laissé dans un état cohérent après l'arrêt de son comportement.

NOTE 2 – Par exemple, si le comportement d'un composant de type sous-tension est arrêté pendant l'affectation d'une nouvelle valeur à une variable déjà associée à ce composant, cette variable doit rester associée à ce composant une fois que celui-ci est arrêté (avec l'ancienne ou la nouvelle valeur). De même, si le composant est arrêté pendant le redémarrage d'une temporisation en cours, la temporisation doit rester dans l'état exécution en cours après la fin du comportement.

Les règles applicables à la terminaison des tests élémentaires et le calcul du verdict de test final sont décrits dans le § 25:

Le mot clé **`all`** ne peut être utilisé que par le composant MTC, afin d'arrêter tous les composants PTC en cours sauf le composant MTC lui-même.

NOTE 3 – Un composant PTC peut arrêter l'exécution du test élémentaire en arrêtant le composant MTC.

EXEMPLE 2:

```
 :
all component.stop      // Le MTC arrête tous les PTC du test élémentaire
                        // mais non lui-même.

 :
```

NOTE 4 – Le mécanisme concret d'arrêt des composants PTC est hors du domaine d'application de la présente Recommandation.

22.7 L'opération "Running" (exécution)

L'opération **`running`** permet l'exécution d'un comportement sur un composant de test afin de vérifier si le comportement fonctionnant sur un composant de test différent s'est terminé. L'opération **`running`** ne peut servir qu'aux composants PTC. Elle retourne la valeur **`true`** pour les composants PTC qui ont été lancés mais qui ne sont pas encore terminés ou arrêtés. Dans les autres cas, elle retourne la valeur **`false`**. L'opération **`running`** est considérée comme étant une expression **`boolean`** et retourne donc une valeur de type **`boolean`** afin d'indiquer si le composant de test spécifié (ou tous les composants de test) sont terminés. Contrairement à l'opération de fin d'exécution **`done`**, l'opération **`running`** peut être utilisée librement dans les expressions de type **`boolean`**.

Quand le mot clé **`all`** est utilisé avec l'opération **`running`**, l'opération retournera la valeur **`true`** si tous les composants PTC, lancés mais non arrêtés explicitement par un autre composant, sont en train d'exécuter leur comportement. Sinon l'opération retourne la valeur **`false`**.

Quand le mot clé **`any`** est utilisé avec l'opération **`running`**, l'opération retournera la valeur **`true`** si au moins un composant PTC est en train d'exécuter son comportement. Sinon l'opération retourne la valeur **`false`**.

EXEMPLE:

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

while (all component.running != true) {   // utilisation de l'opération
                                          // running dans une condition de boucle
  MySpecialFunction()
}
```

22.8 L'opération "done" (fin d'exécution)

L'opération **done** permet d'exécuter un comportement sur un composant de test afin de vérifier si le comportement fonctionnant sur un composant de test différent est terminé. L'opération de fin d'exécution ne peut servir qu'à des composants PTC.

L'opération de fin d'exécution **done** doit être utilisée de la même façon qu'une opération de réception ou qu'une opération de temporisation **timeout**. Autrement dit, elle ne doit pas être utilisée dans une expression **boolean** mais peut servir à déterminer une option dans une instruction **alt** ou à devenir une instruction autonome dans une description de comportement. Dans ce dernier cas, l'opération de fin d'exécution **done** est considérée comme étant un abrégé pour une instruction **alt** n'offrant qu'une seule option, c'est-à-dire que cette opération a une sémantique bloquante et offre donc la capacité d'attente passive de terminaison des composants de test.

Lorsqu'elle est appliquée à un composant PTC, l'opération de fin d'exécution **done** ne s'apparie que si le comportement de ce composant PTC a été arrêté (implicitement ou explicitement) ou si celui-ci a été supprimé. Sinon, l'appariement échoue.

Quand le mot clé **all** est utilisé avec l'opération de fin d'exécution **done**, celle-ci s'apparie si aucun composant PTC n'est en train d'exécuter son comportement. Elle s'apparie également si aucun composant PTC n'a été créé.

Quand le mot clé **any** est utilisé avec l'opération de fin d'exécution **done**, celle-ci s'apparie si au moins le comportement d'un composant PTC a été arrêté ou supprimé. Sinon, l'appariement échoue.

NOTE – L'arrêt du comportement d'un composant hors tension a également pour effet de supprimer ce composant du système de test, alors que l'arrêt d'un composant de type sous tension laisse le composant sous tension dans le système de test. Dans les deux cas, l'opération **done** s'apparie.

EXEMPLE:

```
// Utilisation de l'opération done dans des instructions alt
:
alt {
   []  MyPTC.done {
        setverdict(pass)
       }

   []  any port.receive {
        repeat
       }
}
:

var MyComp c := MyComp.create alive;
c.start(MyPTCBehaviour());
:
c.done;
   // s'apparie dès que la fonction MyPTCBehaviour (ou la
   // fonction/variante appelée par cette
   // fonction s'arrête
c.done;
   // s'apparie également à la fin de la fonction MyPTCBehaviour
   // (ou de la fonction/variante appelée par cette fonction
if(c.running) {c.done}
   // ici la fonction d'exécution ne s'apparie qu'à la fin du prochain
   // comportement
   // l'opération "done" suivante, qui est une instruction autonome:
 :
all component.done;
 :

// a la signification suivante:
 :
```

```
alt {
   []  all component.done {}
}
 :

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

22.9 L'opération "Kill" (suppression de composant de test)

Appliquée à un composant de test, l'opération **kill** arrête l'exécution du comportement en cours – s'il y a lieu – de ce composant et libère toutes les ressources qui lui sont associées (y compris toutes les connexions de port du composant supprimé) et supprime le composant du système de test. L'opération **kill** peut être appliquée au composant de test actuel lui-même par une simple instruction **kill** ou en s'adressant à lui-même au moyen de l'opération **self** conjointement avec l'opération de suppression. L'opération **kill** peut aussi être appliquée à un autre composant de test. Dans ce cas, le composant à supprimer doit être adressé au moyen de sa référence de composant. Si l'opération **kill** est appliquée au composant MTC, par exemple **mtc.kill**, elle met fin au test élémentaire.

EXEMPLE 1:

```
var PTCType MyAliveComp := PTCType.create alive;     // Crée un composant de test de type sous
                                                     // de type sous tension
MyAliveComp.start(MyFirstBehavior());                // Le nouveau composant est lancé
MyAliveComp.done;                                    // Attente de terminaison
MyAliveComp.start(MySecondBehavior());               // Lancer le composant une deuxième fois
MyAliveComp.done;                                    // Attente de terminaison
MyAliveComp.kill;                                    // Libérer ses ressources
```

Le mot clé **all** ne peut être utilisé par le composant MTC que pour arrêter et supprimer tous les composants PTC en cours d'exécution, sauf le composant MTC lui-même.

EXEMPLE 2:

```
all component.kill;          // Le composant MTC commence par arêter tous
                             // les composants PTC (de type
                             // sous tension et normaux) du test élémentaire,
                             // puis libère leurs ressources.
```

22.10 L'opération "Alive" (sous tension)

L'opération **alive** est une opération booléenne qui vérifie si un composant de test a été créé et est prêt à exécuter, ou est déjà en train d'exécuter, une fonction comportementale. Appliquée à un composant de test normal, l'opération **alive** retourne la valeur **true** si le composant est inactif ou exécute une fonction et la valeur false sinon. Appliquée à un composant de test de type sous tension, l'opération retourne la valeur true si le composant est inactif, en cours d'exécution ou arrêté. Elle retourne la valeur false si le composant a été supprimé.

L'opération **alive** ne peut être utilisée de manière similaire à l'opération **running** que sur des composants PTC (voir § 22.7). En particulier, en combinaison avec le mot clé **all**, elle retourne la valeur true si tous les composants PTC (de type sous tension ou normaux) sont sous tension.

Utilisée en combinaison avec le mot clé **any**, l'opération **alive** retourne la valeur true si au moins un composant PTC est sous tension.

EXEMPLE:

```
 :
PTC1.done;                          // Attend la terminaison du composant
if (PTC1.alive) {                   // Si le composant est encore sous tension …
   PTC1.start(AnotherFunction());   // … exécuter une autre fonction sur celui-ci.
}
```

22.11 Opération "Killed" (supprimé)

L'opération **killed** permet de vérifier si un composant de test différent est sous tension ou a été supprimé du système de test.

L'opération **killed** doit être utilisée de la même façon que des opérations de réception. Autrement dit, elle ne doit pas être utilisée dans une expression **boolean** mais peut servir à déterminer une option dans une instruction **alt** ou à devenir une instruction autonome dans une description de comportement. Dans ce dernier cas, l'opération de fin d'exécution **done** est considérée comme étant un abrégé pour une instruction **alt** n'offrant qu'une seule option,

c'est-à-dire que cette opération a une sémantique bloquante et offre donc la capacité d'attente passive de terminaison des composants de test.

NOTE – Lors de la vérification de composants de test normaux, une opération **killed** s'apparie si elle a arrêté (implicitement ou explicitement) l'exécution de son comportement a été supprimée explicitement, autrement dit l'opération est équivalente à l'opération de fin d'exécution **done** (voir § 22.8). Lors de la vérification des composants de test de type sous tension, cependant, l'opération **killed** ne s'apparie que si le composant a été supprimé au moyen de l'opération **kill**. Sinon, l'opération **killed** échoue.

L'opération **killed** ne peut être utilisée que pour des composants PTC.

Quand le mot clé **all** est utilisé avec l'opération **killed**, celle-ci s'apparie si tous les composants PTC du test élémentaire ont cessé d'exister. Il s'apparie également si aucun composant PTC n'a été créé.

Quand le mot clé **any** est utilisé avec l'opération **killed**, celle-ci s'apparie si au moins un composant PTC a cessé d'exister. Sinon, l'appariement échoue.

EXEMPLE:

```
var MyPTCType ptc := MyPTCType.create alive;   // créer un composant de
                                               // test de type sous tension
timer T(10.0);                                 // créer une temporisation
T.start;                                       // lancer la temporisation
ptc.start(MyTestBehavior());                   // lancer l'exécution d'une fonction
                                               // sur le composant PTC
alt {
[] ptc.killed {                                // si le composant PTC a été supprimé
                                               // pendant l'exécution …

  T.stop;                                      // … arrêter la temporisation et …
  setverdict(inconc);                          // … mettre le verdict à la valeur
                                               // "inconclusive" (non concluant)

   }
[] ptc.done {                                  // si le composant PTC a pris fin
                                               // régulièrement …
  T.stop;                                      // arrêter la temporisation et …
  ptc.start(AnotherFunction());                // lancer une autre fonction sur le
                                               // composant PTC
[] T.timeout {                                 // si l'expiration de la temporisation
                                               // se produit avant que le composant PTC
                                               // se soit arrêté
  ptc.kill;                                    // … mettre le verdict à …
  setverdict(fail);                            // … la valeur "fail"(échec)
   }
}
```

22.12 Utilisation de matrices de composants

Les opérations **create**, **connect**, **start**, **stop** et **kill** ne s'appliquent pas directement à des matrices de composants. En revanche, un élément spécifique de la matrice doit être fourni comme paramètre pour ces opérations. Dans les composants, l'effet d'une matrice est obtenu au moyen d'une matrice de références de composant et par l'attribution de l'élément matriciel approprié au résultat de l'opération **create**.

EXEMPLE:

```
// Cet exemple montre la façon de modéliser l'effet de la création, de la
// connexion et de l'exécution de matrices de composants au moyen d'une
// boucle et au moyen de la mémorisation de la référence de composant créée
// dans une matrice de références de composant.

testcase MyTestCase() runs on MyMtcType system MyTestSystemInterface
{
   :
  var integer i;
  var MyPTCType1 MyPtc[11];
   :
  for (i:= 0; i<=10; i:=i+1)
  {
     MyPtc[i] := MyPTCType1.create;
     connect(self:PtcCoordination, MyPtc[i]:MtcCoordination);
     MyPtc[i].start(MyPtcBehaviour());
  }
   :
}
```

22.13 Résumé de l'utilisation de "any" et "all" avec des composants

Les mots clés `any` et `all` peuvent être utilisés avec des opérations de configuration comme indiqué dans le Tableau 16.

Tableau 16/Z.140 – Utilisation des mots clés "Any" et "All" avec des composants

Opération	Utilisation permise		Exemple	Commentaire
	`any` (voir Note)	`all` (voir Note)		
`create`				
`start`				
`running`	Oui mais à partir de MTC seulement	Oui mais à partir de MTC seulement	`any component.running;` `all component.running;`	Un composant PTC est-il en train d'exécuter le comportement de test? Tous les composants PTC sont-ils en train d'exécuter le comportement de test?
`alive`	Oui mais à partir de MTC seulement	Oui mais à partir de MTC seulement	`any component.alive;` `all component.alive;`	Y a-t-il un composant PTC sous tension? Tous les composants PTC sont-ils sous tension?
`done`	Oui mais à partir de MTC seulement	Oui mais à partir de MTC seulement	`any component.done;` `all component.done;`	Un composant PTC a-t-il terminé l'exécution? Tous les composants PTC ont-ils terminé leur exécution?
`killed`	Oui mais à partir de MTC seulement	Oui mais à partir de MTC seulement	`any component.killed;` `all component.killed;`	Un composant PTC a-t-il cessé d'exister? Tous les composants PTC ont-ils cessés d'exister?
`stop`		Oui mais à partir de MTC seulement	`all component.stop;`	Arrêter le comportement sur tous les composants PTC.
`kill`		Oui mais à partir de MTC seulement	`all component.kill;`	Supprimer tous les composants PTC qui, autrement dit, cessent d'exister.
NOTE – Les mots clés `any` et `all` ne font référence qu'aux composants PTC, autrement dit le composant MTC n'est pas pris en considération.				

23 Opérations de communication

23.0 Généralités

La notation TTCN-3 prend en charge les communications *en mode message* et *en mode procédure*. Par ailleurs, la notation TTCN-3 permet d'examiner l'élément sommital des files d'attente d'entrée dans un port et de commander l'arrivée à des ports au moyen d'*opérations de commande*.

Tableau 17/Z.140 – Aperçu général des opérations de communication TTCN-3

Opérations de communication			
Opération de communication	**Mot clé**	**Peut servir à un port en mode message**	**Peut servir à un port en mode procédure**
Communication en mode message			
Envoi de message	`send`	Oui	
Réception de message	`receive`	Oui	
Déclenchement sur message	`trigger`	Oui	
Communication en mode procédure			
Invocation d'appel de procédure	`call`		Oui
Acceptation d'appel de procédure à partir d'entité distante	`getcall`		Oui
Réponse à appel de procédure à partir d'entité distante	`reply`		Oui
Propagation d'une exception (vers un appel accepté)	`raise`		Oui

Tableau 17/Z.140 – Aperçu général des opérations de communication TTCN-3

Opérations de communication			
Opération de communication	**Mot clé**	**Peut servir à un port en mode message**	**Peut servir à un port en mode procédure**
Traitement de la réponse à partir d'un appel précédent	`getreply`		Oui
Acquisition d'une exception (issue de l'entité appelée)	`catch`		Oui
Examen de l'élément sommital de files d'attente d'entrée dans un port			
Vérification de message/appel/exception/réponse reçu	`check`	Oui	Oui
Opérations de commande			
Suppression de file d'attente à un port	`clear`	Oui	Oui
Suppression de file d'attente et activation de l'émission et de la réception à un port	`start`	Oui	Oui
Désactivation de l'émission et interdiction des opérations de réception à apparier à un port	`stop`	Oui	Oui
Désactivation de l'émission et interdiction des opérations de réception visant à apparier de nouveaux messages/appels	`halt`	Oui	Oui

23.1 Format général des opérations de communication

23.1.0 Généralités

Les opérations de type tel que **`send`** et **`call`** servent à l'échange d'informations entre les composants de test et entre un système SUT et les composants de test. Pour expliquer le format général de ces opérations, celles-ci peuvent être structurées en deux groupes:

a) un composant de test envoie un message (opération **`send`**), appelle une procédure (opération **`call`**), ou répond à un appel accepté (opération **`reply`**) ou propage une exception (opération **`raise`**). Ces actions sont collectivement désignées par le terme *opération d'envoi*;

b) un composant reçoit un message (opération **`receive`**), attend un message (opération **`trigger`**), accepte un appel de procédure (opération **`getcall`**), reçoit une réponse à une procédure déjà appelée (opération **`getreply`**) ou acquiert une exception (opération **`catch`**). Ces actions sont collectivement désignées par le terme *opérations de réception*.

23.1.1 Format général de l'opération d'envoi

L'opération d'envoi se compose d'une partie *envoi* et, dans le cas d'une opération d'appel **`call`** par communication bloquante en mode procédure, d'une partie *réponse* et d'une partie *manipulation d'exception*.

La partie envoi:

- spécifie le port auquel l'opération spécifiée doit avoir lieu;
- définit l'appel en mode message ou en mode procédure à transmettre;
- donne une partie adresse (facultative) qui identifie de façon univoque un ou plusieurs correspondants de la communication auquel(s) un message, un appel, une réponse ou une exception doit être envoyé.

Le nom du port, le nom de l'opération et la valeur doivent être présents dans toutes les opérations d'envoi. La partie adresse (indiquée par le mot clé **`to`**) est facultative et n'a besoin d'être spécifiée qu'en cas de connexions point-multipoint où:

- la communication est établie en mode monodiffusion et une entité réceptrice doit être explicitement identifiée;
- la communication est établie en mode multidiffusion et un ensemble d'entités réceptrices doit être explicitement identifié;
- la communication est établie en mode diffusion et toutes les entités connectées au port spécifié doivent être adressés.

NOTE – L'utilisation des termes mode monodiffusion, mode multidiffusion et mode diffusion se rapporte à la communication entre ports. Autrement dit, il n'est possible d'adresser qu'un seul, plusieurs ou tous les composants de test qui sont connectés au port spécifié. Les modes monodiffusion, multidiffusion et diffusion peuvent aussi être utilisés pour des ports mappés. Dans ce cas, une seule, plusieurs ou toutes les entités faisant partie du système SUT peuvent être atteintes via le port mappé spécifié.

EXEMPLE 1:

Partie envoi			**Partie (facultative) réponse et traitement d'exceptions**
Accès et opération	**Partie valeur**	**Partie adresse (facultative)**	
`MyP1.send`	`(MyVariable + YourVariable - 2)`	`to MyPartner;`	

La partie réponse et traitement d'exceptions n'est requise qu'en cas de communication en mode procédure. La partie réponse et traitement d'exception de l'opération d'appel **call** est facultative et est requise dans les cas où la procédure appelée retourne une valeur ou a des paramètres **out** ou **inout** dont les valeurs sont requises dans le composant appelant et dans les cas où la procédure appelée peut déclencher des exceptions qui nécessitent d'être traités par le composant appelant.

La partie réponse et traitement d'exceptions de l'opération d'appel fait usage des opérations **getreply** et **catch** afin d'offrir la fonctionnalité requise.

EXEMPLE 2:

Partie envoi			**Partie (facultative) réponse et traitement d'exceptions**
Accès et opération	**Partie valeur**	**Partie adresse (facultative)**	
`MyP1.call`	`(MyProc:{MyVar1})`		`{ [] MyP1.getreply(MyProc:{MyVar2}) {} [] MyP1.catch(MyProc, ExceptionOne) {} }`

23.1.2 Format général des opérations de réception

Une opération de réception se compose d'une partie *réception* et d'une partie (facultative) *affectation*.

La partie réception:

a) spécifie le port auquel l'opération doit avoir lieu;

b) définit une partie correspondante qui spécifie l'entrée acceptable qui va correspondre à l'instruction;

c) donne une expression d'adressage (facultative) qui identifie de façon univoque le correspondant de la communication (en cas de connexions point-multipoint).

Le nom du port, le nom de l'opération et la partie valeur de toutes les opérations de réception doivent être présents. L'identification du correspondant de la communication (indiqué par le mot clé **from**) est facultative et n'a besoin d'être spécifiée qu'en cas de connexions point-multipoint où l'entité réceptrice a besoin d'être explicitement identifiée.

La partie affectation dans une opération de réception est facultative. Aux ports en mode message, elle est utilisée quand il est nécessaire de mémoriser les messages reçus. Dans le cas des ports en mode procédure, elle est utilisée pour la mémorisation des paramètres **in** et **inout** d'un appel accepté, pour la mémorisation de la valeur de retour ou pour la mémorisation d'exceptions. Pour la partie affectation, un typage fort est requis: par exemple, la variable utilisée pour mémoriser un message doit avoir le même type que le message entrant.

En outre, la partie affectation peut également servir à affecter à une variable l'adresse de l'expéditeur **sender** d'un message, une exception, une réponse de type **reply** ou un appel **call**. Cela est utile pour les connexions point-multipoint où, par exemple, le même message ou appel peut être reçu de différents composants. Mais le message, la réponse ou l'exception doit impérativement être renvoyé au composant d'envoi original.

EXEMPLE:

Partie réception				Partie affectation (facultative)		
Port et opération	**Partie correspondante**	**Expression d'adressage (facultative)**		**Affectation de valeur (facultative)**	**Affectation de valeur paramétrique (facultative)**	**Affectation de valeur d'expéditeur (facultative)**
`MyP1.getreply`	`(AProc:{?} value 5)`		`->`		`param (V1)`	`sender APeer`

Partie réception				Partie affectation (facultative)		
Port et opération	**Partie correspondante**	**Expression d'adressage (facultative)**		**Affectation de valeur (facultative)**	**Affectation de valeur paramétrique (facultative)**	**Affectation de valeur d'expéditeur (facultative)**
`MyP2.receive`	`(MyTemplate(5,7))`	`from APeer`	`->`	`value MyVar`		

23.2 Communication en mode message

23.2.0 Généralités

La communication en mode message est fondée sur un échange asynchrone de messages. La communication en mode message n'est pas bloquante dans l'opération **`send`**, comme illustré dans la Figure 11, où le traitement dans la partie EXPEDITEUR continue immédiatement après l'exécution de l'opération **`send`**. La partie RECEPTEUR est bloquée sur l'opération **`receive`** jusqu'à ce qu'elle traite le message reçu.

En plus de l'opération **`receive`**, la notation TTCN-3 fournit une opération **`trigger`** qui filtre les messages répondant à certains critères de correspondance extraits d'un flux de messages reçus à un certain port entrant. Les messages situés au sommet de la file qui ne satisfont pas aux critères d'appariement sont supprimés du port sans aucune action complémentaire.

Figure 11/Z.140 – Illustration de l'envoi et de la réception en mode asynchrone

23.2.1 L'opération "Send" (envoi)

23.2.1.0 Généralités

L'opération **`send`** sert à placer un message sur un port sortant. Ce message peut être spécifié au moyen d'une référence à un modèle défini ou peut être défini sous la forme d'un modèle en ligne. Lors de la définition du message en ligne, la partie type facultative doit être utilisée s'il y a ambiguïté quant au type du message en cours d'expédition.

L'opération **`send`** ne doit être utilisée qu'à des ports en mode message (ou mixte) et le type du modèle à envoyer doit figurer dans la liste des types sortants de la définition du type de port.

EXEMPLE:

```
MyPort.send(MyTemplate(5,MyVar));  // Envoie le modèle MyTemplate avec les
                                   // paramètres effectifs 5 et MyVar via MyPort.

MyPort.send(5);                    // Envoie la valeur d'entier 5 (qui est un modèle en ligne)
```

23.2.1.1 Emission en mode monodiffusion, multidiffusion ou diffusion

La notation TTCN-3 prend en charge la communication en mode monodiffusion, multidiffusion et diffusion. Le mécanisme de communication utilisé peut être déterminé par la clause **`to`** facultative dans l'opération **`send`**. Une clause **`to`** peut être omise dans le cas d'une connexion point à point où la communication en mode monodiffusion est utilisée

et où le récepteur du message est déterminé de façon univoque par la structure du système de test. Une clause **to** doit être présente dans le cas de connexions point-multipoint.

La communication en mode monodiffusion est spécifiée, si la clause **to** n'adresse qu'un seul correspondant de communication. La communication en mode multidiffusion est utilisée, si la clause **to** inclut une liste de correspondants de la communication. Le mode diffusion est défini par l'utilisation de la clause **to** avec le mot clé **all component**.

EXEMPLE:

```
MyPort.send(charstring:"My string") to MyPartner;
                    // Envoie la chaîne "My string" à un
                    // composant ayant une
                    // référence de composant mémorisée
                    // dans la variable MyPartner

MyPCO.send(MyVariable + YourVariable - 2) to MyPartner;
                    // Envoie le résultat de l'expression
                    // arithmétique à MyPartner.

MyPCO2.send(MyTemplate) to (MyPeerOne, MyPeerTwo);
                    // Spécifie une communication en mode
                    // multidiffusion, où la
                    // valeur de MyTemplate est envoyée
                    // aux deux références de
                    // composant mémorisées dans les
                    // variables MyPeerOne et MyPeerTwo.

MyPCO3.send(MyTemplate) to all component;
                    // Communication en mode diffusion:
                    // la valeur de Mytemplate est
                    // envoyée à tous les composants qui
                    // peuvent être adressés via
                    // ce port. SiMyPCO3 est un port
                    // mappé, les composants peuvent
                    // résider dans le système SUT.
```

23.2.2 L'opération "Receive" (réception)

23.2.2.0 Généralités

L'opération **receive** sert à recevoir un message à partir d'une file d'attente de messages entrant dans un port. Ce message peut être spécifié au moyen d'une référence à un modèle défini ou peut être défini sous la forme d'un modèle en ligne. Lors de la définition du message en ligne, la partie de type facultative doit être présente chaque fois que le type du message reçu est ambigu. L'opération **receive** ne doit être utilisée qu'à des ports en mode message (ou mixte) et le type de la valeur à recevoir doit être inclus dans la liste des types entrants de la définition du type de port.

L'opération **receive** supprime le message sommital de la file d'attente de messages entrants associée à un port si et seulement si ce message sommital satisfait tous les critères de correspondance associés à l'opération **receive**. Aucune association des valeurs entrantes aux termes de l'expression ou au modèle ne doit se produire.

Si la mise en correspondance n'est pas efficace, le message sommital ne doit pas être supprimé de la file d'attente à un port, c'est-à-dire que si l'opération **receive** est utilisée comme option d'une instruction **alt** et qu'elle ne soit pas efficace, l'exécution du test élémentaire doit se poursuivre par la prochaine option de l'instruction **alt**.

Les critères de correspondance sont associés au type et à la valeur du message à recevoir. Le type et la valeur du message à recevoir sont déterminés par l'argument de l'opération **receive**, c'est-à-dire qu'ils peuvent être issus du modèle défini ou être spécifiés en ligne. Un champ de type facultatif doit être utilisé dans les critères de correspondance avec l'opération **receive** afin d'éviter toute ambiguïté au sujet du type de la valeur qui est reçue.

NOTE 1 – Des attributs de codage participent également, de façon implicite, à la mise en correspondance en empêchant le décodeur de produire une valeur abstraite à partir d'un message reçu mais codé autrement que spécifié par ces attributs.

Dans le cas de connexions point-multipoint, l'opération **receive** peut être limitée à un certain correspondant de la communication. Cette restriction doit être indiquée au moyen du mot clé **from**.

EXEMPLE 1:

```
MyPort.receive(MyTemplate(5, MyVar));   // Correspond à un message qui satisfait les
                                        // conditions définies par le modèle MyTemplate
                                        // au port MyPort.
```

```
MyPort.receive(A<B);                     // Correspond à une valeur boléenne qui dépend du
                                         // résultat de A<B

MyPort.receive(integer:MyVar);           // Correspond à une valeur d'entier avec
                                         // la valeur de MyVar au port MyPort

MyPort.receive(MyVar);                   // est une variante de l'exemple précédent

MyPort.receive(charstring:"Hello")from MyPeer; // Correspond à la chaîne "Hello" from MyPeer
```

Si la mise en correspondance est efficace, la valeur supprimée de la file d'attente à un port peut être extraite et mémorisée dans une variable. Cela est indiqué par le symbole '->' et par le mot clé **value**.

Il est également possible d'extraire et de mémoriser la référence de composant ou l'adresse de l'expéditeur d'un message. Cela est indiqué par le mot clé **sender**.

NOTE 2 – Quand le message est reçu sur un port connecté, seule la référence de composant est mémorisée dans le mot clé **sender** suivant, mais le système de test doit aussi enregistrer dans sa mémoire interne le nom de composant, s'il est présent (à utiliser dans la journalisation).

EXEMPLE 2:

```
MyPort.receive(MyType:?) -> value MyVar;       // La valeur du message reçu est
                                               // affectée à MyVar.

MyPort.receive(A<B) -> sender MyPeer;          // L'adresse de l'expéditeur est
                                               // affectée à MyPeer

MyPort.receive(MyTemplate:{5, MyVarOne}) -> value MyVarTwo sender MyPeer;
// La valeur du message reçu est mémorisée dans MyVarTwo et l'adresse de
// l'expéditeur est mémorisée dans MyPeer.
```

23.2.2.1 Réception de message quelconque

Une opération **receive** sans liste d'arguments pour les critères de correspondance du type et de la valeur du message à recevoir doit supprimer le message situé au sommet de la file (éventuelle) d'attente d'entrée à un port si tous les autres critères de correspondance sont satisfaits.

Un message reçu par l'opération *ReceiveAnyMessage* ne doit pas être affecté à une variable.

EXEMPLE:

```
MyPort.receive;                     // Supprime la valeur sommitale de MyPort.

MyPort.receive from MyPeer;         // Supprime le message sommital de MyPort si son
                                    // expéditeur est MyPeer

MyPort.receive -> sender MySenderVar;  // Supprime le message sommital de
                                    // MyPort et affecte l'adresse de
                                    // l'expéditeur à MySenderVar
```

23.2.2.2 Réception à un port quelconque

Afin de recevoir **receive** un message à tout port, il convient d'utiliser les mots clés **any port**.

EXEMPLE:

```
any port.receive(MyMessage);
```

23.2.3 L'opération "Trigger" (déclenchement)

23.2.3.0 Généralités

L'opération **trigger** supprime le message sommital de la file d'attente de messages entrants associée à un port. Si ce message sommital satisfait les critères de correspondance, l'opération **trigger** se comporte de la même façon qu'une opération **receive**. Si ce message sommital ne satisfait pas les critères de correspondance, il doit être supprimé de la file sans aucune action complémentaire. L'opération **trigger** ne doit être utilisée qu'aux ports en mode message (ou mixte) et le type de la valeur à recevoir doit être inclus dans la liste des types entrants de la définition du type de port.

NOTE – La Note 1 du § 22.2.2.0 est également valide pour l'opération **trigger**.

L'opération **trigger** opération peut être utilisée comme instruction autonome dans une description de comportement. Dans ce dernier cas, l'opération **trigger** est considérée comme un abrégé d'une instruction **alt** n'offrant qu'une seule option, c'est-à-dire qu'elle a une sémantique bloquante et permet d'attendre le prochain message correspondant au modèle spécifié ou à la valeur spécifiée dans cette file d'attente.

EXEMPLE 1:

```
MyPort.trigger(MyType:?);
// Spécifie que l'opération va se déclencher dès réception du premier
// message observé du type MyType avec une valeur arbitraire au port
// MyPort.
```

L'opération **trigger** opération exige le nom du port, les critères de correspondance de type et de valeur, une restriction facultative **from** (c'est-à-dire une sélection de correspondant de la communication) et une affectation facultative à des variables du composant correspondant de message et d'expéditeur.

EXEMPLE 2:

```
MyPort.trigger(MyType:?) from MyPartner;
// Se déclenche dès réception du premier message de type MyType au port
// MyPort, reçu de MyPartner.

MyPort.trigger(MyType:?) from MyPartner -> value MyRecMessage;
// Cet exemple est presque identique à l'exemple précédent. En outre, le
// message qui déclenche, c'est-à-dire que tous les critères de
// correspondance sont réunis, est mémorisé dans la variable MyRecMessage.

MyPort.trigger(MyType:?) -> sender MyPartner;
// Cet exemple est presque identique à l'exemple précédent. En outre, la
// référence du composant expéditeur sera extraite et mémorisée dans la
// variable MyPartner.

MyPort.trigger(integer:?) -> value MyVar sender MyPartner;
// Déclenchement dès réception d'une valeur arbitraire d'entier qui ensuite
// est mémorisée dans la variable MyVar. La référence du composant
// expéditeur sera mémorisée dans la variable MyPartner.
```

23.2.3.1 Déclenchement lors d'un message quelconque

Une opération **trigger** sans liste d'arguments doit se déclencher dès réception d'un message quelconque. Sa signification est donc identique à celle de la réception d'un message quelconque. Un message reçu par l'opération *TriggerOnAnyMessage* ne doit pas être affecté à une variable.

EXEMPLE:

```
MyPort.trigger;

MyPort.trigger from MyPartner;

MyPort.trigger -> sender MySenderVar;
```

23.2.3.2 Déclenchement à un port quelconque

Les mots clés **any port** sont utilisés afin d'effectuer une opération **trigger** sur un message à un port quelconque.

EXEMPLE:

```
any port.trigger
```

23.3 Communication en mode procédure

23.3.0 Généralités

Le principe de la communication en mode procédure consiste à appeler des procédures dans des entités distantes. La notation TTCN-3 prend en charge les communications *bloquantes* et *non bloquantes* en mode procédure. Les communications bloquantes en mode procédure sont bloquantes du côté appelant comme du côté appelé, tandis que les communications non bloquantes en mode procédure ne sont bloquantes que du côté appelé. Les signatures de procédure qui servent aux communications non bloquantes en mode procédure doivent être spécifiées conformément aux règles indiquées dans le § 13.

Le schéma de principe des communications bloquantes en mode procédure est représenté dans la Figure 12. L'APPELANT appelle une procédure distante dans l'entité APPELE au moyen de l'opération **call**. L'APPELE accepte l'appel au moyen d'une opération **getcall** et y réagit soit par une opération de type **reply** afin de répondre à l'appel ou par une propagation (opération **raise**) d'exception. L'APPELANT traite la réponse ou l'exception au moyen de l'opération **getreply** ou **catch**. Dans la Figure 12, le blocage des entités APPELANT et APPELE est indiqué au moyen de lignes pointillées.

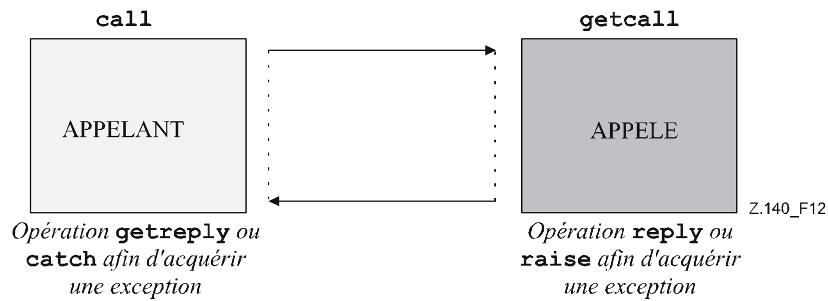


Figure 12/Z.140 – Illustration des communications bloquantes en mode procédure

Le principe des communications non bloquantes en mode procédure est représenté dans la Figure 13. L'APPELANT appelle une procédure distante dans l'entité APPELE au moyen de l'opération d'appel **call** et continue son exécution, c'est-à-dire qu'il n'attend ni réponse ni exception. L'APPELE accepte l'appel au moyen d'une opération **getcall** et exécute la procédure demandée. Si l'exécution n'est pas efficace, l'APPELE peut déclencher une exception afin d'informer l'APPELANT. Celui-ci peut traiter l'exception au moyen d'une opération **catch** insérée dans une instruction **alt**. dans la Figure 13, le blocage de l'APPELE jusqu'à la fin du traitement d'appel et déclenchement possible d'une exception sont indiqués au moyen d'une ligne pointillée.

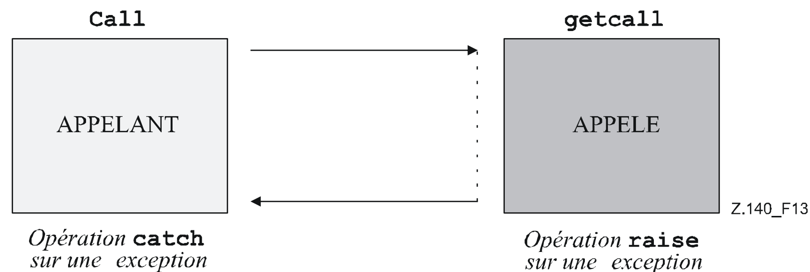


Figure 13/Z.140 – Illustration des communications non bloquantes en mode procédure

23.3.1 L'opération "Call" (appel)

23.3.1.0 Généralités

L'opération d'appel **call** sert à spécifier qu'un composant de test appelle une procédure dans le système SUT ou dans un autre composant de test. L'opération d'appel **call** ne doit être utilisée qu'aux ports en mode procédure (ou mixte). La définition de type du port auquel l'opération d'appel a lieu doit comprendre le nom de la procédure dans sa liste de types **out** ou **inout**; c'est-à-dire que le composant doit impérativement être autorisé à appeler cette procédure à ce port.

Les informations à transmettre dans la partie envoi de l'opération d'appel **call** est une signature qui peut soit être définie sous la forme d'un modèle de signature ou être définie en ligne. Tous les paramètres **in** et **inout** de la signature doivent avoir une valeur spécifique; c'est-à-dire que l'utilisation de mécanismes d'appariement comme *AnyValue* n'est pas autorisée.

Les arguments de signature de l'opération d'appel **call** ne sont pas utilisés afin d'extraire des noms de variable pour les paramètres **out** et **inout**. L'affectation effective à des variables de la valeur de retour de procédure et des valeurs des paramètres **out** et **inout** doit être explicitement effectuée dans la partie réponse et traitement d'exception de l'opération d'appel **call**, au moyen des opérations **getreply** et **catch**. Cela permet d'utiliser des modèles de signature dans des opérations d'appel **call** de la même façon que des modèles peuvent être utilisés pour des types.

EXEMPLE 1:

```
// Si l'on a …
signature MyProc (out integer MyPar1, inout boolean MyPar2);
 :
// un appel de MyProc
MyPort.call(MyProc:{ -, MyVar2}) {       // modèle de signature en ligne pour
                                         // pour l'appel de MyProc
```

```
   [] MyPort.getreply(MyProc:{?, ?}) { }
}

// … et un autre appel de MyProc
MyPort.call(MyProcTemplate) {           // utilisant le modèle de signature
                                        // pour l'appel de MyProc
   [] MyPort.getreply(MyProc:{?, ?}) { }
}
```

Dans le cas de connexions point-multipoint, le correspondant de la communication doit être spécifié de façon unique. Cela doit être indiqué au moyen du mot clé **`to`**.

EXEMPLE 2:

```
MyPort.call(MyProcTemplate) to MyPeer {     // Appel de MyProc à MyPeer
   [] MyPort.getreply(MyProc:{?, ?}) { }
}
```

23.3.1.1 Manipulation des réponses et exceptions à un appel

En cas de communications non bloquantes en mode procédure (voir § 23.3.1.4), le traitement des exceptions à des opérations d'appel **`call`** est assurée au moyen des opérations **`catch`** (voir § 23.3.6) sous forme d'options dans des instructions **`alt`**.

Si l'option **`nowait`** option est choisie (voir § 23.3.1.2), le traitement des réponses ou exceptions à des opérations d'appel **`call`** est assurée au moyen des opérations **`getreply`** (voir § 23.3.4) et **`catch`** (voir § 23.3.6) sous forme d'options dans des instructions **`alt`**.

En cas de communications bloquantes en mode procédure, le traitement des réponses ou exceptions à un appel est assurée dans la partie réponse et traitement d'exception de l'opération d'appel **`call`**, au moyen des opérations **`getreply`** (voir § 23.3.4) et **`catch`** (voir § 23.3.6).

La partie réponse et traitement d'exception d'une opération d'appel **`call`** ressemble au corps d'une instruction de type **`alt`**. Elle définit un ensemble d'options décrivant les réponses et exceptions possibles à l'appel. La sélection des options ne doit être fondée que sur les opérations **`getreply`** et **`catch`** pour la procédure appelée. Les opérations **`getreply`** et **`catch`** non qualifiées ne doivent traiter que les réponses à la procédure appelée et les exceptions propagées par celle-ci. L'utilisation de branches d'échappement de type **`else`** et l'invocation de variantes ne sont pas autorisées.

Si nécessaire, il est possible d'activer/de désactiver une option au moyen d'une expression de type **`boolean`** placée entre les '[ ]' crochets de l'option.

La partie réponse et traitement d'exception d'une opération d'appel est exécutée comme une instruction **`alt`**, sans aucune valeur par défaut active. Autrement dit, un instantané correspondant comprend toutes les informations nécessaires afin d'évaluer les sentinelles booléennes (facultatives), peut inclure l'élément sommital (éventuel) du port auquel la procédure a été appelée et peut inclure une exception d'expiration de temporisation produite par la temporisation (facultative) qui supervise l'appel (voir § 23.3.1.2).

L'évaluation des expressions booléennes gardant les options offertes dans la partie réponse et traitement d'exception peut avoir des effets secondaires. Afin d'éviter des effets secondaires inattendus, les mêmes règles que pour les sentinelles booléennes contenues dans les instructions **`alt`** doivent être appliquées (voir § 20.1.1).

EXEMPLE:

```
// Si l'on a
signature MyProc3 (out integer MyPar1, inout boolean MyPar2) return MyResultType
   exception (ExceptionTypeOne, ExceptionTypeTwo);
  :

// appel de MyProc3
MyPort.call(MyProc3:{ -, true }) to MyPartner {

  [] MyPort.getreply(MyProc3:{?, ?}) -> value MyResult param (MyPar1Var,MyPar2Var) { }

  [] MyPort.catch(MyProc3, MyExceptionOne) {
      setverdict(fail);
      stop;
     }
  [] MyPort.catch(MyProc3, ExceptionTypeTwo : ?) {
      setverdict(inconc);
     }
  [MyCondition] MyPort.catch(MyProc3, MyExceptionThree) { }
}
```

23.3.1.2 Traitement des exceptions d'expiration de temporisation d'appel

L'opération d'appel **call** peut facultativement comprendre une temporisation, laquelle est définie comme une valeur explicite ou comme une constante de type **float** qui indique l'intervalle de temps qui s'écoule après le début de l'opération d'appel **call** jusqu'à ce qu'une exception **timeout** soit produite par le système de test. Si aucune partie contenant une valeur de temporisation n'est présente dans l'opération d'appel **call**, aucune exception d'expiration de temporisation **timeout** ne doit être produite.

EXEMPLE 1:

```
MyPort.call(MyProc:{5,MyVar}, 20E-3) {

  [] MyPort.getreply(MyProc:{?, ?}) { }

  [] MyPort.catch(timeout) {           // exception de fin de temporisation après 20 ms
      setverdict(fail);
      stop;
     }
}
```

L'utilisation du mot clé **nowait** au lieu d'une valeur d'exception d'expiration de temporisation située dans une opération d'appel **call** permet à un appel de procédure de continuer sans attendre de réponse ou d'exception propagée par la procédure appelée, ou une exception d'expiration de temporisation.

EXEMPLE 2:

```
MyPort.call(MyProc:{5, MyVar}, nowait);  // Le composant de test d'appel va continuer
                                         // son exécution sans attendre la terminaison
                                         // de MyProc
```

Si le mot clé **nowait** est utilisé, une éventuelle réponse ou exception à la procédure appelée doit être supprimée de la file d'attente à un port au moyen d'une opération **getreply** ou **catch** dans une instruction **alt** subséquente.

23.3.1.3 Appel de procédures bloquantes sans valeur de retour, sans paramètres de type "out", sans paramètres "inout" et sans exceptions

Une procédure bloquante ne peut avoir aucune valeur de retour, aucun paramètre de type **out** ou **inout** et ne peut déclencher aucune exception. L'opération d'appel pour de tels cas de procédure doit également avoir une partie réponse et traitement d'exception afin de traiter le blocage de façon uniforme.

EXEMPLE:

```
// Si l'on a …
signature MyBlockingProc (in integer MyPar1, in boolean MyPar2);
 :
// un appel de MyBlockingProc
MyPort.call(MyBlockingProc:{ 7, false }) {
  [] MyPort.getreply( MyBlockingProc:{ -, - } ) { }
}
```

23.3.1.4 Appel de procédures non bloquantes

Une procédure non bloquante n'a aucun paramètre de type **out** ou **inout**, aucune valeur de retour et la propriété non bloquante est indiquée dans la définition de signature correspondante au moyen du mot clé **noblock**.

L'opération d'appel **call** pour une procédure non bloquante ne doit avoir aucune partie réponse et traitement d'exception, ne doit déclencher aucune exception d'expiration de temporisation et ne doit pas utiliser le mot clé **nowait**.

Les éventuelles exceptions propagées par procédure non bloquante doivent être supprimées de la file d'attente à un port au moyen d'opérations de type **catch** dans des instructions **alt** ou **interleave** subséquentes.

23.3.1.5 Appels de procédures en mode monodiffusion, multidiffusion et diffusion

Comme pour l'opération **send**, la notation TTCN-3 prend aussi en charge les appels de procédures en mode monodiffusion, multidiffusion et diffusion. Cette prise en charge peut être assurée comme indiquée dans le § 23.2.1.1, à savoir que l'argument de la clause **to** d'une opération **call** est, pour les appels en mode monodiffusion, l'adresse d'une entité réceptrice (ou peut être omis dans le cas de connexions point à point), pour les appels en mode multidiffusion une liste d'adresses d'un ensemble de récepteurs et, pour les appels en mode diffusion, le mot clé **all component**. Dans le cas de connexions point à point, la clause **to** peut être omise, car l'entité réceptrice est identifiée de façon univoque par la structure du système.

Le traitement des réponses et exceptions à une opération d'appel **call** bloquante ou non bloquante en mode monodiffusion a été exposée aux § 23.3.1.1 et 23.3.1.4. Une opération d'appel **call** en mode multidiffusion ou diffusion peut donner lieu à plusieurs réponses et exceptions de la part des différents correspondants de la communication.

Dans le cas d'une opération d'appel **call** en mode multidiffusion ou diffusion par communication non bloquante en mode procédure, toutes les exceptions qui peuvent être propagées par les différents correspondants de la communication peuvent être traitées dans des instructions **catch**, **alt** ou **interleave** subséquentes.

Dans le cas d'une opération d'appel **call** en mode multidiffusion ou diffusion par communication bloquante en mode procédure, des options sont possibles. Soit une seule réponse ou exception est traitée dans la partie réponse et traitement d'exception de l'opération d'appel **call**, auquel cas d'autres réponses et exceptions peuvent être traitées dans des instructions **alt** ou **interleave** subséquentes. Soit plusieurs réponses ou exceptions sont traitées par l'utilisation d'instructions de répétition dans un ou plusieurs des blocs d'instructions et de déclarations de la partie réponse et traitement d'exception de l'opération d'appel: l'exécution d'une instruction de répétition déclenche la réévaluation du corps de l'appel.

NOTE – Dans le second cas, l'utilisateur a besoin de traiter le nombre de répétitions.

EXEMPLE 1:

```
var boolean first:= true;
MyPort.call(MyProc:{5,MyVar}, 20E-3) to (MyPeerOne, MyPeerTwo) {
   // Appel en mode multidiffusion de MyProc
   // Traite la réponse reçue de MyPeerOne
   [first] MyPort.getreply(MyProc:{?, ?}) from MyPeerOne {
       if (first) { first := false; repeat; }
       :
   }
   // Traite la réponse reçue de MyPeerTwo
   [first] MyPort.getreply(MyProc:{?, ?}) from MyPeerTwo {
       if (first) { first := false; repeat; }
       :
   }
   [] MyPort.catch(timeout) {   // exception de fin de temporisation après 20 ms
       setverdict(fail);
       stop;
   }
}

alt {
  [] MyPort.getreply(MyProc:{?, ?}) {     // Traite toutes les autres réponses à
                                          // l'appel en mode diffusion
       repeat
  }
}
```

Dans le cas d'une opération d'appel **call** en mode multidiffusion ou diffusion par communication bloquante en mode procédure, où le mot clé **nowait** est utilisé, toutes les réponses et exceptions doivent être traitées dans des instructions **alt** ou **interleave** subséquentes.

EXEMPLE 2:

```
MyPort.call(MyProc:{5,MyVar}) to (MyPeer1, MyPeer2) nowait;
                   // Appel en mode multidiffusion de MyProc

interleave {
  [] MyPort.getreply(MyProc:{?, ?}) from MyPeer1 { } // Traite la réponse de MyPeer1
  [] MyPort.getreply(MyProc:{?, ?}) from MyPeer2 { } // Traite la réponse de MyPeer2
}
```

23.3.2 Opération Getcall (accepter un appel)

23.3.2.0 Généralités

L'opération **getcall** sert à spécifier qu'un composant de test accepte un appel à partir du système SUT, ou d'un autre composant de test. L'opération **getcall** ne doit être utilisée qu'à des ports en mode procédure (ou mixte) et la signature de l'appel de procédure à accepter doit être incluse dans la liste des procédures entrantes autorisées dans la définition du type de port.

L'opération **getcall** doit supprimer l'appel sommital de la file d'attente d'entrée à un port, si et seulement si, les critères d'appariement associés à l'opération **getcall** sont satisfaits. Ces critères d'appariement sont associés à la

signature de l'appel à traiter et au correspondant de la communication. Les critères d'appariement pour la signature peuvent soit être spécifiés en ligne ou être issus d'un modèle de signature.

Une opération **getcall** peut être limitée à un certain correspondant de la communication dans le cas de connexions point-multipoint. Cette restriction doit être indiquée au moyen du mot clé **from**.

EXEMPLE 1:

```
MyPort.getcall(MyProc: MyProcTemplate(5, MyVar));    // accepte un appel de
                                                     // MyProc au port MyPort

MyPort.getcall(MyProc:{5, MyVar}) from MyPeer;       // accepte un appel de MyProc au port
                                                     // MyPort à partir de MyPeer
```

L'argument de signature de l'opération **getcall** ne doit pas servir à faire passer des noms de variable vers des paramètres **in** et **inout**. L'affectation de valeurs paramétriques **in** et **inout** à des variables doit être effectuée dans la partie affectation de l'opération **getcall**. Cela permet d'utiliser des modèles de signature dans des opérations de type **getcall** de la même façon que des modèles sont utilisés pour des types.

La partie affectation (facultative) de l'opération **getcall** inclut l'affectation à des variables de valeurs paramétriques **in** et **inout** et l'extraction de l'adresse du composant appelant. La partie affectation de valeurs ne doit pas être utilisée avec l'opération **getcall**. Le mot clé **param** sert à extraire la valeur paramétrique d'un appel.

Le mot clé **sender** est utilisé quand il est nécessaire d'extraire l'adresse de l'expéditeur (p. ex. afin d'adresser une réponse **reply** ou une exception à au correspondant appelant dans une configuration point-multipoint).

EXEMPLE 2:

```
MyPort.getcall(MyProc:{?, ?}) from MyPartner -> param (MyPar1Var, MyPar2Var);
// La valeur paramétrique in ou inout de MyProc est affectée à MyPar1Var et à MyPar2Var.

MyPort.getcall(MyProc:{5, MyVar}) -> sender MySenderVar;
// Accepte un appel de MyProc à MyPort avec les paramètres in ou
// inout 5 et MyVar.
// L'adresse du correspondant appelant est extraite et mémorisée
// dans MySenderVar.

// Les exemples suivants d'opération getcall montrent les possibilités
// d'utiliser des attributs d'appariement et d'omettre des parties
// facultatives, qui peuvent n'avoir aucune importance pour
// la spécification du test.

MyPort.getcall(MyProc:{5, MyVar}) -> param(MyVar1, MyVar2) sender MySenderVar;

MyPort.getcall(MyProc:{5, ?}) -> param(MyVar1, MyVar2);

MyPort.getcall(MyProc:{?, MyVar}) -> param( - , MyVar2);
// La valeur du premier paramètre de type inout n'est pas importante
// ou n'est pas utilisée

// Les exemples suivants doivent expliquer les possibilités d'affecter
// des valeurs paramétriques de type in et inout à des variables.
// La signature suivante est censée exister pour la procédure à appeler:

signature MyProc2(in integer A, integer B, integer C, out integer D, inout integer E);

MyPort.getcall(MyProc2:{?, ?, 3, - , ?}) -> param (MyVarA, MyVarB, - , -, MyVarE);
// Les paramètres A, B, et E sont affectés aux variables MyVarA, MyVarB, et
// MyVarE. Le paramètre de type out D n'a pas besoin d'être pris en considération.

MyPort.getcall(MyProc2:{?, ?, 3, -, ?}) -> param (MyVarA:= A, MyVarB:= B, MyVarE:= E);
// Autre notation pour l'affectation à des variables de valeur de paramètre
// in et inout. Il est à noter que les noms contenus dans la liste d'affectation
// se rapportent aux noms utilisés dans la signature de MyProc2

MyPort.getcall(MyProc2:{1, 2, 3, -, *}) -> param (MyVarE:= E);
// Seule la valeur paramétrique inout est requise pour la suite de
// l'exécution du test élémentaire
```

23.3.2.1 Acceptation d'un appel quelconque

Une opération **getcall** sans liste d'arguments pour les critères d'appariement de signature va supprimer l'appel sommital de la file d'attente (éventuelle) d'entrée à un port si tous les autres critères d'appariement sont satisfaits. Les paramètres des appels acceptés par l'opération *AcceptAnyCall* ne doivent pas être affectés à une variable.

EXEMPLE:

```
MyPort.getcall;                         // Supprime l'appel sommital de MyPort.

MyPort.getcall from MyPartner;          // Supprime un appel de MyPartner au port MyPor

MyPort.getcall -> sender MySenderVar;   // Supprime un appel de MyPort et
                                        // extrait l'adresse de l'entité
                                        // appelante
```

23.3.2.2 Traitement d'appel à un port quelconque

L'opération **getcall** à un port quelconque est indiquée par le mot clé **any**.

EXEMPLE:

```
any port.getcall(MyProc)
```

23.3.3 L'opération "Reply" (réponse)

L'opération **reply** sert à répondre à un appel déjà accepté conformément à la signature de procédure. Une opération de type **reply** ne doit être utilisée qu'à un port en mode procédure (ou mixte). La définition de type du port doit comprendre le nom de la procédure à laquelle l'opération de type **reply** se rapporte.

NOTE – La relation entre un appel accepté et une opération de type **reply** ne peut pas toujours être vérifiée statiquement. Pour les tests, il est permis de spécifier une opération de type **reply** sans opération **getcall** associée.

La partie valeur de l'opération de type **reply** se compose d'une référence de signature avec une liste associée de paramètres effectifs et une valeur de retour (facultative). La signature peut être définie soit sous la forme d'un modèle de signature ou en ligne. Tous les paramètres **out** et **inout** de la signature doivent avoir une valeur spécifique; c'est-à-dire que l'utilisation de mécanismes d'appariement comme *AnyValue* n'est pas autorisée.

Les réponses à une ou plusieurs opération **call** peuvent être envoyées à une, plusieurs ou la totalité des entités homologues connectées au port adressé. Cela peut être spécifié comme indiqué dans le § 23.2.1.1. Autrement dit, l'argument de la clause **to** d'une opération de type **reply** est, pour les réponses en mode monodiffusion, l'adresse d'une entité réceptrice, pour les réponses en mode multidiffusion une liste d'adresses d'un ensemble de récepteurs et, pour les réponses en mode diffusion, les mots clés **all component**.

Dans le cas de connexions point à point, la clause **to** peut être omise, parce que l'entité réceptrice est définie de façon univoque par la structure du système.

Si une valeur doit être retournée au correspondant de l'appel, cela doit être explicitement indiqué au moyen du mot clé **value**.

EXEMPLE:

```
MyPort.reply(MyProc2:{ - ,5});                       // Réponses à un appel accepté de MyProc2.

MyPort.reply(MyProc2:{ - ,5}) to MyPeer;            // Réponses à un appel accepté de
                                                    // MyProc2 à partir de MyPeer
MyPort.reply(MyProc2:{ - ,5}) to (MyPeer1, MyPeer2); // Réponse en mode multidiffusion à
                                                     // MyPeer1 et MyPeer2

MyPort.reply(MyProc2:{ - ,5}) to all component;     // Réponse en mode diffusion à toutes les
                                                    // entités connectées au port MyPort

MyPort.reply(MyProc3:{5,MyVar} value 20);           // Réponses à un appel accepté de MyProc3.
```

23.3.4 L'opération "Getreply" (traitement de réponse)

23.3.4.0 Généralités

L'opération **getreply** sert à traiter les réponses à une procédure déjà appelée. Une opération **getreply** ne doit être utilisée qu'à un port en mode procédure (ou mixte). La définition de type du port doit comprendre le nom de la procédure à laquelle l'opération de type **getreply** se rapporte.

L'opération **getreply** opération doit supprimer la réponse sommitale de la file d'attente d'entrée à un port, si et seulement si les critères d'appariement associés à l'opération **getreply** sont satisfaits. Ces critères d'appariement sont associés à la signature de la procédure à traiter et au correspondant de la communication. Les critères d'appariement pour la signature peuvent soit être spécifiés en ligne ou être issus d'un modèle de signature.

L'appariement en fonction d'une valeur de retour reçue peut être spécifié au moyen du mot clé **value**.

Une opération **getreply** peut être limitée à un certain correspondant de la communication dans le cas de connexions point-multipoint. Cette restriction doit être indiquée au moyen du mot clé **from**.

EXEMPLE 1:

```
MyPort.getreply(MyProc:{5, ?} value 20); // Accepte une réponse de MyProc avec deux
                                         // paramètres out ou inout et une valeur de
                                         // retour égale à 20

MyPort.getreply(MyProc2:{ - , 5}) from MyPeer;  // Accepte une réponse de MyProc2
                                                // à partir de MyPeer
```

L'argument de signature de l'opération **getreply** ne doit pas servir à faire passer des noms de variable vers des paramètres **out** et **inout**. L'affectation de valeurs paramétriques **out** et **inout** à des variables doit être effectuée dans la partie affectation de l'opération **getreply**. Cela permet d'utiliser des modèles de signature dans des opérations **getreply** de la même façon que des modèles sont utilisés pour des types.

La partie affectation (facultative) de l'opération **getreply** inclut l'affectation de valeurs paramétriques **out** et **inout** à des variables et l'extraction de l'adresse d'expéditeur de la réponse. Le mot clé **value** sert à extraire les valeurs de retour et le mot clé **param** sert à extraire les valeurs paramétriques d'une réponse. Le mot clé **sender** est utilisé quand il est nécessaire d'extraire l'adresse de l'expéditeur.

EXEMPLE 2:

```
MyPort.getreply(MyProc1:{?, ?} value ?) -> value MyRetValue param(MyPar1,MyPar2);
  // La valeur retournée est affectée à la variable MyRetValue et la valeur
  // des deux paramètres out ou inout est affectée aux variables MyPar1 et
  // MyPar2.

MyPort.getreply(MyProc1:{?, ?} value ?) -> value MyRetValue param( - , MyPar2) sender MySender;
  // La valeur du premier paramètre n'est pas prise en compte pour la suite
  // de l'exécution du test et l'adresse du composant expéditeur
  // est extraite et mémorisée dans la variable MySender.

  // L'exemple suivant décrit certaines possibilités d'affectation de valeurs
  // paramétriques de type out et inout à des variables. La signature
  // suivante est censée exister pour la procédure qui a été appelée

signature MyProc2(in integer A, integer B, integer C, out integer D, inout integer E);

MyPort.getreply(ATemplate) -> param( - , - , - , MyVarOut1, MyVarInout1);

MyPort.getreply(ATemplate) -> param(MyVarOut1:=D, MyVarOut2:=E);

MyPort.getreply(MyProc2:{ - , - , - , 3, ?}) -> param(MyVarInout1:=E);
```

23.3.4.1 Opération "Getreply" sur une réponse quelconque

Une opération **getreply** sans liste d'arguments pour les critères d'appariement de signature doit supprimer le message de réponse sommital de la file d'attente (éventuelle) d'entrée à un port si tous les autres critères d'appariement sont satisfaits. Les paramètres ou valeurs de retour des réponses acceptées par l'opération *GetAnyReply* ne doivent pas être affectés à une variable. Si l'opération *GetAnyReply* est utilisée dans la partie réponse et traitement d'exception d'une opération **call**, elle ne doit traiter que les réponses de la procédure invoquée par l'opération **call**.

EXEMPLE:

```
MyPort.getreply;              // Supprime la réponse sommitale de MyPort.

MyPort.getreply from MyPeer;  // Supprime la réponse sommitale reçue de
                              // MyPeer à partir de MyPort.

MyPort.getreply -> sender MySenderVar;  // Supprime la réponse sommitale de MyPort
                                        // et extrait l'adresse de l'entité expéditrice
```

23.3.4.2 Traitement d'une réponse à un port quelconque

Afin de traiter une réponse à un port quelconque, il convient d'utiliser les mots clés **any port**.

EXEMPLE:

```
any port.getreply(Myproc)
```

23.3.5 L'opération "Raise" (propagation)

L'opération **raise** sert à propager une exception. Une exception ne doit être propagée qu'à un port en mode procédure (ou mixte). Une exception est une réaction à un appel de procédure accepté, dont le résultat conduit à un événement exceptionnel. Le type de l'exception doit être spécifié dans la signature de la procédure appelée. La définition de type du port doit comprendre, dans sa liste d'appels de procédure acceptés, le nom de la procédure à laquelle l'exception se rapporte.

NOTE – La relation entre un appel accepté et une opération **raise** ne peut pas toujours être vérifiée statiquement. Pour les tests, il est permis de spécifier une opération **raise** sans opération **getcall** associée.

La partie valeur de l'opération **raise** se compose de la référence de signature suivie par la valeur de l'exception.

Les exceptions sont spécifiées comme des types. La valeur de l'exception peut donc soit être issue d'un modèle ou être la valeur résultant d'une expression (qui évidemment peut être une valeur explicite). Le champ facultatif de type contenu dans la spécification de valeur de l'opération **raise** doit être utilisé quand il est nécessaire d'éviter toute ambiguïté quant au type de la valeur en cours d'expédition.

Les exceptions à une ou plusieurs opérations **call** peuvent être envoyées à une, plusieurs ou la totalité des entités homologues connectées au port adressé. Cela peut être spécifié comme indiqué dans le § 23.2.1.1. Autrement dit, l'argument de la clause to d'une opération **raise** est, pour les exceptions en mode monodiffusion, l'adresse d'une entité réceptrice, pour les exceptions en mode multidiffusion une liste d'adresses d'un ensemble de récepteurs et, pour les exceptions en mode diffusion, les mots clés **all component**.

Dans le cas de connexions point à point, la clause **to** peut être omise, car l'entité réceptrice est identifiée de façon univoque par la structure du système.

EXEMPLE:

```
MyPort.raise(MySignature, MyVariable + YourVariable - 2);
// Propage une exception avec une valeur qui est le résultat de
// l'expression arithmétique au port MyPort

MyPort.raise(MyProc, integer:5});   // Propage une exception avec la valeur
                                    // d'entier 5 pour MyProc

MyPort.raise(MySignature, "My string") to MyPartner;
// Propage une exception avec la valeur "My string" à MyPort pour
// MySignature et l'envoie à MyPartner

MyPort.raise(MySignature, "My string") to (MyPartnerOne, MyPartnerTwo);
// Propage une exception avec la valeur "My string" au port MyPort et
// l'envoie à MyPartnerOne et MyPartnerTwo (c'est-à-dire par communication
// en mode multidiffusion)

MyPort.raise(MySignature, "My string") to all component;
// Propage une exception avec la valeur "My string" au port MyPort pour
// MySignature et l'envoie à toutes les extrémités connectées au port
// MyPort (c'est-à-dire par communication en mode diffusion)
```

23.3.6 L'opération "Catch" (acquisition)

23.3.6.0 Généralités

L'opération **catch** sert à acquérir des exceptions propagées par un composant de test ou le système SUT en réaction à un appel de procédure. L'opération **catch** ne doit être utilisée qu'à des ports en mode procédure (ou mixtes). Le type de l'exception acquise doit être spécifié dans la signature de la procédure qui a propagé l'exception. Les exceptions sont spécifiées comme des types et peuvent donc être traitées comme des messages. Des modèles peuvent par exemple servir à faire la distinction entre différentes valeurs du même type d'exception.

L'opération **catch** supprime l'exception sommitale de la file d'attente de messages entrants associée à un port si et seulement si cette exception sommitale satisfait tous les critères de correspondance associés à l'opération **catch**. Aucune association des valeurs entrantes aux termes de l'expression ou au modèle ne doit se produire. L'opération des valeurs de l'exception à des variables doit être effectuée dans la partie affectation de l'opération **catch**.

Une opération **catch** peut être limitée à un certain correspondant de la communication dans le cas de connexions point-multipoint. Cette restriction doit être indiquée au moyen du mot clé **from**.

EXEMPLE 1:

```
MyPort.catch(MyProc, integer: MyVar);     // Acquiert une exception de type
                                          // entier de valeur MyVar propagée
                                          // par MyProc au port MyPort.

MyPort.catch(MyProc, MyVar);              // Variante de l'exemple précédent.

MyPort.catch(MyProc, A<B);                // Acquiert une exception de type boléen

MyPort.catch(MyProc, MyType:{5, MyVar});  // Définition de modèle en ligne
                                          // d'une valeur d'exception.
MyPort.catch(MyProc, charstring:"Hello")from MyPeer;
                                          // Acquiert l'exception "Hello" à partir de MyPeer
```

La partie affectation (facultative) de l'opération **catch** inclut l'affectation de la valeur de l'exception et l'extraction de l'adresse du composant d'appel. Le mot clé **value** sert à extraire la valeur d'une exception et le mot clé **sender** est utilisé quand il est nécessaire d'extraire l'adresse de l'expéditeur.

EXEMPLE 2:

```
MyPort.catch(MyProc, MyType:?) from MyPartner -> value MyVar;
// Acquiert une exception à partir de MyPartner et affecte sa valeur à MyVar.

MyPort.catch(MyProc, MyTemplate(5)) -> value MyVarTwo sender MyPeer;
// Acquiert une exception, affecte sa valeur
// à MyVarTwo et extrait l'adresse de l'expéditeur.
```

L'opération **catch** peut faire partie de la partie réponse et traitement d'exception d'une opération d'appel **call** ou servir à déterminer une option dans une instruction **alt**. Si l'opération **catch** est utilisée dans la partie acceptrice d'une opération d'appel **call**, les informations relatives au nom du port et à la référence de signature afin d'indiquer la procédure qui a propagé l'exception sont redondantes parce que ces informations découlent de l'opération d'appel **call**. Cependant, pour des raisons de lisibilité (par exemple dans le cas d'instructions d'appel **call** complexes), ces informations doivent être répétées.

23.3.6.1 L'exception d'expiration de temporisation

Il y a une seule exception **timeout** spéciale, qui peut être acquise par l'opération **catch**. L'exception **timeout** est un échappement d'urgence dans les cas où une procédure appelée ne répond à aucune exception – ni n'en propage – à un moment donné (voir § 23.3.1.2).

EXEMPLE:

```
MyPort.call(MyProc:{5,MyVar}, 20E-3) {
  [] MyPort.getreply(MyProc:{?, ?}) { }
  [] MyPort.catch(timeout) {   // exception de fin de temporisation après 20 m
       setverdict(fail);
       stop;
     }
}
```

L'acquisition d'exceptions de type **timeout** doit être limitée au traitement de la partie exception d'un appel. Aucun autre critère d'appariement (y compris une partie **from**) ni aucune partie affectation n'est autorisé pour une opération **catch** qui traite une exception de type **timeout**.

23.3.6.2 Acquisition d'une exception quelconque

Une opération **catch** sans liste d'arguments permet d'acquérir toute exception valide. Le cas le plus général est celui de la non-utilisation du mot clé **from**. Les valeurs d'exception acceptées par l'opération *CatchAnyException* ne doivent pas être affectées à une variable. Si l'opération *CatchAnyException* est utilisée dans la partie réponse et traitement d'exception de l'opération **call**, elle ne doit traiter que les exceptions propagées par la procédure invoquée par l'opération **call**. L'opération *CatchAnyException* permet également l'acquisition de l'exception **timeout**.

EXEMPLE:

```
MyPort.catch;

MyPort.catch from MyPartner;

MyPort.catch -> sender MySenderVar;
```

23.3.6.3 Acquisition à un port quelconque

Afin d'acquérir **catch** une exception à un port quelconque, il convient d' utiliser le mot clé **any**.

EXEMPLE:

```
any port.catch;
```

23.4 L'opération "Check" (vérification)

23.4.0 Généralités

L'opération de vérification **check** est une opération générique qui permet de lire l'arrivée à l'élément sommital d'une file d'attente en mode message ou en mode procédure *entrant* dans un port sans supprimer cet élément sommital de la file. L'opération **check** doit traiter des valeurs d'un certain type au niveau de ports en mode message et doit distinguer entre appels à accepter, exceptions à acquérir et réponses issues d'appels précédents à des ports en mode procédure.

Les opérations de réception **receive**, **getcall**, **getreply** et **catch**, avec leurs parties d'appariement et d'affectation, sont utilisées par l'opération **check** afin de définir la condition qui doit être vérifiée et d'extraire la ou les valeurs de ses paramètres si nécessaire.

C'est l'élément *sommital* d'une file d'attente d'entrée à un port qui doit être vérifié (il n'est pas possible de rechercher *dans* la file). Si celle-ci est vide, l'opération de vérification **check** échoue. Si la file n'est pas vide, une copie de l'élément sommital est prise et l'opération de réception spécifiée dans l'opération de vérification **check** est effectuée sur la copie. L'opération de vérification **check** échoue si l'opération de réception échoue; c'est-à-dire que les critères d'appariement ne sont pas satisfaits. Dans ce cas, la *copie* de l'élément sommital de la file est rejetée et l'exécution du test continue de la façon normale; c'est-à-dire que la prochaine instruction ou variante de l'opération de vérification est évaluée. L'opération de vérification **check** est efficace si l'opération de réception est efficace.

L'utilisation de l'opération de vérification **check** de façon erronée, p. ex. la vérification d'une exception à un port en mode message, doit provoquer une erreur de test élémentaire.

NOTE – Dans la plupart des cas, l'utilisation correcte de l'opération de vérification peut être vérifiée statiquement, c'est-à-dire avant/pendant la compilation.

EXEMPLE:

```
MyPort1.check(receive(5)); // Vérifications pour un message d'entier de valeur 5.

MyPort2.check(getcall(MyProc:{5, MyVar}) from MyPartner);
// Vérifications pour un appel de MyProc au port MyPort2
// à partir de MyPartner

MyPort2.check(getreply(MyProc:{5, MyVar} value 20));
// Vérifications pour une réponse à partir de la procédure MyProc à MyPort2
// où la valeur retournée est 20 et où les valeurs des deux
// paramètres out et inout sont les suivantes: 5 et  la valeur de MyVar.

MyPort2.check(catch(MyProc, MyTemplate(5, MyVar)));

MyPort2.check(getreply(MyProc1:{?, MyVar} value *) -> value MyReturnValue param(MyPar1,-));

MyPort.check(getcall(MyProc:{5, MyVar}) from MyPartner -> param (MyPar1Var, MyPar2Var));

MyPort.check(getcall(MyProc:{5, MyVar}) -> sender MySenderVar);
```

23.4.1 L'opération "Check any" (vérification sans liste d'arguments)

Une opération de vérification **check** sans liste d'arguments permet de vérifier si quelque chose est en attente de traitement dans une file d'attente d'entrée à un port. L'opération **check** sans liste permet de distinguer entre différents expéditeurs (en cas de connexions point-multipoint) au moyen d'une clause **from** et permet d'extraire l'expéditeur au moyen d'une partie affectation d'abrégé avec une clause **sender**. En cas de ports mixtes, l'opération **check** sans liste vérifie les deux files d'attente d'entrée, en mode message et en mode procédure, du port mixte. Si l'opération **check** sans liste correspond aux deux files d'attente d'entrée du port mixte, les informations relatives à la file d'attente en mode procédure doivent être prioritaires, c'est-à-dire retournées à la suite de l'opération **check** sans liste. Par exemple, si les files d'attente d'entrée en mode message et en mode procédure d'un port mixte ne sont pas vides et que les informations relatives à l'expéditeur devraient être extraites par une opération **check** sans liste, l'expéditeur de l'appel, de la réponse ou de l'exception dans la file d'attente d'entrée en mode procédure doit être retourné.

NOTE – Les informations relatives à la file d'attente d'entrée en mode message d'un port mixte peuvent être extraites facilement au moyen de l'opération **check** en combinaison avec une opération **receive** sans liste, par exemple

```
MyPort.check(receive) -> sender Mysender.
```

EXEMPLE:

```
MyPort.check;

MyPort.check(from MyPartner);

MyPort.check(-> sender MySenderVar);
```

23.4.2 Vérification à un port quelconque

Afin de procéder à une opération de vérification **`check`** à un port quelconque, utiliser les mots clés **`any port`**. Dans le cas d'une opération de vérification **`check`** à un port quelconque sans argument, les files d'attente d'entrée des ports mixtes doivent être vérifiées comme indiqué dans le § 23.4.1.

EXEMPLE:

```
any port.check;
```

23.5 Opérations de commande des ports de communication

23.5.0 Généralités

En notation TTCN-3, les opérations de commande des ports en mode message, en mode procédure et mixtes sont les suivantes:

- **`clear`**: supprimer le contenu de la file d'attente d'entrée à un port;
- **`start`**: supprimer le contentu de la file d'attente d'entrée à un port et activer les opérations d'envoi et de réception à celui-ci;
- **`stop`**: désactiver les opérations d'envoi et interdire les opérations de réception pour l'appariement avec le port;
- **`halt`**: désactiver immédiatement les opérations d'envoi au port et interdire les opérations de réception pour l'appariement des nouveaux messages/appels/réponses/exceptions qui entrent dans la file d'attente du port après que l'opération d'interruption halt a été exécutée. Les entrées qui figurent déjà dans la liste d'attente peuvent encore être traitées.

23.5.1 L'opération "Clear port" (libération de port)

L'opération **`clear`** supprime le contenu de la file *entrante* du port spécifié. Si la file d'attente à un port est déjà vide, alors cette opération ne doit avoir aucun effet.

EXEMPLE:

```
MyPort.clear;  // libère le port MyPort
```

23.5.2 L'opération "Start port" (ouverture de port)

Si un port est défini comme permettant des opérations de réception de type **`receive`**, **`getcall`**, etc., l'opération **`start`** libère la file entrante du port nommé et commence la détection du trafic à ce port. Si celui-ci est défini de façon à permettre des opérations d'envoi, alors les opérations de type **`send`**, **`call`**, **`raise`**, etc., peuvent également être effectuées à ce port.

EXEMPLE:

```
MyPort.start;  // ouvre le port MyPort
```

Par défaut, tous les ports d'un composant doivent être lancés implicitement quand un composant est créé. L'opération d'ouverture de port provoquera la réouverture des ports non fermés, en supprimant tous les messages en attente dans la file entrante.

23.5.3 L'opération "Stop port" (fermeture de port)

Si un port est défini comme permettant des opérations de réception comme **`receive`** et **`getcall`**, l'opération **`stop`** provoque l'arrêt de la détection au port nommé. Si celui-ci est défini de façon à permettre les opérations d'envoi, alors l'opération **`stop`** interdit l'exécution d'opérations comme **`send`**, **`call`**, **`raise`**, etc.

EXEMPLE 1:

```
MyPort.stop;   // ferme le port MyPort
```

NOTE – Arrêter la détection au port signifie que toutes les opérations de réception définies avant l'opération de fermeture doivent être complètement effectuées avant que le fonctionnement du port soit suspendu.

EXEMPLE 2:

```
MyPort.receive (MyTemplate1) -> value RecPDU;
                                    // la valeur reçue est décodée,
                                    // appariée en fonction de
                                    // MyTemplate1 et la valeur
                                    // d'appariement est mémorisée
                                    // dans la variable RecPDU
MyPort.stop;            // Aucune opération de réception définie après
                        // l'opération de fermeture n'est exécutée (à moins que
                        // le port ne soit rouvert par une opération
MyPort.receive (MyTemplate2);       // Cette opération ne s'apparie pas et se
                                    // bloquera (à supposer qu'aucune valeur
```

23.5.4 L'opération "halt port" (interruption de port)

Si un port autorise des opérations de réception telles que **receive**, **trigger** et **getcall**, l'opération d'interruption **halt** interdit l'aboutissement des opérations de réception pour les messages et les éléments d'appel de procédure qui entrent dans la file d'attente à un port après exécution de l'opération **halt** à ce port. Les messages et les éléments d'appel de procédure qui étaient déjà dans la file d'attente avant l'opération **halt** peuvent toujours être traités avec les opérations de réception. Si le port autorise les opérations d'envoi, alors le port **halt** interdit immédiatement l'exécution des opérations d'envoi telles que **send**, **call**, **raise** etc. Les opérations d'interruption subséquentes n'ont aucun effet sur l'état du port ou de sa file d'attente.

NOTE 1 – L'opération d'interruption de port **halt** met pour ainsi dire un marqueur après la dernière entrée reçue dans la file d'attente au moment où l'opération a été exécutée. Les entrées figurant devant le marqueur peuvent être traitées normalement. Une fois que toutes les entrées figurant dans la file d'attente devant le marqueur ont été traitées, l'état du port est équivalent à l'état fermé (stopped).

NOTE 2 – Si une opération de fermeture de port **stop** est exécutée sur un port interrompu avant que toutes les entrées figurant dans la file d'attente devant le marqueur aient été traitées, les autres opérations de réception sont immédiatement interdites (c'est-à-dire que le marqueur est pour ainsi dire transféré au sommet de la file d'attente).

NOTE 3 – Une opération d'ouverture de port **start** sur un port interrompu libère toutes les entrées figurant dans la file d'attente indépendamment du fait qu'elles soient arrivées avant ou après l'exécution de l'opération d'interruption de port **halt**. Elle supprime également le marqueur.

NOTE 4 – Une opération de libération de port **clear** sur un port interrompu libère toutes les entrées figurant dans la file d'attente indépendamment du fait qu'elles soient arrivées avant ou après l'exécution de l'opération d'interruption de port **halt**. Elle met aussi pour ainsi dire le marqueur au sommet de la file d'attente.

EXEMPLE:

```
MyPort.halt;                      // aucune opération d'envoi autorisée sur le port
                                  // Myport à partir de ce moment;
                                  // le traitement des messages dans la file d'attente
                                  // reste possible.
MyPort.receive (MyTemplate1);     // si un message était déjà dans la file
                                  // d'attente avant l'opération d'interruption halt et
                                  // s'il correspond à MyTemplate 1, il est traité;
                                  // sinon, l'opération de réception se bloque.
```

23.6 Utilisation de "any" et de "all" avec des ports

Les mots clés **any** et **all** peuvent être utilisés avec des opérations de configuration et de communication comme indiqué dans le Tableau 18.

Tableau 18/Z.140 – Utilisation des mots clés "any" et "all" avec des ports

Opération	Mot clé autorisé		Exemple
	any	**all**	
receive, trigger, getcall, getreply, catch, check)	oui		any port.receive
connect/map			
start, stop, clear, halt		oui	all port.start

24 Opérations de temporisation

24.0 Généralités

La notation TTCN-3 prend en charge un certain nombre d'opérations de temporisation qui peuvent être utilisées dans des tests élémentaires, dans des fonctions, dans des variantes et dans la partie commande du module.

L'on part du principe que chaque unité de portée TTCN-3 dans laquelle des temporisations sont déclarées, entretient sa propre *liste de temporisations en cours* et sa propre *liste de temporisations expirées*, c'est-à-dire la liste de toutes les temporisations qui sont effectivement en cours et la liste de toutes les temporisations qui ont expiré. Les listes de temporisations font partie des instantanés d'analyse qui sont pris quand un test élémentaire est exécuté. Une liste des temporisations est mise à jour si une temporisation située dans l'unité de portée est lancée, est arrêtée, arrive à expiration ou si une opération de type **`timeout`** est exécutée.

NOTE 1 – La liste des temporisations en cours et la liste des temporisations expirées ne sont que des listes théoriques qui ne limitent pas l'implémentation des temporisations. D'autres structures de données, comme un ensemble où l'accès aux événements d'expiration de temporisation n'est pas limité, par exemple, par l'ordre dans lequel les événements d'expiration de temporisation se sont produits, peuvent également être utilisées.

NOTE 2 – L'on part du principe que, pour chaque composant de test, il existe une liste spéciale des temporisations en cours et une liste des temporisations expirées, qui gèrent les événements de lancement/d'arrêt des temporisations déclarées dans la définition correspondante du type de composant.

Quand une temporisation expire (théoriquement immédiatement avant un traitement instantané d'un ensemble d'événements contingents), un événement d'expiration de temporisation est placé dans la liste des temporisations expirées de l'unité de portée dans laquelle la temporisation a été déclarée. Cette temporisation devient immédiatement inactive. Une seule entrée par temporisation particulière peut apparaître dans la liste des temporisations expirées de l'unité de portée dans laquelle cette temporisation a été déclarée à un moment quelconque.

Toutes les temporisations en cours doivent automatiquement être annulées quand le composant est explicitement ou implicitement arrêté.

Tableau 19/Z.140 – Aperçu général des opérations de temporisation TTCN-3

Opérations de temporisation	
Instruction	**Mot clé ou symbole associé**
Lancement de temporisation	**`start`**
Arrêt de temporisation	**`stop`**
Lecture de la durée écoulée	**`read`**
Vérifier si la temporisation est en cours	**`running`**
Evénement d'expiration de temporisation	**`timeout`**

24.1 L'opération "Start timer" (lancement de temporisation)

L'opération de lancement de temporisation **`start`** sert à indiquer qu'une temporisation devrait être lancée. Les valeurs de temporisation doivent être des nombres non négatifs de type **`float`** (c'est-à-dire supérieurs ou égaux à 0,0). Quand une temporisation est lancée, son nom est ajouté à la liste des temporisations en cours (pour l'unité de portée indiquée).

EXEMPLE:

```
MyTimer1.start;        // MyTimer1 est lancé avec la durée par défaut
MyTimer2.start(20E-3); // MyTimer2 est lancé avec une durée de 20 ms

// Des éléments de matrices de temporisation peuvent également être lancés en boucle,
// par exemple:
timer t_Mytimer [5];
var float v_timerValues [5];

for (var integer i := 0; i<=4; i:=i+1)
  { v_timerValues [i] := 1.0 }

for (var integer i := 0; i<=4; i:=i+1)
  {t_Mytimer [i].start ( v_timerValues [i])}
```

Le paramètre facultatif de valeur de temporisation doit être utilisé si aucune durée par défaut n'est indiquée, ou si l'on souhaite neutraliser la valeur par défaut spécifiée dans la déclaration de temporisation. Quand une durée de temporisation est neutralisée, la nouvelle valeur s'applique seulement à l'instance actuelle de la temporisation et toute

opération ultérieure de type **start** pour cette temporisation, qui ne spécifie pas de durée, doit utiliser la durée par défaut.

Lancer une temporisation avec la valeur de temporisation 0,0 signifie que la temporisation arrive à expiration immédiatement. Lancer une temporisation avec une valeur négative de temporisation (par exemple, si la valeur de temporisation est le résultat d'une expression ou n'est pas spécifiée) doit provoquer une erreur d'exécution.

L'horloge de temporisation fonctionne à partir de la valeur à virgule flottante zéro (0,0) jusqu'à la valeur maximale indiquée par le paramètre de durée.

L'opération **start** peut être appliquée à une temporisation en cours, auquel cas celle-ci est arrêtée et relancée. Toute entrée dans une liste des temporisations expirées pour cette temporisation doit être supprimée de cette liste.

24.2 L'opération "Stop timer" (arrêt de temporisateur)

L'opération **stop** sert à arrêter une temporisation en cours et à la supprimer de la liste des temporisations en cours. Une temporisation arrêtée devient inactive et sa durée écoulée est réglée à la valeur à virgule flottante zéro (0,0).

L'arrêt d'une temporisation inactive est une opération valide, bien qu'elle n'ait aucun effet. L'arrêt d'une temporisation expirée a pour effet de supprimer l'entrée correspondant à cette temporisation dans la liste des temporisations expirées. Le mot clé **all** peut servir à arrêter toutes les temporisations qui sont visibles dans l'unité de portée dans laquelle l'opération **stop** a été appelée.

EXEMPLE:

```
MyTimer1.stop;      // Arrête MyTimer1
all timer.stop;     // Arrête toutes les temporisations en cours
```

24.3 L'opération "Read timer" (lecture de temporisation)

L'opération **read** sert à extraire la durée écoulée depuis que la temporisation spécifiée a été lancée. La valeur retournée doit être de type **float**.

EXEMPLE:

```
var float Myvar;
MyVar := MyTimer1.read;  // Affecter à MyVar la durée écoulée
                         // depuis que MyTimer1 a été lancé
```

L'application de l'opération **read** à une temporisation inactive, c'est-à-dire non énumérée dans la liste des temporisations en cours, retournera la valeur à virgule flottante zéro (0,0).

24.4 L'opération "Running timer" (temporisation en cours)

L'opération de temporisation en cours **running** sert à vérifier si une temporisation est énumérée dans la liste des temporisations en cours de l'unité de portée indiquée (c'est-à-dire qu'elle a été lancée et n'a ni expiré ni été arrêtée). L'opération retourne la valeur **true** si la temporisation est énumérée dans cette liste, **false** sinon.

EXEMPLE:

```
if (MyTimer1.running) { … }
```

24.5 L'opération "Timeout" (expiration de temporisation)

L'opération d'expiration de temporisation **timeout** permet de vérifier l'expiration d'une temporisation, ou de toutes les temporisations, dans une unité de portée d'un composant de test ou dans le module de commande dans lequel l'opération d'expiration de temporisation a été appelée.

Quand une opération **timeout** est traitée, si un nom de temporisation est indiqué, les listes de temporisations expirées sont explorées conformément aux règles de portée TTCN-3. S'il y a un événement d'expiration de temporisation correspondant au nom de cette temporisation, cet événement est supprimé de la liste des temporisations expirées et l'opération **timeout** réussit. L'opération **timeout** ne doit pas être utilisée dans une expression **boolean**, mais elle peut servir à déterminer une option dans une instruction **alt** ou servir d'instruction autonome dans une description de comportement. Dans ce dernier cas, une opération **timeout** est considérée comme étant un abrégé pour une instruction **alt** avec seulement une seule option, c'est-à-dire qu'elle a une sémantique bloquante et permet donc l'attente passive de l'expiration de temporisation(s).

EXEMPLE 1:

```
MyTimer1.timeout; // vérifications d'expiration de la temporisation MyTimer1 déjà lancée
```

Le mot clé **any** utilisé avec l'opération **timeout** (plutôt qu'une temporisation explicitement nommée) est efficace si la liste des temporisations expirées n'est pas vide.

EXEMPLE 2:

```
any timer.timeout; // Vérifications d'expiration de toute temporisation déjà lancée
```

24.6 Résumé de l'utilisation de "any" et de "all" avec des temporisations

Les mots clés **any** et **all** peuvent être utilisés avec des opérations de temporisation comme indiqué dans le Tableau 20.

Tableau 20/Z.140 – Utilisation des mots clés Any et All avec des temporisations

Opération	Mot clé autorisé		Exemple
	any	all	
start			
stop		oui	all timer.stop
read			
running	oui		if (any timer.running) {…}
timeout	oui		any timer.timeout

25 Opérations de verdict de test

25.0 Généralités

Les opérations de verdict permettent d'insérer et d'extraire des verdicts au moyen, respectivement, des opérations **setverdict** et **getverdict**. Ces opérations ne doivent être utilisées que dans des tests élémentaires, des variantes et des fonctions.

Tableau 21/Z.140 – Aperçu général des opérations de verdict de test

Opérations de verdict de test	
Instruction	**Mot clé ou symbole associé**
Mise à jour de verdict local	setverdict
Requête de verdict local	getverdict

Chaque composant de test de la configuration active doit conserver son propre verdict local. Le verdict local est un objet qui est créé pour chaque composant de test au moment de sa création. Il sert à suivre le verdict individuel dans chaque composant de test (c'est-à-dire dans le composant MTC et dans chacun des composants PTC).

25.1 Verdict de test élémentaire

Il y a par ailleurs un verdict global de test élémentaire, instancié et traité par le système de test, qui est mis à jour quand chaque composant de test (c'est-à-dire le composant MTC et chacun des composants PTC) met fin à son exécution. Ce verdict n'est pas accessible aux opérations **getverdict** et **setverdict**. La valeur de ce verdict doit être retournée par le test élémentaire quand il met fin à l'exécution. Si le verdict retourné n'est pas explicitement sauvegardé dans la partie commande (par exemple, affecté à une variable), alors il est perdu.

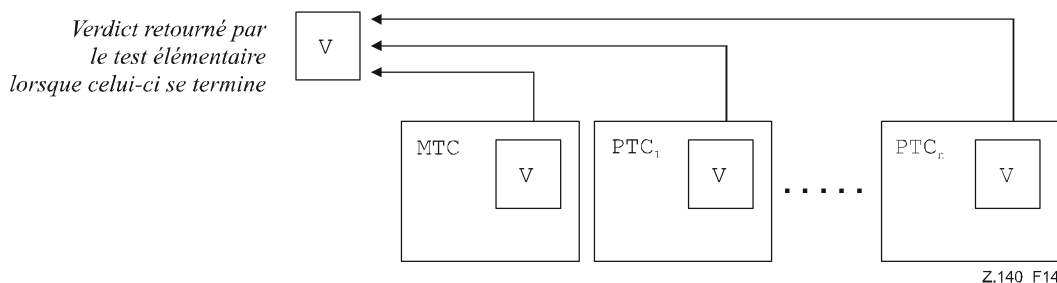


Figure 14/Z.140 – Illustration de la relation entre verdicts

NOTE – La notation TTCN-3 ne spécifie pas les mécanismes effectifs qui effectuent la mise à jour des verdicts locaux et de test élémentaire. Ces mécanismes dépendent de l'implémentation.

25.2 Valeurs de verdict et règles de surécriture

25.2.0 Généralités

Le verdict peut avoir cinq valeurs différentes: **pass**, **fail**, **inconc**, **none** et **error**, qui sont les valeurs distinctives du type **verdicttype** (voir § 6.1).

NOTE – La valeur **inconc** indique un verdict non concluant.

L'opération **setverdict** ne doit être utilisée qu'avec les valeurs **pass**, **fail**, **inconc** et **none**.

EXEMPLE 1:

```
setverdict(pass);
setverdict(inconc);
```

La valeur du verdict local peut être extraite au moyen de l'opération **getverdict**.

EXEMPLE 2:

```
MyResult := getverdict; // Où MyResult est une variable de type verdicttype
```

Quand un composant de test est instancié, son objet de verdict local est créé et mis à la valeur **none**.

Lors d'une modification de la valeur du verdict local (c'est-à-dire au moyen de l'opération **setverdict**) l'effet de cette modification doit suivre les règles de surécriture énumérées dans le Tableau 22. Le verdict de test élémentaire est implicitement mis à jour à la fin d'un composant de test. L'effet de cette opération implicite doit également suivre les règles de surécriture énumérées dans le Tableau 22.

Tableau 22/Z.140 – Règles de surécriture pour le verdict

Valeur actuelle du verdict	Nouvelle valeur d'attribution de verdict			
	pass	**inconc**	**fail**	**none**
none	**pass**	**inconc**	**fail**	**none**
pass	**pass**	**inconc**	**fail**	**pass**
inconc	**inconc**	**inconc**	**fail**	**inconc**
fail	**fail**	**fail**	**fail**	**fail**

EXEMPLE 3:

```
 :
setverdict(pass); // Le verdict local est réglé à pass
 :
setverdict(fail); // Jusqu'à ce que cette ligne soit exécutée, ce qui
 :                // produira le recouvrement de la valeur du verdict
                  // local par la valeur fail
 :                // Quand le composant PTC s'achève, le verdict de test
                  // élémentaire est réglé à fail
```

25.2.1 Verdict d'erreur

Le verdict **error** est spécial parce qu'il est posé par le système de test afin d'indiquer qu'une erreur de test élémentaire (c'est-à-dire d'exécution) s'est produite. Il ne doit pas être posé par l'opération **setverdict** et ne sera pas retourné par

l'opération **getverdict**. Aucune autre valeur de verdict ne peut neutraliser un verdict d'erreur. Autrement dit, un verdict d'erreur **error** ne peut résulter que d'une opération **execute** d'exécution de test élémentaire.

26 Actions externes

Dans certaines situations de test, une ou plusieurs interfaces avec le système SUT peuvent faire défaut ou être a priori inconnues (par exemple, une interface de gestion) mais il peut être nécessaire que le système SUT soit invité à effectuer certaines actions (par exemple, envoyer un message au système de test). Certaines actions peuvent également être requises de la part du personnel d'exécution des tests (par exemple, afin de modifier les conditions climatiques des tests comme la température, la tension de l'alimentation en énergie, etc.).

L'action requise peut être décrite comme une expression chaîne, c'est-à-dire que l'utilisation de chaînes de littéraux, de variables et de paramètres de type chaîne, etc., et de toute concaténation de ces éléments est autorisée.

EXEMPLE:

```
var charstring myString:= " now."
action("Send MyTemplate on lower PCO" & myString);    // Description informelle de
                                                      // l'action externe.
```

Il n'y a aucune spécification concernant l'instruction à destination ou en provenance du système SUT qui déclenche cette action. Il n'y a qu'une description informelle de l'action requise proprement dite.

Les actions externes peuvent être utilisées dans des tests élémentaires, des fonctions, des variantes et dans la partie commande du module.

27 Partie d'un module relative à la commande

27.0 Généralités

Les tests élémentaires sont définis dans la partie d'un module relative aux définitions alors que la partie d'un module relative à la commande gère leur exécution. Toutes les variables (éventuelles) définies dans la partie commande d'un module doivent être transmises au test élémentaire par paramétrage si elles doivent être utilisées dans la définition comportementale de ce test élémentaire; c'est-à-dire que la notation TTCN-3 ne prend pas en charge les variables globales de quelque sorte que ce soit.

Au début de chaque test élémentaire, la configuration de test doit être réinitialisée. Autrement dit, tous les composants et ports pilotés par des opérations de type **create**, **connect,** etc. dans un précédent test élémentaire ont été détruits quand ce test élémentaire a été arrêté (et ne sont donc pas 'visibles' par le nouveau test élémentaire).

27.1 Exécution de tests élémentaires

Un test élémentaire est appelé au moyen d'une instruction de type **execute**. A la suite de l'exécution d'un test élémentaire, un verdict de test élémentaire de valeur égale à **none**, **pass**, **inconc**, **fail** ou **error** doit être retourné et peut être affecté à une variable pour traitement complémentaire.

En variante, l'instruction **execute** permet la supervision d'un test élémentaire au moyen d'une durée de temporisation (voir § 27.5).

EXEMPLE:

```
execute(MyTestCase1());                      // exécute MyTestCase1, sans mémoriser le
                                             // verdict de test retourné et sans
                                             // supervision temporelle

MyVerdict := execute(MyTestCase2());         // exécute MyTestCase2 et mémorise
                                             // le verdict résultant dans la
                                             // variable MyVerdict

MyVerdict := execute(MyTestCase3(),5E-3);    // exécute MyTestCase3 et mémorise le verdict
                                             // résultant dans la variable MyVerdict.
                                             // Si le test élémentaire ne se termine pas dans
                                             // les 5 ms, MyVerdict prend la valeur 'error'
```

27.2 Terminaison de tests élémentaires

Un test élémentaire s'achève en même temps que le composant MTC. Après la fin de celui-ci (explicitement ou implicitement), tous les composants de test s'exécutant en parallèle doivent être supprimés par le système de test.

NOTE 1 – Le mécanisme concret d'arrêt de tous les composants PTC dépend de l'utilitaire employé et est donc hors du domaine d'application de la présente Recommandation.

Le verdict final d'un test élémentaire est calculé sur la base des verdicts locaux finals des différents composants de test, conformément aux règles définies dans le § 25. Le verdict local réel d'un composant de test devient son verdict local final quand le composant de test s'achève de lui-même ou est arrêté par lui-même, par un autre composant de test ou par le système de test.

NOTE 2 – Afin d'éviter des conditions critiques lors du calcul de verdicts de test en raison de l'arrêt différé de composants PTC, le composant MTC devrait veiller à ce que tous les composants PTC soient arrêtés (au moyen de l'instruction **`done`** ou **`killed`**) avant de s'arrêter lui-même.

27.3 Contrôle de l'exécution de tests élémentaires

Des instructions de programmation limitées à celles qui sont définies dans les Tableaux 11 et 12 peuvent être utilisées dans la partie commande d'un module afin de spécifier des comportements tels que l'ordre dans lequel les tests élémentaires doivent être exécutés ou le nombre de fois qu'un test élémentaire devrait être exécuté.

EXEMPLE:

```
module MyTestSuite () {
    :
   control {
       :
      // Exécuter ce test 10 fois
      count:=0;
      while (count < 10)
      {    execute (MySimpleTestCase1());
       count := count+1;
      }
   }
}
```

Si aucune instruction de programmation n'est utilisée, alors, par défaut, les tests élémentaires sont exécutés dans l'ordre séquentiel de leur apparition dans la partie commande du module.

NOTE – Cela n'exclut pas la possibilité que certain utilitaires puissent chercher à neutraliser ce séquencement par défaut afin de permettre à un utilisateur ou à un utilitaire de sélectionner un ordre d'exécution différent.

La sélection et la désélection des test élémentaire peuvent aussi être utilisées pour contrôler l'exécution des tests élémentaires (voir § 27.4).

27.4 Sélection de tests élémentaires

Il existe différentes façons, en notation TTCN-3, de sélectionner et de désélectionner des tests élémentaires. Par exemple, des expressions booléennes peuvent servir à sélectionner et à désélectionner les tests élémentaires qui doivent être exécutés. Cela comprend, évidemment, l'utilisation de fonctions qui retournent une valeur **`boolean`**.

EXEMPLE 1:

```
module MyTestSuite () {
    :
   control {
       :
      if (MySelectionExpression1()) {
          execute(MySimpleTestCase1());
          execute(MySimpleTestCase2());
          execute(MySimpleTestCase3());
      }
      if (MySelectionExpression2()) {
          execute(MySimpleTestCase4());
          execute(MySimpleTestCase5());
          execute(MySimpleTestCase6());
      }
       :
   }
}
```

Une autre façon d'exécuter collectivement des tests élémentaires consiste à les regrouper dans une fonction et à exécuter celle-ci à partir de la partie commande du module.

EXEMPLE 2:

```
function MyTestCaseGroup1()
{  execute(MySimpleTestCase1());
   execute(MySimpleTestCase2());
   execute(MySimpleTestCase3());
}
function MyTestCaseGroup2()
{  execute(MySimpleTestCase4());
   execute(MySimpleTestCase5());
   execute(MySimpleTestCase6());
}
 :
control
{  if (MySelectionExpression1()) { MyTestCaseGroup1(); }
   if (MySelectionExpression2()) { MyTestCaseGroup2(); }
    :
}
```

Comme un test élémentaire retourne une valeur unique de type **verdicttype**, il est également possible de commander l'ordre d'exécution d'un tel test en fonction du résultat de celui-ci. L'utilisation du type **verdicttype** TTCN-3 est une autre façon de sélectionner des tests élémentaires.

EXEMPLE 3:

```
if ( execute (MySimpleTestCase()) == pass )
 { execute (MyGoOnTestCase()) }
else
 { execute (MyErrorRecoveryTestCase()) };
```

27.5 Utilisation de temporisateurs dans les commandes

Le temporisateur peut servir à superviser l'exécution d'un test élémentaire. L'on peut utiliser à cette fin une temporisation explicite dans l'instruction **execute**. Si le test élémentaire ne se termine pas dans cette durée, le résultat de l'exécution du test élémentaire doit être un verdict d'erreur et le système de test doit terminer le test élémentaire. Le temporisateur utilisée pour la supervision d'un test élémentaire dépend du système et n'a pas besoin d'être déclaré ni armé.

EXEMPLE 1:

```
MyReturnVal := execute (MyTestCase(), 7E-3);
// Où le verdict retourné sera une erreur si MyTestCase
// ne termine pas son exécution dans les 7 ms
```

Des opérations de temporisation peuvent également être utilisées explicitement afin de commander l'exécution d'un test élémentaire.

EXEMPLE 2:

```
// Exemple de l'utilisation de l'opération de temporisateur armé
while (T1.running or x<10) // Où T1 est un temporisateur déjà armé
{  execute(MyTestCase());
   x := x+1;
}

// Exemple de l'utilisation des opérations "start" et "timeout"

timer T1 := 1.0;
 :
execute(MyTestCase1());
T1.start;
T1.timeout; // Pause avant l'exécution du prochain test élémentaire
execute(MyTestCase2());
```

28 Spécification des attributs

28.0 Généralités

Des attributs peuvent être associés aux éléments de langage TTCN-3 au moyen de l'instruction **with**. La syntaxe de l'argument de l'instruction **with** (c'est-à-dire les attributs effectifs) est définie comme une chaîne de texte libre.

Il y a quatre sortes d'attributs:

a) **display**: permet la spécification d'attributs d'affichage associés à des formats de présentation spécifiques;

b) **encode**: permet de faire référence à des règles de codage spécifiques;

c) **variant**: permet de faire référence à des variantes de codage spécifiques;

d) **extension**: permet la spécification d'attributs définis par l'utilisateur.

28.1 Attributs d'affichage

Tous les éléments de langage TTCN-3 peuvent avoir des attributs de type **display** afin de spécifier la façon dont des éléments de langage particuliers devraient être affichés, par exemple en format de présentation tabulaire.

Des chaînes d'attributs particulières, associées aux attributs d'affichage pour le format de présentation tabulaire (de conformité) peuvent être trouvées dans la Rec. UIT-T Z.141 [1].

Des chaînes d'attributs particulières, associées aux attributs d'affichage pour le format de présentation graphique, peuvent être trouvées dans la Rec. UIT-T Z.142 [2].

D'autres attributs de type **display** peuvent être définis par l'utilisateur.

NOTE – Etant donné que les attributs définis par l'utilisateur ne sont pas normalisés, l'interprétation de ces attributs peut différer entre utilitaires ou peut même ne pas être prise en charge.

28.2 Codage de valeurs

28.2.0 Généralités

Les règles de codage définissent la façon dont une valeur particulière, un modèle particulier, etc. doit être codé et transmis à un **port** de communication, ainsi que la façon dont les signaux reçus doivent être décodés. La notation TTCN-3 ne possède pas de mécanisme de codage de valeurs par défaut. Autrement dit, ces règles ou directives de codage sont définies d'une certaine façon externe à la notation TTCN-3.

En notation TTCN-3, des règles de codage générales ou particulières peuvent être spécifiées au moyen des attributs de type **encode** et **variant**.

28.2.1 Attributs de type "encode"

L'attribut **encode** permet d'effectuer l'association, à une définition TTCN-3, d'une certaine règle ou directive de codage référencée.

La façon dont les règles de codage effectives sont définies (par exemple en langage courant, sous forme de fonctions, etc.) est hors du domaine d'application de la présente Recommandation. Si aucune règle spécifique n'est indiquée, alors le codage doit relever de chaque implémentation.

Les attributs de codage seront le plus souvent utilisés de façon hiérarchique. Le niveau sommital est le module entier; le niveau suivant est un groupe de types et le plus bas niveau est un type individuel ou une définition individuelle:

a) **module**: le codage s'applique à tous les types définis dans le module, y compris les types TTCN-3 (types intégrés);

b) **group**: le codage s'applique à un groupe de définitions de type défini par l'utilisateur;

c) **type ou definition**: le codage s'applique à un seul type défini par l'utilisateur ou à une seule définition;

d) **field**: le codage s'applique à un champ contenu dans un type **record** ou **set** ou dans un modèle **template**.

EXEMPLE:

```
module MyTTCNmodule
{  :
   import from MySecondModule {
      type MyRecord
   }
   with { encode "MyRule 1" }   // Les instances de MyRecord seront codées conformément
                                // à MyRule 1
   :
   type charstring MyType;      // Codage normal conformément à la règle de codage globale
   :
```

```
    group MyRecords
    {   :
        type record MyPDU1
        {
            integer field1,    // field1 sera codé conformément à la 'règle 3'
            boolean field2,    // field1 sera codé conformément à la 'règle 3'
            Mytype  field3     // field1 sera codé conformément à la 'règle 2'
        }
        with { encode (field1, field2) "Rule 3" }
        :
    }
    with { encode "Rule 2" }

}
with { encode "Global encoding rule" }
```

28.2.2 Attributs de type "variant"

Afin de spécifier un raffinement du procédé de codage actuellement spécifié au lieu de son remplacement, l'attribut **`variant`** doit être utilisé. Les attributs de type variant diffèrent des autres attributs en ce qu'ils sont étroitement associés aux attributs de codage. En conséquence, d'autres règles d'écrasement s'appliquent aux attributs de type variant (voir § 28.5.1).

EXEMPLE:

```
module MyTTCNmodule1
{  :
    type charstring MyType;  // Codage normal conformément à la 'règle de codage globale'
    :
    group MyRecords
    {   :
        type record MyPDU1
        {
         integer       field1,     // le champ field1 sera codé conformément à la
                                   // 'règle 2' au moyen de la variante de codage
                                   // 'forme de longueur 3'
         Mytype        field3      // le champ field3 sera codé conformément
                                   // à la 'règle 2' au moyen de tout format
                                   // possible de codage de longueur
        }
        with { variant (field1) "length form 3" }
        :
    }
    with { encode "Rule 2" }

}
with { encode "Global encoding rule" }
```

28.2.3 Chaînes spéciales

Les chaînes suivantes sont les attributs prédéfinis (normalisés) de type **`variant`** pour les simples types de base (voir § E.2.1):

a) les attributs `"8 bit"` et `"unsigned 8 bit"` signifient, quand ils sont appliqués aux types "integer" et "enumerated", que la valeur d'entier ou les nombres entiers associés à des énumérations doivent être manipulés comme s'ils étaient représentés par 8 éléments binaires (octet unique) dans le système;

b) les attributs `"16 bit"` et `"unsigned 16 bit"` signifient, quand ils sont appliqués aux types "integer" et "enumerated", que la valeur d'entier ou les nombres entiers associés à des énumérations doivent être manipulés comme s'ils étaient représentés par 16 éléments binaires (deux octets) dans le système;

c) les attributs `"32 bit"` et `"unsigned 32 bit"` signifient, quand ils sont appliqués aux types "integer" et "enumerated", que la valeur d'entier ou les nombres entiers associés à des énumérations doivent être manipulés comme s'ils étaient représentés par 32 éléments binaires (quatre octets) dans le système;

d) les attributs `"64 bit"` et `"unsigned 64 bit"` signifient, quand ils sont appliqués aux types "integer" et "enumerated", que la valeur d'entier ou les nombres entiers associés à des énumérations doivent être manipulés comme s'ils étaient représentés par 64 éléments binaires (huit octets) dans le système;

e) les attributs "IEEE754 float","IEEE754 double", "IEEE754 extended float" et "IEEE754 extended double" signifient, quand ils sont appliqués à un type "float", que la valeur doit être codée et décodée conformément à la norme IEEE 754 (voir bibliographie).

Les chaînes suivantes sont les attributs prédéfinis (normalisés) de type **variant** pour **charstring** et **universal charstring** (voir § E.2.2):

a) l'attribut "UTF-8" signifie, quand il est appliqué au type "universal charstring", que chaque caractère de la valeur doit être individuellement codé et décodé conformément au format de codage du jeu UCS 8 (UTF-8) défini dans l'Annexe R de l'ISO/CEI 10646 [10];

b) l'attribut "UCS-2" signifie, quand il est appliqué au type "universal charstring", que chaque caractère de la valeur doit être individuellement codé et décodé conformément au format de représentation codée du jeu UCS-2 (voir § 14.1 de l'ISO/CEI 10646 [10]);

c) l'attribut "UTF-16" signifie, quand il est appliqué au type "universal charstring", que chaque caractère de la valeur doit être individuellement codé et décodé conformément au format de codage du jeu UCS 16 (UTF-16) défini dans l'Annexe Q de l'ISO/CEI 10646 [10];

d) l'attribut "8 bit" signifie, quand il est appliqué aux types "charstring" et "universal charstring", que chaque caractère de la valeur doit être individuellement codé et décodé conformément à la représentation codée qui est spécifiée dans l'ISO/CEI 8859 (codage sur 8 éléments binaires).

La chaîne suivante est l'attribut prédéfini (normalisé) de type **variant** pour les types structurés (voir § E.2.3):

a) l'attribut "IDL:fixed FORMAL/01-12-01 v.2.6" signifie, quand il est appliqué à un type "record", que la valeur doit être manipulée comme une valeur décimale en virgule fixe du langage IDL (voir bibliographie).

Ces attributs de type "variant" peuvent être utilisés en combinaison avec les attributs plus généraux de type "encode" qui sont spécifiés à un niveau supérieur. Par exemple, une chaîne de type **universal charstring,** spécifiée avec l'attribut de type **variant** "UTF-8" dans un module qui lui-même possède un attribut de codage global "BER:1997" (voir § 12.2/Z.146 [6]), fera que chaque caractère des valeurs contenues dans cette chaîne sera d'abord codé suivant les règles UTF-8; puis cette valeur UTF-8 sera codée suivant les règles plus globales BER.

28.2.4 Codages non valides

Si l'on souhaite spécifier des règles de codage non valides, alors celles-ci doivent être spécifiées dans une source pouvant faire l'objet de références et située à l'extérieur du module, de la même façon que les règles de codage valides sont citées en référence.

28.3 Attributs d'extension

Tous les éléments de langage TTCN-3 peuvent avoir des attributs de type **extension,** spécifiés par l'utilisateur.

NOTE – Etant donné que les attributs définis par l'utilisateur ne sont pas normalisés, leur interprétation peut différer ou même ne pas être prise en charge selon les utilitaires fournis par différents vendeurs.

28.4 Portée des attributs

Une instruction **with** peut associer des attributs à un élément de langage unique. Il est également possible d'associer des attributs à un certain nombre d'éléments de langage, par exemple en énumérant les champs d'un type structuré dans une instruction d'attribut associée à une unique définition de type ou en associant une instruction **with** à l'unité de portée environnante ou au groupe environnant (**group**) d'éléments de langage.

EXEMPLE:

```
// MyPDU1 sera affichée comme PDU
type record MyPDU1 { … } with { display "PDU"}

// MyPDU2 sera affichée comme PDU avec l'attribut d'extension
// propre à l'application MyRule
type record MyPDU2 { … }
with
{
  display "PDU";
  extension "MyRule"
}

// La définition de groupe suivante …
group MyPDUs {
```

```
   type record MyPDU3 { … }
   type record MyPDU4 { … }
}
with {display "PDU"}    // Tous les types d'unité MyPDUs de groupe seront
                        // affichés comme des unités PDU

// est identique à
group MyPDUs {
   type record MyPDU3 { … } with { display "PDU"}
   type record MyPDU4 { … } with { display "PDU"}
}
```

28.5 Règles d'écrasement pour attributs

Une définition d'attribut située dans une unité de portée inférieure neutralisera un définition générale d'attribut dans une unité de portée supérieure. Les autres règles d'écrasement pour les attributs de type **variant** sont définies dans le § 28.5.1.

EXEMPLE 1:

```
type record MyRecordA
{
   :
} with { encode "RuleA" }

// Ci-dessous, MyRecordA est codé conformément à la règle RuleA et non
// conformément à la règle RuleB
type record MyRecordB
{
    :
   field   MyRecordA
} with { encode "RuleB" }
```

Une instruction **with** qui est placée à l'intérieur de l'unité de portée d'une autre instruction **with** doit neutraliser l'instruction **with** située le plus à l'extérieur. Cela doit également s'appliquer à l'utilisation de l'instruction **with** avec des groupes. Il convient de prendre des précautions quand le procédé de surécriture est utilisé en combinaison avec des références à des définitions isolées. La règle générale est que les attributs doivent être attribués et écrasés conformément à leur ordre d'apparition.

```
// Exemple de l'utilisation du procédé d'écrasement de l'instruction with
group MyPDUs
{
   type record MyPDU1 { … }
   type record MyPDU2 { … }

   group MySpecialPDUs
   {
      type record MyPDU3 { … }
      type record MyPDU4 { … }
   }
   with {extension "MySpecialRule"}  // MyPDU3 et MyPDU4 auront
                                     // l'attribut d'extension propre à
                                     // l'application MySpecialRule
}
with
{
   display "PDU";          // Tous les types d'unité MyPDU de groupe seront
   extension "MyRule";     // affichés comme des unités PDU et auront (s'ils
                           // ne sont pas écrasés) l'attribut d'extension MyRule
}

// est identique à …
group MyPDUs
{
   type record MyPDU1 { … } with {display "PDU"; extension "MyRule" }
   type record MyPDU2 { … } with {display "PDU"; extension "MyRule" }
   group MySpecialPDUs {
      type record MyPDU3 { … } with {display "PDU"; extension "MySpecialRule" }
      type record MyPDU4 { … } with {display "PDU"; extension "MySpecialRule" }
   }
}
```

Une définition d'attribut située dans un niveau de portée inférieur peut être écrasée dans une unité de portée supérieure au moyen de l'instruction **override**.

EXEMPLE 2:

```
type record MyRecordA
{
    :
} with { encode "RuleA" }

// Ci-dessous, MyRecordA est codé conformément à RuleB
type record MyRecordB
{
    :
   fieldA  MyRecordA
} with { encode override "RuleB" }
```

L'instruction **override** force à l'attribut spécifié tous les types contenus dans des unités de portée inférieure.

28.5.1 Autres règles d'écrasement pour les attributs de type variant

Un attribut de type **variant** est toujours associé à un attribut de type **encode**. Alors qu'une variante d'un codage peut changer, un codage ne doit pas changer sans que tous les attributs actuels de type variant soient écrasés. En conséquence, les règles d'écrasement suivantes s'appliquent aux attributs de type variant:

- un attribut de type **variant** écrase un attribut actuel de type **variant** conformément aux règles définies dans le § 28.5;
- un attribut de type **encoding**, qui écrase un attribut actuel de type **encoding** conformément aux règles définies dans le § 28.5, écrase également un attribut actuel correspondant de type **variant**, c'est-à-dire qu'aucun nouvel attribut de type **variant** n'est fourni, mais l'attribut actuel de type **variant** devient inactif;
- un attribut de type **encoding**, qui modifie un attribut actuel de type **encoding** d'un élément de langage importé conformément aux règles définies dans le § 28.6, modifie également un attribut actuel correspondant de type **variant**, c'est-à-dire qu'aucun nouvel attribut de type **variant** n'est fourni, mais l'attribut actuel de type **variant** devient inactif.

EXEMPLE:

```
module MyVariantEncodingModule {
   :
   type charstring MyCharString;     // Codage normal conformément à "Encoding 1"
   :
   group MyVariantsOne {
      :
      type record MyPDUone
      {
       integer         field1,       // Le champ field1 sera codé conformément à
                                     // "Encoding 2" uniquement. "Encoding 2" écrase
                                     // "Encoding 1" et la variante 'Variant 1'
       Mytype          field3        // Le champ field3 sera codé conformément à
                                     // "Encoding 1" au moyen de la variante
                                     // "Variant 1".
      }
      with { encoding (field1) "Encoding 2" }
      :
   }
   with { variant "Variant 1" }

   group MyVariantsTwo
   {  :
      type record MyPDUtwo
      {

       integer    field1,   // Le champ field1 sera codé conformément à "Encoding 3"
                            // au moyen de la variante de codage "Variant 3"
       Mytype     field3    // Le champ field3 sera codé conformément à "Encoding 3"
                            // au moyen de la variante de codage "Variant 2"
      }
      with { variant (field1) "Variant 3" }
      :
   }
   with { encode "Encoding 3"; variant 'Variant 2'}

}
with { encode "Encoding 1" }
```

28.6 Modification des attributs d'éléments de langage importés

En général, un élément de langage est importé en même temps que ses attributs. Dans certains cas, ces attributs peuvent nécessiter une modification lors de l'importation de l'élément de langage, par exemple un type peut être affiché dans un module sous forme de primitive ASP, puis être importé par un autre module où il devra être affiché sous forme d'unité PDU. Dans de tels cas, il est permis de modifier les attributs dans l'instruction d'importation **`import`**.

EXEMPLE:

```
import from MyModule {
  type MyType
}
with { display "ASP" }  // MyType sera affiché comme primitive ASP

import from MyModule {
  group MyGroup
}
with {
  display "PDU";       // Par défaut, tous les types seront affichés comme
                       // comme unité PDU

  extension "MyRule"
}
```

Annexe A

Formalisme BNF et sémantique statique

A.1 Formalisme BNF de la notation TTCN-3

A.1.0 Généralités

La présente annexe définit la syntaxe de la notation TTCN-3 au moyen du formalisme BNF étendu (abrégé ci-dessous en *BNF*).

A.1.1 Conventions pour la description syntaxique

Le Tableau A.1 définit la métanotation utilisée afin de spécifier la grammaire du formalisme BNF étendu en notation TTCN-3.

Tableau A.1/Z.140 – La métanotation syntaxique

::=	est défini comme étant
abc xyz	abc suivi par xyz
|	alternative (entre 2 options)
[abc]	0 ou 1 instance de abc
{abc}	0 ou plusieurs instances de abc
{abc}+	1 ou plusieurs instances de abc
(...)	groupement textuel
Abc	symbole non terminal abc
"abc"	symbole terminal abc

A.1.2 Symboles de terminateur d'instruction

En général, toutes les structures de langage TTCN-3 (c'est-à-dire les définitions, les déclarations, les instructions et les opérations) se terminent par un point-virgule (;). Celui-ci est facultatif si la structure de langage se termine par une accolade de fermeture (}) oui si le symbole suivant est une accolade de fermeture (}), c'est-à-dire que la structure de langage est la dernière instruction d'un bloc d'instructions, d'opérations et de déclarations.

A.1.3 Identificateurs

En notation TTCN-3, les identificateurs sont sensibles à l'inversion majuscules/minuscules et ne peuvent contenir que des lettres minuscules (a-z), des lettres majuscules (A-Z) et des chiffres numériques (0-9). L'utilisation du symbole de soulignement (_) est également autorisée. Un identificateur doit commencer par une lettre (c'est-à-dire par un caractère autre qu'un nombre ou un soulignement).

A.1.4 Commentaires

Des commentaires écrits en langage courant peuvent apparaître n'importe où dans une spécification en notation TTCN-3.

Les commentaires-blocs doivent être ouverts par la paire de symboles /* et fermés par la paire de symboles */.

EXEMPLE 1:

```
/* Ceci est un commentaire-bloc
étalé sur deux lignes */
```

Les commentaires-blocs ne doivent pas être imbriqués.

```
/* Ceci n'est pas /* un commentaire */ légal */
```

Les commentaires-lignes doivent être ouverts par la paire de symboles // et fermés par un caractère d'interligne <*newline*>.

EXEMPLE 2:

```
// Ceci est un commentaire-ligne
// étalé sur deux lignes
```

Les commentaires-lignes peuvent suivre des instructions de programmation TTCN-3 mais ne doivent pas être imbriqués dans une instruction.

EXEMPLE 3:

```
// Ce qui suit n'est pas légal
const // Ceci est MyConst integer MyConst := 1;

// Ce qui suit est légal
const integer MyConst := 1; // Ceci est MyConst
```

A.1.5 Symboles terminaux de la notation TTCN-3

En notation TTCN-3, les symboles terminaux et les mots réservés sont énumérés dans les Tableaux A.2 et A.3.

Tableau A.2/Z.140 – Liste des symboles terminaux spéciaux en notation TTCN-3

Symboles de début/fin de bloc	`{ }`
Symboles de début/fin de liste	`( )`
Symboles d'option	`[ ]`
Symbole de transition (dans une étendue)	`..`
Commentaires-lignes et commentaires-blocs	`/* */ //`
Symbole terminateur de ligne/d'instruction	`;`
Symboles d'opérateur arithmétique	`+ / -`
Symbole d'opérateur de concaténation de chaîne	`&`
Symboles d'opérateur d'équivalence	`!= == >= <=`
Symboles de délimitation de chaîne	`" '`
Symboles de structure générique/d'appariement	`? *`
Symbole d'affectation	`:=`
Affectation d'une opération de communication	`->`
Valeurs de chaîne binaire, de chaîne hexadécimale et de chaîne d'octets	`B H O`
Exposant à virgule flottante	`E`

Les identificateurs de fonctions prédéfinies qui sont énumérés dans le Tableau 10 et décrits dans l'Annexe C doivent également être traités comme des mots réservés.

Tableau A.3/Z.140 – Liste des symboles terminaux TTCN-3 qui sont des mots réservés

action	fail	noblock	select
activate	false	none	self
address	float	not	send
alive	for	not4b	sender
all	from	nowait	set
alt	function	null	setverdict
altstep			signature
and	getverdict	octetstring	start
and4b	getcall	of	stop
any	getreply	omit	subset
anytype	goto	on	superset
	group	optional	system
bitstring		or	
boolean	hexstring	or4b	template
		out	testcase
case	if	override	timeout
call	ifpresent		timer
catch	import	param	to
char	in	pass	trigger
charstring	inconc	pattern	true
check	infinity	port	type
clear	inout	procedure	
complement	integer		union
component	interleave	raise	universal
connect		read	unmap
const	kill	receive	
control	killed	record	value
create			valueof
	label	rem	var
deactivate	language	repeat	variant
default	length	reply	verdicttype
disconnect	log	return	
display		running	while
do	map	runs	with
done	match		
	message		xor
else	mixed		xor4b
encode	mod		
enumerated	modifies		
error	module		
except	modulepar		
exception	mtc		
execute			
extends			
extension			
external			

Les symboles terminaux de la notation TTCN-3, énumérés dans le Tableau A.3, ne doivent pas être utilisés comme identificateurs dans un module TTCN-3. Ces symboles terminaux doivent être écrits entièrement en lettres minuscules.

A.1.6 Productions BNF de la syntaxe TTCN-3

A.1.6.0 Module TTCN-3

```
1. TTCN3Module ::= TTCN3ModuleKeyword TTCN3ModuleId
                   "{"
                   [ModuleDefinitionsPart]
                   [ModuleControlPart]
                   "}"
                   [WithStatement] [SemiColon]
2. TTCN3ModuleKeyword ::= "module"
3. TTCN3ModuleId ::= ModuleId
4. ModuleId ::= GlobalModuleId [LanguageSpec]
/* SEMANTIQUE STATIQUE - LanguageSpec ne peut être omis que si le module référencé contient la
/* notation TTCN-3 */
5. GlobalModuleId ::= ModuleIdentifier
6. ModuleIdentifier ::= Identifier
7. LanguageSpec ::= LanguageKeyword FreeText
8. LanguageKeyword ::= "language"
```

A.1.6.1 Partie d'un module relative aux définitions

A.1.6.1.0 Généralités

```
9. ModuleDefinitionsPart ::= ModuleDefinitionsList
10. ModuleDefinitionsList ::= {ModuleDefinition [SemiColon]}+
11. ModuleDefinition ::= (TypeDef |
                          ConstDef |
                          TemplateDef |
                          ModuleParDef |
                          FunctionDef |
                          SignatureDef |
                          TestcaseDef |
                          AltstepDef |
                          ImportDef |
                          GroupDef |
                          ExtFunctionDef |
                          ExtConstDef) [WithStatement]
```

A.1.6.1.1 Définitions de TypeDef

```
12. TypeDef ::= TypeDefKeyword TypeDefBody
13. TypeDefBody ::= StructuredTypeDef | SubTypeDef
14. TypeDefKeyword ::= "type"
15. StructuredTypeDef ::= RecordDef |
                          UnionDef |
                          SetDef |
                          RecordOfDef |
                          SetOfDef |
                          EnumDef |
                          PortDef |
                          ComponentDef
16. RecordDef ::= RecordKeyword StructDefBody
17. RecordKeyword ::= "record"
18. StructDefBody ::= (StructTypeIdentifier [StructDefFormalParList] | AddressKeyword)
                      "{" [StructFieldDef {"," StructFieldDef}] "}"
19. StructTypeIdentifier ::= Identifier
20. StructDefFormalParList ::= "(" StructDefFormalPar {"," StructDefFormalPar} ")"
21. StructDefFormalPar ::=  FormalValuePar
/* SEMANTIQUE STATIQUE - FormalValuePar doit se réduire à un paramètre de type in */
22. StructFieldDef ::= (Type | NestedTypeDef) StructFieldIdentifier [ArrayDef] [SubTypeSpec]
                       [OptionalKeyword]
23. NestedTypeDef ::= NestedRecordDef |
                      NestedUnionDef |
                      NestedSetDef |
                      NestedRecordOfDef |
                      NestedSetOfDef |
                      NestedEnumDef
24. NestedRecordDef ::= RecordKeyword "{" [StructFieldDef {"," StructFieldDef}] "}"
25. NestedUnionDef ::= UnionKeyword "{" UnionFieldDef {"," UnionFieldDef} "}"
26. NestedSetDef ::= SetKeyword "{" [StructFieldDef {"," StructFieldDef}] "}"
27. NestedRecordOfDef ::= RecordKeyword [StringLength] OfKeyword (Type | NestedTypeDef)
28. NestedSetOfDef ::= SetKeyword [StringLength] OfKeyword (Type | NestedTypeDef)
29. NestedEnumDef ::= EnumKeyword "{" EnumerationList "}"
30. StructFieldIdentifier ::= Identifier
31. OptionalKeyword ::= "optional"
32. UnionDef ::= UnionKeyword UnionDefBody
33. UnionKeyword ::= "union"
34. UnionDefBody ::= (StructTypeIdentifier [StructDefFormalParList] | AddressKeyword)
```

```
                     "{" UnionFieldDef {"," UnionFieldDef} "}"
35. UnionFieldDef ::= (Type | NestedTypeDef) StructFieldIdentifier [ArrayDef] [SubTypeSpec]
36. SetDef ::= SetKeyword StructDefBody
37. SetKeyword ::= "set"
38. RecordOfDef ::= RecordKeyword [StringLength] OfKeyword StructOfDefBody
39. OfKeyword ::= "of"
40. StructOfDefBody ::= (Type | NestedTypeDef) (StructTypeIdentifier | AddressKeyword) [SubTypeSpec]
41. SetOfDef ::= SetKeyword [StringLength] OfKeyword StructOfDefBody
42. EnumDef ::= EnumKeyword (EnumTypeIdentifier | AddressKeyword)
                "{" EnumerationList "}"
43. EnumKeyword ::= "enumerated"
44. EnumTypeIdentifier ::= Identifier
45. EnumerationList ::= Enumeration {"," Enumeration}
46. Enumeration ::= EnumerationIdentifier ["("[Minus] Number ")"]
47. EnumerationIdentifier ::= Identifier
48. SubTypeDef ::= Type (SubTypeIdentifier | AddressKeyword)  [ArrayDef] [SubTypeSpec]
49. SubTypeIdentifier ::= Identifier
50. SubTypeSpec ::= AllowedValues [StringLength] | StringLength
/* SEMANTIQUE STATIQUE - Les valeurs AllowedValues doiventre être du même type que le champ qui est
en cours de sous-typage */
51. AllowedValues ::= "(" (ValueOrRange {"," ValueOrRange}) | CharStringMatch ")"
52. ValueOrRange ::= RangeDef | ConstantExpression
/* SEMANTIQUE STATIQUE - La production RangeDef ne doit être utilisée qu'avec les types integer,
charstring, universal charstring ou float */
/* SEMANTIQUE STATIQUE - Lors du sous-typage d'un type charstring ou universal charstring, l'étendue
et les valeurs ne doivent pas être mélangées dans la même spécification de sous-type, SubTypeSpec */
53. RangeDef ::= LowerBound ".." UpperBound
54. StringLength ::= LengthKeyword "(" SingleConstExpression [".." UpperBound] ")"
/* SEMANTIQUE STATIQUE - La production StringLength ne doit être utilisée qu'avec des types
concaténés ou qu'afin de limiter des types set of et record of. Les productions
SingleConstExpression et UpperBound doivent prendre des valeurs d'entier non négatives (en cas de
limite supérieure UpperBound incluant l'infini) */
55.  LengthKeyword ::= "length"
56. PortType ::= [GlobalModuleId Dot] PortTypeIdentifier
57. PortDef ::= PortKeyword PortDefBody
58. PortDefBody ::= PortTypeIdentifier PortDefAttribs
59. PortKeyword ::= "port"
60. PortTypeIdentifier ::= Identifier
61. PortDefAttribs ::= MessageAttribs | ProcedureAttribs | MixedAttribs
62. MessageAttribs ::= MessageKeyword
                      "{" {MessageList [SemiColon]}+ "}"
63. MessageList ::= Direction AllOrTypeList
64. Direction ::= InParKeyword | OutParKeyword | InOutParKeyword
65. MessageKeyword ::= "message"
66. AllOrTypeList ::= AllKeyword | TypeList
/* NOTE: L'utilisation de AllKeyword dans les définitions de port est déconseillée */
67. AllKeyword ::= "all"
68. TypeList ::= Type {"," Type}
69. ProcedureAttribs ::= ProcedureKeyword
                       "{" {ProcedureList  [SemiColon]}+ "}"
70. ProcedureKeyword ::= "procedure"
71. ProcedureList ::= Direction AllOrSignatureList
72. AllOrSignatureList ::= AllKeyword | SignatureList
73. SignatureList ::= Signature {"," Signature}
74. MixedAttribs ::= MixedKeyword
                     "{" {MixedList [SemiColon]}+ "}"
75. MixedKeyword ::= "mixed"
76. MixedList ::= Direction ProcOrTypeList
77. ProcOrTypeList ::= AllKeyword | (ProcOrType {"," ProcOrType})
78. ProcOrType ::= Signature | Type
79. ComponentDef ::= ComponentKeyword ComponentTypeIdentifier
                     [ExtendsKeyword ComponentType {"," ComponentType}]
                     "{" [ComponentDefList] "}"
80. ComponentKeyword ::= "component"
81. ExtendsKeyword ::= "extends"
82. ComponentType ::= [GlobalModuleId Dot] ComponentTypeIdentifier
83. ComponentTypeIdentifier ::= Identifier
84. ComponentDefList ::= {ComponentElementDef [SemiColon]}
85. ComponentElementDef ::= PortInstance | VarInstance | TimerInstance | ConstDef
86. PortInstance ::= PortKeyword PortType PortElement {"," PortElement}
87. PortElement ::= PortIdentifier [ArrayDef]
88. PortIdentifier ::= Identifier
```

A.1.6.1.2 Définitions de constante

```
89. ConstDef ::= ConstKeyword Type ConstList
/* SEMANTIQUE STATIQUE - Le type doit suivre les règles indiquées dans le § 9.*/
90. ConstList ::= SingleConstDef  {"," SingleConstDef}
91. SingleConstDef ::= ConstIdentifier [ArrayDef] AssignmentChar ConstantExpression
```

```
/* SEMANTIQUE STATIQUE - La valeur de la production ConstantExpression doit être du même type que
celui qui est indiqué pour les constantes */
92. ConstKeyword ::= "const"
93. ConstIdentifier ::= Identifier
```

A.1.6.1.3 Définitions de modèle

```
94. TemplateDef ::= TemplateKeyword BaseTemplate  [DerivedDef] AssignmentChar TemplateBody
95. BaseTemplate ::= (Type | Signature) TemplateIdentifier  ["(" TemplateFormalParList ")"]
96. TemplateKeyword ::= "template"
97. TemplateIdentifier ::= Identifier
98. DerivedDef ::= ModifiesKeyword TemplateRef
99. ModifiesKeyword ::= "modifies"
100. TemplateFormalParList ::= TemplateFormalPar {"," TemplateFormalPar}
101. TemplateFormalPar ::= FormalValuePar | FormalTemplatePar
/* SEMANTIQUE STATIQUE - La production FormalValuePar doit correspondre à un paramètre de type in */
102. TemplateBody ::= (SimpleSpec | FieldSpecList | ArrayValueOrAttrib) | [ExtraMatchingAttributes]
/* SEMANTIQUE STATIQUE - A l'intérieur de la production TemplateBody, la production
ArrayValueOrAttrib peut être utilisée pour les types array, record, record of et set of. */
103. SimpleSpec ::= SingleValueOrAttrib
104. FieldSpecList ::= "{"[FieldSpec {"," FieldSpec}] "}"
105. FieldSpec ::= FieldReference AssignmentChar TemplateBody
106. FieldReference ::= StructFieldRef | ArrayOrBitRef | ParRef
/* SEMANTIQUE STATIQUE - A l'intérieur de la production FieldReference, la production ArrayOrBitRef
ne peut être utilisée que pour les modèles/champs de modèle record of et set of */
107. StructFieldRef ::= StructFieldIdentifier| PredefinedType | TypeReference
/* SEMANTIQUE STATIQUE - La production PredefinedType et la production TypeReference ne doivent être
utilisées que pour la notation de valeur anytype. La production PredefinedType ne doit pas être
AnyTypeKeyword.*/

108. ParRef ::= SignatureParIdentifier
/* SEMANTIQUE STATIQUE - La production SignatureParIdentifier doit être un paramètre formel
d'identification de la définition de signature associée */
109. SignatureParIdentifier ::= ValueParIdentifier
110. ArrayOrBitRef ::= "[" FieldOrBitNumber "]"
/* SEMANTIQUE STATIQUE - La production ArrayRef doit être facultativement utilisée pour les types
array, pour les types ASN.1 SET OF et SEQUENCE OF et pour les types TTCN-3 record of and set of. La
même notation peut servir à une référence binaire à l'intérieur d'un type "bitstring" en notation
ASN.1 ou TTCN-3 */
111. FieldOrBitNumber ::= SingleExpression
/* SEMANTIQUE STATIQUE - SingleExpression va correspondre à une valeur de type entier (integer) */
112. SingleValueOrAttrib ::= MatchingSymbol |
                             SingleExpression |
                             TemplateRefWithParList
/* SEMANTIQUE STATIQUE - La production VariableIdentifier (manipulée via singleExpression) ne peut
être utilisée que dans des définitions en ligne de modèle afin de faire référence à des variables
dans l'unité de portée actuelle */
113. ArrayValueOrAttrib ::= "{" ArrayElementSpecList "}"
114. ArrayElementSpecList ::= ArrayElementSpec {"," ArrayElementSpec}
115. ArrayElementSpec ::= NotUsedSymbol | PermutationMatch | TemplateBody
116. NotUsedSymbol ::= Dash
117. MatchingSymbol ::= Complement |
                        AnyValue |
                        AnyOrOmit |
                        ValueOrAttribList |
                        Range |
                        BitStringMatch |
                        HexStringMatch |
                        OctetStringMatch |
                        CharStringMatch |
                        SubsetMatch |
                        SupersetMatch
118. ExtraMatchingAttributes ::= LengthMatch | IfPresentMatch | LengthMatch  IfPresentMatch
119. BitStringMatch ::= "'" {BinOrMatch} "'" "B"
120. BinOrMatch ::= Bin | AnyValue | AnyOrOmit
121. HexStringMatch ::= "'" {HexOrMatch} "'" "H"
122. HexOrMatch ::= Hex | AnyValue | AnyOrOmit
123. OctetStringMatch ::= "'" {OctOrMatch} "'" "O"
124. OctOrMatch ::= Oct | AnyValue | AnyOrOmit
125. CharStringMatch ::= PatternKeyword Cstring
126. PatternKeyword ::= "pattern"
127. Complement ::= ComplementKeyword ValueList
128. ComplementKeyword ::= "complement"
129. ValueList ::= "(" ConstantExpression {"," ConstantExpression} ")"
130. SubsetMatch ::= SubsetKeyword ValueList
/* SEMANTIQUE STATIQUE - Le sous-ensemble correspondant ne doit être utilisé qu'avec le type set of
*/
131. SubsetKeyword ::= "subset"
132. SupersetMatch ::= SupersetKeyword ValueList
```

```
/* SEMANTIQUE STATIQUE - L'appariement d'un type Superset ne doit être utilisé qu'avec le type "set
of" */
133. SupersetKeyword ::= "superset"
134. PermutationMatch ::= PermutationKeyword PermutationList
135. PermutationKeyword ::= "permutation"
136. PermutationList ::= "(" TemplateBody { "," TemplateBody } ")"
/* SEMANTIQUE STATIQUE – Les restrictions applicables au contenu de TemplateBody sont indiquées dans
le § B.1.3.3 */
137. AnyValue ::= "?"
138. AnyOrOmit ::= "*"
139. ValueOrAttribList ::= "(" TemplateBody {"," TemplateBody}+ ")"
140. LengthMatch ::= StringLength
141. IfPresentMatch ::= IfPresentKeyword
142. IfPresentKeyword ::= "ifpresent"
143. Range ::= "(" LowerBound ".." UpperBound ")"
144. LowerBound ::= SingleConstExpression | Minus InfinityKeyword
145. UpperBound ::= SingleConstExpression | InfinityKeyword
/* SEMANTIQUE STATIQUE - Les productions LowerBound et UpperBound doivent s'évaluer comme des types
integer, charstring, universal charstring ou float. Si LowerBound ou UpperBound s'évalue comme un
type charstring ou universal charstring, seule la production SingleConstExpression peut être
présente et la longueur de chaîne doit être de 1 */
146. InfinityKeyword ::= "infinity"
147. TemplateInstance ::= InLineTemplate
148. TemplateRefWithParList ::= [GlobalModuleId Dot] TemplateIdentifier [TemplateActualParList] |
                               TemplateParIdentifier
149. TemplateRef ::= [GlobalModuleId Dot] TemplateIdentifier | TemplateParIdentifier
150. InLineTemplate ::= [(Type | Signature) Colon] [DerivedRefWithParList AssignmentChar]
                       TemplateBody
/* SÉMANTIQUE STATIQUE – Le champ de type ne peut être omis que quand le type est implicitement
univoque */
151. DerivedRefWithParList ::= ModifiesKeyword TemplateRefWithParList
152. TemplateActualParList ::= "(" TemplateActualPar {"," TemplateActualPar} ")"
153. TemplateActualPar ::= TemplateInstance
/* SÉMANTIQUE STATIQUE – Quand le paramètre formel correspondant n'est pas de type "template", la
production TemplateInstance doit correspondre à une ou plusieurs productions SingleExpressions */
154. TemplateOps ::= MatchOp | ValueofOp
155. MatchOp ::= MatchKeyword "(" Expression "," TemplateInstance")"
/* SÉMANTIQUE STATIQUE – Le type de la valeur retournée par l'expression doit impérativement être le
même que le type du modèle et chaque champ de celui-ci doit correspondre à une valeur unique */
156. MatchKeyword ::= "match"
157. ValueofOp ::= ValueofKeyword "(" TemplateInstance")"
158. ValueofKeyword ::= "valueof"
```

A.1.6.1.4 Définition de fonctions

```
159. FunctionDef ::= FunctionKeyword FunctionIdentifier
                     "("[FunctionFormalParList] ")" [RunsOnSpec] [ReturnType]
                     StatementBlock
160. FunctionKeyword ::= "function"
161. FunctionIdentifier ::= Identifier
162. FunctionFormalParList ::= FunctionFormalPar {"," FunctionFormalPar}
163. FunctionFormalPar ::= FormalValuePar |
                          FormalTimerPar |
                          FormalTemplatePar |
                          FormalPortPar
164. ReturnType ::= ReturnKeyword [TemplateKeyword] Type
/* SÉMANTIQUE STATIQUE - L'utilisation du mot clé modèle doit être conforme aux restrictions
indiquées dans le § 16.1.0 */
165. ReturnKeyword ::= "return"
166. RunsOnSpec ::= RunsKeyword OnKeyword ComponentType
167. RunsKeyword ::= "runs"
168. OnKeyword ::= "on"
169. MTCKeyword ::= "mtc"
170. StatementBlock ::= "{" [FunctionStatementOrDefList] "}"
171. FunctionStatementOrDefList ::= {FunctionStatementOrDef [SemiColon]}+
172. FunctionStatementOrDef ::= FunctionLocalDef |
                               FunctionLocalInst |
                               FunctionStatement
173. FunctionLocalInst ::= VarInstance | TimerInstance
174. FunctionLocalDef ::= ConstDef | TemplateDef
175. FunctionStatement ::= ConfigurationStatements |
                          TimerStatements |
                          CommunicationStatements |
                          BasicStatements |
                          BehaviourStatements |
                          VerdictStatements |
                          SUTStatements
176. FunctionInstance ::= FunctionRef "(" [FunctionActualParList] ")"
177. FunctionRef ::= [GlobalModuleId Dot] (FunctionIdentifier | ExtFunctionIdentifier ) |
                     PreDefFunctionIdentifier
```

```
178. PreDefFunctionIdentifier ::= Identifier
/* SÉMANTIQUE STATIQUE – L'identificateur sera un des identificateurs de fonction prédéfinis en
notation TTCN-3 d'après l'Annexe C */
179. FunctionActualParList ::= FunctionActualPar {"," FunctionActualPar}
180. FunctionActualPar ::= TimerRef |
                           TemplateInstance |
                           Port |
                           ComponentRef
/* SÉMANTIQUE STATIQUE – Quand le paramètre formel correspondant n'est pas de type template, la
production TemplateInstance doit correspondre à une ou plusieurs productions SingleExpression c'est-
à-dire qu'elle doit être équivalente à la production Expression */
```

A.1.6.1.5 Définitions de signature

```
181. SignatureDef ::= SignatureKeyword SignatureIdentifier
                      "("[SignatureFormalParList] ")" [ReturnType | NoBlockKeyword]
                      [ExceptionSpec]
182. SignatureKeyword ::= "signature"
183. SignatureIdentifier ::= Identifier
184. SignatureFormalParList ::= SignatureFormalPar {"," SignatureFormalPar}
185. SignatureFormalPar ::=  FormalValuePar
186. ExceptionSpec ::= ExceptionKeyword "(" ExceptionTypeList ")"
187. ExceptionKeyword ::= "exception"
188. ExceptionTypeList ::= Type {"," Type}
189. NoBlockKeyword ::= "noblock"
190. Signature ::= [GlobalModuleId Dot] SignatureIdentifier
Signature ::= [GlobalModuleId Dot] SignatureIdentifier
```

A.1.6.1.6 Définitions de test élémentaire

```
191. TestcaseDef ::= TestcaseKeyword TestcaseIdentifier
                     "("[TestcaseFormalParList] ")" ConfigSpec
                     StatementBlock
192. TestcaseKeyword ::= "testcase"
193. TestcaseIdentifier ::= Identifier
194. TestcaseFormalParList ::= TestcaseFormalPar {"," TestcaseFormalPar}
195. TestcaseFormalPar ::= FormalValuePar |
                           FormalTemplatePar
196. ConfigSpec ::= RunsOnSpec [SystemSpec]
197. SystemSpec ::= SystemKeyword ComponentType
198. SystemKeyword ::= "system"
199. TestcaseInstance ::= ExecuteKeyword "(" TestcaseRef "(" [TestcaseActualParList] ")"
                          ["," TimerValue] ")"
200. ExecuteKeyword ::= "execute"
201. TestcaseRef ::= [GlobalModuleId Dot] TestcaseIdentifier
202. TestcaseActualParList ::= TestcaseActualPar {"," TestcaseActualPar}
203. TestcaseActualPar ::= TemplateInstance
/* SÉMANTIQUE STATIQUE – Quand le paramètre formel correspondant n'est pas de type "template", la
production TemplateInstance doit correspondre à une ou plusieurs productions SingleExpression c'est-
à-dire qu'elle doit être équivalente à la production Expression */
```

A.1.6.1.7 Définitions de variante

```
204. AltstepDef ::= AltstepKeyword AltstepIdentifier
                    "("[AltstepFormalParList] ")" [RunsOnSpec]
                    "{" AltstepLocalDefList AltGuardList "}"
205. AltstepKeyword ::= "altstep"
206. AltstepIdentifier ::= Identifier
207. AltstepFormalParList ::= FunctionFormalParList
/* SÉMANTIQUE STATIQUE – Les variantes qui sont activées par défaut ne doivent avoir que des
paramètres de type in, des paramètres de port ou des paramètres de temporisation */
/* SÉMANTIQUE STATIQUE - Les variantes qui ne sont invoqués que comme une option dans une
inscription alt ou que comme une instruction autonome dans une description de comportement TTCN-3
peuvent avoir des paramètres in, out et inout. */
208. AltstepLocalDefList ::= {AltstepLocalDef [SemiColon]}
209. AltstepLocalDef ::= VarInstance | TimerInstance | ConstDef | TemplateDef
/* SÉMANTIQUE STATIQUE - AltstepLocalDef shall conform to restrictions in clause 16.2.2.1 */
210. AltstepInstance ::= AltstepRef "(" [FunctionActualParList] ")"
/* SÉMANTIQUE STATIQUE - Toutes les instances de temporisation contenues dans FunctionActualParList
doivent être déclarées en tant que temporisations localisées (voir aussi la production
ComponentElementDef) */
211. AltstepRef ::= [GlobalModuleId Dot] AltstepIdentifier
```

A.1.6.1.8 Définitions d'importation

```
212. ImportDef ::= ImportKeyword ImportFromSpec (AllWithExcepts | ("{" ImportSpec "}"))
213. ImportKeyword ::= "import"
214. AllWithExcepts ::= AllKeyword [ExceptsDef]
```

```
215. ExceptsDef ::= ExceptKeyword "{" ExceptSpec "}"
216. ExceptKeyword ::= "except"
217. ExceptSpec ::= {ExceptElement [SemiColon]}
/* SÉMANTIQUE STATIQUE - Chacun des composants d'une production (ExceptGroupSpec, ExceptTypeDefSpec
etc.) ne peut être présent qu'une seule fois dans la production ExceptSpec */
218. ExceptElement ::= ExceptGroupSpec |
                      ExceptTypeDefSpec |
                      ExceptTemplateSpec |
                      ExceptConstSpec |
                      ExceptTestcaseSpec |
                      ExceptAltstepSpec |
                      ExceptFunctionSpec |
                      ExceptSignatureSpec |
                      ExceptModuleParSpec
219. ExceptGroupSpec ::= GroupKeyword (ExceptGroupRefList | AllKeyword)
220. ExceptTypeDefSpec ::= TypeDefKeyword (TypeRefList | AllKeyword)
221. ExceptTemplateSpec ::= TemplateKeyword (TemplateRefList | AllKeyword)
222. ExceptConstSpec ::= ConstKeyword (ConstRefList | AllKeyword)
223. ExceptTestcaseSpec ::= TestcaseKeyword (TestcaseRefList | AllKeyword)
224. ExceptAltstepSpec ::= AltstepKeyword (AltstepRefList | AllKeyword)
225. ExceptFunctionSpec ::= FunctionKeyword (FunctionRefList | AllKeyword)
226. ExceptSignatureSpec ::= SignatureKeyword  (SignatureRefList | AllKeyword)
227. ExceptModuleParSpec ::= ModuleParKeyword  (ModuleParRefList | AllKeyword)
228. ImportSpec ::= {ImportElement [SemiColon]}
229. ImportElement ::= ImportGroupSpec |
                      ImportTypeDefSpec |
                      ImportTemplateSpec |
                      ImportConstSpec |
                      ImportTestcaseSpec |
                      ImportAltstepSpec |
                      ImportFunctionSpec |
                      ImportSignatureSpec |
                      ImportModuleParSpec
230. ImportFromSpec ::= FromKeyword ModuleId [RecursiveKeyword]
/* NOTE - L'utilisation de RecursiveKeyword est déconseillée */
231. RecursiveKeyword ::= "recursive"
232. ImportGroupSpec ::= GroupKeyword (GroupRefListWithExcept | AllGroupsWithExcept)
233. GroupRefList ::= FullGroupIdentifier {"," FullGroupIdentifier}
234. GroupRefListWithExcept ::= FullGroupIdentifierWithExcept {"," FullGroupIdentifierWithExcept}
235. AllGroupsWithExcept ::= AllKeyword [ExceptKeyword GroupRefList]
236. FullGroupIdentifier ::= GroupIdentifier {Dot GroupIdentifier}
237. FullGroupIdentifierWithExcept ::= FullGroupIdentifier [ExceptsDef]
238. ExceptGroupRefList ::= ExceptFullGroupIdentifier {"," ExceptFullGroupIdentifier}
239. ExceptFullGroupIdentifier ::= FullGroupIdentifier
240. ImportTypeDefSpec ::= TypeDefKeyword (TypeRefList | AllTypesWithExcept)
241. TypeRefList ::= TypeDefIdentifier {"," TypeDefIdentifier}
242. AllTypesWithExcept ::= AllKeyword [ExceptKeyword TypeRefList]
243. TypeDefIdentifier ::= StructTypeIdentifier |
                          EnumTypeIdentifier |
                          PortTypeIdentifier |
                          ComponentTypeIdentifier |
                          SubTypeIdentifier
244. ImportTemplateSpec ::= TemplateKeyword (TemplateRefList | AllTemplsWithExcept)
245. TemplateRefList ::= TemplateIdentifier {"," TemplateIdentifier}
246. AllTemplsWithExcept ::= AllKeyword [ExceptKeyword TemplateRefList]
247. ImportConstSpec ::= ConstKeyword (ConstRefList | AllConstsWithExcept)
248. ConstRefList ::= ConstIdentifier {"," ConstIdentifier}
249. AllConstsWithExcept ::= AllKeyword [ExceptKeyword ConstRefList]
250. ImportAltstepSpec ::= AltstepKeyword (AltstepRefList | AllAltstepsWithExcept)
251. AltstepRefList ::= AltstepIdentifier {"," AltstepIdentifier}
252. AllAltstepsWithExcept ::= AllKeyword [ExceptKeyword AltstepRefList]
253. ImportTestcaseSpec ::= TestcaseKeyword (TestcaseRefList | AllTestcasesWithExcept)
254. TestcaseRefList ::= TestcaseIdentifier {"," TestcaseIdentifier}
255. AllTestcasesWithExcept ::= AllKeyword [ExceptKeyword TestcaseRefList]
256. ImportFunctionSpec ::= FunctionKeyword (FunctionRefList | AllFunctionsWithExcept)
257. FunctionRefList ::= FunctionIdentifier {"," FunctionIdentifier}
258. AllFunctionsWithExcept ::= AllKeyword [ExceptKeyword FunctionRefList]
259. ImportSignatureSpec ::= SignatureKeyword  (SignatureRefList | AllSignaturesWithExcept)
260. SignatureRefList ::= SignatureIdentifier {"," SignatureIdentifier}
261. AllSignaturesWithExcept ::= AllKeyword [ExceptKeyword SignatureRefList]
262. ImportModuleParSpec ::= ModuleParKeyword  (ModuleParRefList | AllModuleParWithExcept)
263. ModuleParRefList ::= ModuleParIdentifier {"," ModuleParIdentifier}
264. AllModuleParWithExcept ::= AllKeyword [ExceptKeyword ModuleParRefList]
```

A.1.6.1.9 Définitions de groupe

```
265. GroupDef ::= GroupKeyword GroupIdentifier
                  "{" [ModuleDefinitionsPart] "}"
266. GroupKeyword ::= "group"
267. GroupIdentifier ::= Identifier
```

A.1.6.1.10 Définitions de fonction externe

```
268. ExtFunctionDef ::= ExtKeyword FunctionKeyword ExtFunctionIdentifier
                        "("[FunctionFormalParList] ")" [ReturnType]
269. ExtKeyword ::= "external"
270. ExtFunctionIdentifier ::= Identifier
```

A.1.6.1.11 Définitions de constante externe

```
271. ExtConstDef ::= ExtKeyword ConstKeyword Type ExtConstIdentifier
/* SÉMANTIQUE STATIQUE - Le type doit suivre les règles indiquées dans le § 9*/
272. ExtConstIdentifier ::= Identifier
```

A.1.6.1.12 Définitions de paramètre de module

```
273. ModuleParDef ::= ModuleParKeyword ( ModulePar | ("{" MultitypedModuleParList "}"))
274. ModuleParKeyword ::= "modulepar"
275. MultitypedModuleParList ::= { ModulePar SemiColon }+
276. ModulePar ::= ModuleParType ModuleParList
/* SÉMANTIQUE STATIQUE – La valeur de la production ConstantExpression doit être du même type que
celui qui est indiqué pour le paramètre */
277. ModuleParType ::= Type
/* SÉMANTIQUE STATIQUE - Le type ne doit pas être de type composant, par défaut ou anytype. Le type
ne doit correspondre qu'au type adresse si une définition pour ce est définie dans le module */
278. ModuleParList ::= ModuleParIdentifier [AssignmentChar ConstantExpression]
                      {","ModuleParIdentifier [AssignmentChar ConstantExpression]}
279. ModuleParIdentifier ::= Identifier
```

A.1.6.2 Partie commande

A.1.6.2.0 Généralités

```
280. ModuleControlPart ::= ControlKeyword
                           "{" ModuleControlBody "}"
                           [WithStatement] [SemiColon]
281. ControlKeyword ::= "control"
282. ModuleControlBody ::= [ControlStatementOrDefList]
283. ControlStatementOrDefList ::= {ControlStatementOrDef [SemiColon]}+
284. ControlStatementOrDef ::= FunctionLocalDef |
                               FunctionLocalInst |
                               ControlStatement
285. ControlStatement ::= TimerStatements |
                          BasicStatements |
                          BehaviourStatements |
                          SUTStatements |
                          StopKeyword
/* SÉMANTIQUE STATIQUE - Les restrictions applicables à l'utilisation d'instructions dans la partie
commande sont indiquées dans le Tableau 11 */
```

A.1.6.2.1 Instanciation de variable

```
286. VarInstance ::= VarKeyword ((Type VarList) | (TemplateKeyword Type TempVarList))
287. VarList ::= SingleVarInstance {"," SingleVarInstance}
288. SingleVarInstance ::= VarIdentifier [ArrayDef] [AssignmentChar VarInitialValue]
289. VarInitialValue ::= Expression
290. VarKeyword ::= "var"
291. VarIdentifier ::= Identifier
292. TempVarList ::= SingleTempVarInstance {"," SingleTempVarInstance}
293. SingleTempVarInstance ::= VarIdentifier [ArrayDef] [AssignmentChar TempVarInitialValue]
294. TempVarInitialValue ::= TemplateBody
295. VariableRef ::= (VarIdentifier | ValueParIdentifier) [ExtendedFieldReference]
```

A.1.6.2.2 Instanciation de temporisateur

```
296. TimerInstance ::= TimerKeyword TimerList
297. TimerList ::= SingleTimerInstance{"," SingleTimerInstance}
298. SingleTimerInstance ::= TimerIdentifier [ArrayDef] [AssignmentChar TimerValue]
299. TimerKeyword ::= "timer"
300. TimerIdentifier ::= Identifier
301. TimerValue ::= Expression
```

```
/* SÉMANTIQUE STATIQUE - Quand la production Expression se réduit à la production SingleExpression,
elle doit impérativement se réduire à une valeur de type float. La production Expression ne doit se
réduire à la production CompoundExpression que lors de l'initialisation d'une affectation de valeur
de temporisation par défaut pour des matrices de temporisation */
302. TimerRef ::= (TimerIdentifier | TimerParIdentifier) {ArrayOrBitRef}
```

A.1.6.2.3 Opérations sur composant

```
303. ConfigurationStatements ::= ConnectStatement |
                                 MapStatement |
                                 DisconnectStatement |
                                 UnmapStatement |
                                 DoneStatement |
                                 KilledStatement |
                                 StartTCStatement |
                                 StopTCStatement |
                                 KillTCStatement
304. ConfigurationOps ::= CreateOp | SelfOp | SystemOp | MTCOp | RunningOp | AliveOp
305. CreateOp ::= ComponentType Dot CreateKeyword ["(" SingleExpession ")"] [AliveKeyword]
/* SEMANTIQUE STATIQUE - Pour les restrictions applicables à SingleExpression, voir § 22.1 */
306. SystemOp ::= SystemKeyword
307. SelfOp ::= "self"
308. MTCOp ::= MTCKeyword
309. DoneStatement ::= ComponentId Dot DoneKeyword
310. KilledStatement ::= ComponentId Dot KilledKeyword
311. ComponentId ::= ComponentOrDefaultReference | (AnyKeyword | AllKeyword) ComponentKeyword
312. DoneKeyword ::= "done"
313. KilledKeyword ::= "killed"
314. RunningOp ::= ComponentId Dot RunningKeyword
315. RunningKeyword ::= "running"
316. AliveOp ::= ComponentId Dot AliveKeyword
317. CreateKeyword ::= "create"
318. AliveKeyword ::= "alive"
319. ConnectStatement ::= ConnectKeyword SingleConnectionSpec
320. ConnectKeyword ::= "connect"
321. SingleConnectionSpec ::= "(" PortRef "," PortRef ")"
322. PortRef ::= ComponentRef Colon Port
323. ComponentRef ::= ComponentOrDefaultReference | SystemOp | SelfOp | MTCOp
324. DisconnectStatement ::= DisconnectKeyword [SingleOrMultiConnectionSpec]
325. SingleOrMultiConnectionSpec ::= SingleConnectionSpec |
                                     AllConnectionsSpec |
                                     AllPortsSpec |
                                     AllCompsAllPortsSpec]
326. AllConnectionsSpec ::= "(" PortRef ")"
327. AllPortsSpec ::= "(" ComponentRef ":" AllKeyword PortKeyword ")"
328. AllCompsAllPortsSpec ::= "(" AllKeyword ComponentKeyword ":" AllKeyword PortKeyword ")"
329. DisconnectKeyword ::= "disconnect"
330. MapStatement ::= MapKeyword SingleConnectionSpec
331. MapKeyword ::= "map"
332. UnmapStatement ::= UnmapKeyword [SingleOrMultiConnectionSpec]
333. UnmapKeyword ::= "unmap"
334. StartTCStatement ::= ComponentOrDefaultReference Dot StartKeyword "(" FunctionInstance ")"
/* SÉMANTIQUE STATIQUE - L'instance de fonction ne peut avoir que des paramètres de type in */
/* SÉMANTIQUE STATIQUE - L'instance de fonction ne doit pas avoir de paramètres de temporisation */
335. StartKeyword ::= "start"
336. StopTCStatement ::= StopKeyword | (ComponentReferenceOrLiteral Dot StopKeyword) |
                        (AllKeyword ComponentKeyword Dot StopKeyword)
337. ComponentReferenceOrLiteral ::= ComponentOrDefaultReference | MTCOp | SelfOp
338. KillTCStatement ::= KillKeyword | (ComponentIdentifierOrLiteral Dot KillKeyword) |
                        (AllKeyword ComponentKeyword Dot KillKeyword)
339. ComponentOrDefaultReference ::= VariableRef | FunctionInstance
/* SÉMANTIQUE STATIQUE - La variable associée à VariableRef ou le type de retour associé à
FunctionInstance doit impérativement être de type component quand elle/il est utilisé(e) dans des
instructions de configuration et doit être de type par défaut quand elle/il est utilisé(e) dans
l'instruction de désactivation. */
340. KillKeyword ::= "kill"
```

A.1.6.2.4 Opérations sur les ports

```
341. Port ::= (PortIdentifier | PortParIdentifier) {ArrayOrBitRef}
342. CommunicationStatements ::= SendStatement |
                                 CallStatement |
                                 ReplyStatement |
                                 RaiseStatement |
                                 ReceiveStatement |
                                 TriggerStatement |
                                 GetCallStatement |
                                 GetReplyStatement |
                                 CatchStatement |
```

```
                                    CheckStatement |
                                    ClearStatement |
                                    StartStatement |
                                    StopStatement
343. SendStatement ::= Port Dot PortSendOp
344. PortSendOp ::= SendOpKeyword "(" SendParameter ")" [ToClause]
345. SendOpKeyword ::= "send"
346. SendParameter ::= TemplateInstance
347. ToClause ::= ToKeyword AddressRef |
                 AddressRefList |
                 AllKeyword ComponentKeyword
/* SÉMANTIQUE STATIQUE - AddressRef ne doit pas contenir de mécanismes appariés */
348. AddressRefList ::= "(" AddressRef {"," AddressRef} ")"
349. ToKeyword ::= "to"
350. AddressRef ::= TemplateInstance
/* SÉMANTIQUE STATIQUE – La production TemplateInstance doit impérativement être de type address ou
"component" */
351. CallStatement ::= Port Dot PortCallOp  [PortCallBody]
352. PortCallOp ::= CallOpKeyword "(" CallParameters ")" [ToClause]
353. CallOpKeyword ::= "call"
354. CallParameters ::= TemplateInstance ["," CallTimerValue]
/* SÉMANTIQUE STATIQUE - Seuls des paramètres de type out peuvent être omis ou spécifiés avec un
attribut apparié */
355. CallTimerValue ::= TimerValue | NowaitKeyword
/* SÉMANTIQUE STATIQUE - La production Value doit impérativement être de type float */
356. NowaitKeyword ::= "nowait"
357. PortCallBody ::= "{" CallBodyStatementList "}"
358. CallBodyStatementList ::= {CallBodyStatement [SemiColon]}+
359. CallBodyStatement ::= CallBodyGuard StatementBlock
360. CallBodyGuard ::= AltGuardChar CallBodyOps
361. CallBodyOps ::= GetReplyStatement | CatchStatement
362. ReplyStatement ::= Port Dot PortReplyOp
363. PortReplyOp ::= ReplyKeyword "(" TemplateInstance [ReplyValue]")" [ToClause]
364. ReplyKeyword ::= "reply"
365. ReplyValue ::= ValueKeyword Expression
366. RaiseStatement ::= Port Dot PortRaiseOp
367. PortRaiseOp ::= RaiseKeyword "(" Signature "," TemplateInstance ")" [ToClause]
368. RaiseKeyword ::= "raise"
369. ReceiveStatement ::= PortOrAny Dot PortReceiveOp
370. PortOrAny ::= Port | AnyKeyword PortKeyword
371. PortReceiveOp ::= ReceiveOpKeyword ["(" ReceiveParameter ")"] [FromClause] [PortRedirect]
/* SÉMANTIQUE STATIQUE - L'option PortRedirect ne peut être présente que si l'option
ReceiveParameter est également présente  */
372. ReceiveOpKeyword ::= "receive"
373. ReceiveParameter ::= TemplateInstance
374. FromClause ::= FromKeyword AddressRef
375. FromKeyword ::= "from"
376. PortRedirect ::= PortRedirectSymbol (ValueSpec [SenderSpec] | SenderSpec)
377. PortRedirectSymbol ::= "->"
378. ValueSpec ::= ValueKeyword VariableRef
379. ValueKeyword ::= "value"
380. SenderSpec ::= SenderKeyword VariableRef
/* SEMANTIQUE STATIQUE - La variable ref doit impérativement être de type address ou component */
381. SenderKeyword ::= "sender"
382. TriggerStatement ::= PortOrAny Dot PortTriggerOp
383. PortTriggerOp ::= TriggerOpKeyword ["(" ReceiveParameter ")"] [FromClause] [PortRedirect]
/* SÉMANTIQUE STATIQUE - L'option PortRedirect ne peut être présente que si l'option
ReceiveParameter est également présente */
384. TriggerOpKeyword ::= "trigger"
385. GetCallStatement ::= PortOrAny  Dot PortGetCallOp
386. PortGetCallOp ::= GetCallOpKeyword ["(" ReceiveParameter ")"] [FromClause]
                      [PortRedirectWithParam]
/* SÉMANTIQUE STATIQUE - L'option PortRedirectWithParam ne peut être présente que si l'option
ReceiveParameter est également présente */
387. GetCallOpKeyword ::= "getcall"
388. PortRedirectWithParam ::= PortRedirectSymbol RedirectWithParamSpec
389. RedirectWithParamSpec ::= ParamSpec [SenderSpec] |
                              SenderSpec
390. ParamSpec ::= ParamKeyword ParamAssignmentList
391. ParamKeyword ::= "param"
392. ParamAssignmentList ::= "(" (AssignmentList | VariableList) ")"
393. AssignmentList ::= VariableAssignment {"," VariableAssignment}
394. VariableAssignment ::= VariableRef AssignmentChar ParameterIdentifier
/* SÉMANTIQUE STATIQUE - La production parameterIdentifier doit impérativement être extraite de la
définition de signature correspondante */
395. ParameterIdentifier ::= ValueParIdentifier
396. VariableList ::= VariableEntry  {"," VariableEntry}
397. VariableEntry ::= VariableRef | NotUsedSymbol
398. GetReplyStatement ::= PortOrAny Dot PortGetReplyOp
399. PortGetReplyOp ::= GetReplyOpKeyword ["(" ReceiveParameter [ValueMatchSpec] ")"]
```

```
                        [FromClause] [PortRedirectWithValueAndParam]
/* SÉMANTIQUE STATIQUE - L'option PortRedirectWithParam ne peut être présente que si l'option
ReceiveParameter est également présente */
400. PortRedirectWithValueAndParam ::= PortRedirectSymbol RedirectWithValueAndParamSpec
401. RedirectWithValueAndParamSpec ::= ValueSpec [ParamSpec] [SenderSpec] |
                                      RedirectWithParamSpec
402. GetReplyOpKeyword ::= "getreply"
403. ValueMatchSpec ::= ValueKeyword TemplateInstance
404. CheckStatement ::= PortOrAny Dot PortCheckOp
405. PortCheckOp ::= CheckOpKeyword ["(" CheckParameter ")"]
406. CheckOpKeyword ::= "check"
407. CheckParameter ::= CheckPortOpsPresent | FromClausePresent | RedirectPresent
408. FromClausePresent ::= FromClause [PortRedirectSymbol SenderSpec]
409. RedirectPresent ::= PortRedirectSymbol SenderSpec
410. CheckPortOpsPresent ::= PortReceiveOp | PortGetCallOp | PortGetReplyOp | PortCatchOp
411. CatchStatement ::= PortOrAny Dot PortCatchOp
412. PortCatchOp ::= CatchOpKeyword ["("CatchOpParameter ")"] [FromClause] [PortRedirect]
/* SÉMANTIQUE STATIQUE - L'option PortRedirect ne peut être présente que si l'option
CatchOpParameter est également présente */
413. CatchOpKeyword ::= "catch"
414. CatchOpParameter ::= Signature "," TemplateInstance | TimeoutKeyword
415. ClearStatement ::= PortOrAll Dot PortClearOp
416. PortOrAll ::= Port | AllKeyword PortKeyword
417. PortClearOp ::= ClearOpKeyword
418. ClearOpKeyword ::= "clear"
419. StartStatement ::= PortOrAll Dot PortStartOp
420. PortStartOp ::= StartKeyword
421. StopStatement ::= PortOrAll Dot PortStopOp
422. PortStopOp ::= StopKeyword
423. StopKeyword ::= "stop"
424. AnyKeyword ::= "any"
```

A.1.6.2.5 Opérations de temporisation

```
425. TimerStatements ::= StartTimerStatement | StopTimerStatement | TimeoutStatement
426. TimerOps ::= ReadTimerOp | RunningTimerOp
427. StartTimerStatement ::= TimerRef Dot StartKeyword ["(" TimerValue ")"]
428. StopTimerStatement ::= TimerRefOrAll Dot StopKeyword
429. TimerRefOrAll ::= TimerRef | AllKeyword TimerKeyword
430. ReadTimerOp ::= TimerRef Dot ReadKeyword
431. ReadKeyword ::= "read"
432. RunningTimerOp ::= TimerRefOrAny  Dot RunningKeyword
433. TimeoutStatement ::= TimerRefOrAny  Dot TimeoutKeyword
434. TimerRefOrAny ::= TimerRef | AnyKeyword TimerKeyword
435. TimeoutKeyword ::= "timeout"
```

A.1.6.3 Type

```
436. Type ::= PredefinedType | ReferencedType
437. PredefinedType ::= BitStringKeyword |
                        BooleanKeyword |
                        CharStringKeyword |
                        UniversalCharString |
                        IntegerKeyword |
                        OctetStringKeyword |
                        HexStringKeyword |
                        VerdictTypeKeyword |
                        FloatKeyword |
                        AddressKeyword |
                        DefaultKeyword |
                        AnyTypeKeyword
438. BitStringKeyword ::= "bitstring"
439. BooleanKeyword ::= "boolean"
440. IntegerKeyword ::= "integer"
441. OctetStringKeyword ::= "octetstring"
442. HexStringKeyword ::= "hexstring"
443. VerdictTypeKeyword ::= "verdicttype"
444. FloatKeyword ::= "float"
445. AddressKeyword ::= "address"
446. DefaultKeyword ::= "default"
447. AnyTypeKeyword ::= "anytype"
448. CharStringKeyword ::= "charstring"
449. UniversalCharString ::= UniversalKeyword CharStringKeyword
450. UniversalKeyword ::= "universal"
451. ReferencedType ::= [GlobalModuleId Dot] TypeReference [ExtendedFieldReference]
452. TypeReference ::= StructTypeIdentifier[TypeActualParList] |
                       EnumTypeIdentifier |
                       SubTypeIdentifier |
                       ComponentTypeIdentifier
```

```
453. TypeActualParList ::= "(" TypeActualPar {"," TypeActualPar} ")"
454. TypeActualPar ::= ConstantExpression
455. ArrayDef ::= {"[" ArrayBounds [".." ArrayBounds] "]"}+
456. ArrayBounds ::= SingleConstExpression
/* SÉMANTIQUE STATIQUE – ArrayBounds se réduira à une valeur non négative de type entier (integer)
*/
```

A.1.6.4 Valeur

```
457. Value ::= PredefinedValue | ReferencedValue
458. PredefinedValue ::= BitStringValue |
                         BooleanValue |
                         CharStringValue |
                         IntegerValue |
                         OctetStringValue |
                         HexStringValue |
                         VerdictTypeValue |
                         EnumeratedValue |
                         FloatValue |
                         AddressValue |
                         OmitValue
459. BitStringValue ::= Bstring
460. BooleanValue ::= "true" | "false"
461. IntegerValue ::= Number
462. OctetStringValue ::= Ostring
463. HexStringValue ::= Hstring
464. VerdictTypeValue ::= "pass" | "fail" | "inconc" | "none" | "error"
465. EnumeratedValue ::= EnumerationIdentifier
466. CharStringValue ::= Cstring | Quadruple
467. Quadruple ::= CharKeyword "(" Group "," Plane "," Row "," Cell ")"
468. CharKeyword ::= "char"
469. Group ::= Number
470. Plane ::= Number
471. Row ::= Number
472. Cell ::= Number
473. FloatValue ::= FloatDotNotation | FloatENotation
474. FloatDotNotation ::= Number Dot DecimalNumber
475. FloatENotation ::= Number [Dot DecimalNumber] Exponential [Minus] Number
476. Exponential ::= "E"
477. ReferencedValue ::= ValueReference [ExtendedFieldReference]
478. ValueReference ::= [GlobalModuleId Dot] (ConstIdentifier | ExtConstIdentifier |
                        ModuleParIdentifier ) |
                        ValueParIdentifier |
                        VarIdentifier
479. Number ::= (NonZeroNum {Num}) | "0"
480. NonZeroNum ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
481. DecimalNumber ::= {Num}+
482. Num ::= "0" | NonZeroNum
483. Bstring ::= "'" {Bin} "'" "B"
484. Bin ::= "0" | "1"
485. Hstring ::= "'" {Hex} "'" "H"
486. Hex ::= Num | "A" | "B" | "C" | "D" | "E" | "F"| "a" | "b" | "c" | "d" | "e" | "f"
487. Ostring ::= "'" {Oct} "'" "O"
488. Oct ::= Hex Hex
489. Cstring ::= """ {Char} """
490. Char ::= /* RÉFÉRENCE – Il s'agit d'un caractère défini par le type applicable CharacterString.
Pour le type charString, il s'agit d'un caractère extrait du jeu de caractères défini dans la
Rec. UIT-T T.50. Pour le type UniversalCharstring, il s'agit d'un caractère extrait de tout jeu de
caractères défini dans l'ISO/CEI 10646 */
491. Identifier ::= Alpha{AlphaNum | Underscore}
492. Alpha ::= UpperAlpha | LowerAlpha
493. AlphaNum ::= Alpha | Num
494. UpperAlpha ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" |
"N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
495. LowerAlpha ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" |
"n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
496. ExtendedAlphaNum ::= /* RÉFÉRENCE – Il s'agit d'un caractère graphique extrait de jeu de
caractères BASIC LATIN ou LATIN-1 SUPPLEMENT défini dans l'ISO/CEI 10646  (caractères depuis char
(0,0,0,32) jusqu'à char (0,0,0,126), depuis char (0,0,0,161) jusqu'à char (0,0,0,172) et depuis char
(0,0,0,174) jusqu'à char (0,0,0,255) */
497. FreeText ::= """ {ExtendedAlphaNum} """
498. AddressValue ::= "null"
499. OmitValue ::= OmitKeyword
500. OmitKeyword ::= "omit"
```

A.1.6.5 Paramétrage

```
501. InParKeyword ::= "in"
502. OutParKeyword ::= "out"
```

```
503. InOutParKeyword ::= "inout"
504. FormalValuePar ::= [(InParKeyword | InOutParKeyword | OutParKeyword)] Type ValueParIdentifier
505. ValueParIdentifier ::= Identifier
506. FormalPortPar ::= [InOutParKeyword] PortTypeIdentifier PortParIdentifier
507. PortParIdentifier ::= Identifier
508. FormalTimerPar ::= [InOutParKeyword] TimerKeyword TimerParIdentifier
509. TimerParIdentifier ::= Identifier
510. FormalTemplatePar ::= [(InParKeyword | OutParKeyword | InOutParKeyword )]
                           TemplateKeyword Type TemplateParIdentifier
511. TemplateParIdentifier ::= Identifier
```

A.1.6.6 Instruction "with"

```
512. WithStatement ::= WithKeyword WithAttribList
513. WithKeyword ::= "with"
514. WithAttribList ::= "{" MultiWithAttrib "}"
515. MultiWithAttrib ::= {SingleWithAttrib [SemiColon]}
516. SingleWithAttrib ::= AttribKeyword  [OverrideKeyword] [AttribQualifier] AttribSpec
517. AttribKeyword ::= EncodeKeyword |
                       VariantKeyword |
                       DisplayKeyword |
                       ExtensionKeyword
518. EncodeKeyword ::= "encode"
519. VariantKeyword ::= "variant"
520. DisplayKeyword ::= "display"
521. ExtensionKeyword ::= "extension"
522. OverrideKeyword ::= "override"
523. AttribQualifier ::= "(" DefOrFieldRefList ")"
524. DefOrFieldRefList ::= DefOrFieldRef {"," DefOrFieldRef}
525. DefOrFieldRef ::= DefinitionRef | FieldReference | AllRef
/* SÉMANTIQUE STATIQUE – La production DefOrFieldRef doit impérativement se rapporter à une
définition ou à un champ qui est contenu dans le module, dans le groupe ou dans la définition auquel
l'instruction "with" est associée */
526. DefinitionRef ::= StructTypeIdentifier |
                       EnumTypeIdentifier |
                       PortTypeIdentifier |
                       ComponentTypeIdentifier |
                       SubTypeIdentifier |
                       ConstIdentifier |
                       TemplateIdentifier |
                       AltstepIdentifier |
                       TestcaseIdentifier |
                       FunctionIdentifier |
                       SignatureIdentifier |
                       VarIdentifier |
                       TimerIdentifier |
                       PortIdentifier |
                       ModuleParIdentifier |
                       FullGroupIdentifier
527. AllRef ::= ( GroupKeyword AllKeyword [ExceptKeyword "{" GroupRefList "}"]) |
                ( TypeDefKeyword AllKeyword [ExceptKeyword "{" TypeRefList "}"]) |
                ( TemplateKeyword AllKeyword [ExceptKeyword "{" TemplateRefList "}"]) |
                ( ConstKeyword AllKeyword [ExceptKeyword "{" ConstRefList "}"]) |
                ( AltstepKeyword AllKeyword [ExceptKeyword "{" AltstepRefList "}"]) |
                ( TestcaseKeyword AllKeyword [ExceptKeyword "{" TestcaseRefList "}"]) |
                ( FunctionKeyword AllKeyword [ExceptKeyword "{" FunctionRefList "}"]) |
                ( SignatureKeyword AllKeyword [ExceptKeyword "{" SignatureRefList "}"]) |
                ( ModuleParKeyword AllKeyword [ExceptKeyword "{" ModuleParRefList "}"])
528. AttribSpec ::= FreeText
```

A.1.6.7 Instructions comportementales

```
529. BehaviourStatements ::= TestcaseInstance |
                             FunctionInstance |
                             ReturnStatement |
                             AltConstruct |
                             InterleavedConstruct |
                             LabelStatement |
                             GotoStatement |
                             RepeatStatement |
                             DeactivateStatement |
                             AltstepInstance |
                             ActivateOp
/* SÉMANTIQUE STATIQUE – La production TestcaseInstance ne doit pas être appelée à partir de
l'intérieur d'un test élémentaire en cours d'exécution ni à partir d'une chaîne de fonctions appelée
à partir d'un test élémentaire, c'est-à-dire que les tests élémentaires ne peuvent être instanciés
qu'à partir de la partie commande ou qu'à partir de fonctions directement appelées à partir de la
partie commande */
```

```
/* SÉMANTIQUE STATIQUE - La production ActivateOp ne doit pas être appelée à partir de l'intérieur
de la partie commande du module */
530. VerdictStatements ::= SetLocalVerdict
531. VerdictOps ::= GetLocalVerdict
532. SetLocalVerdict ::= SetVerdictKeyword "(" SingleExpression ")"
/* SÉMANTIQUE STATIQUE - La production SingleExpression doit impérativement correspondre à une
valeur de type verdict */
/* SÉMANTIQUE STATIQUE – La production SetLocalVerdict ne doit pas servir à affecter l'erreur sur la
valeur */
533. SetVerdictKeyword ::= "setverdict"
534. GetLocalVerdict ::= "getverdict"
535. SUTStatements ::= ActionKeyword "(" [ActionText ] {StringOp ActionText} ")"
536. ActionKeyword ::= "action"
537. ActionText ::= FreeText | Expression
/* SÉMANTIQUE STATIQUE - La production Expression doit avoir le type de base "charstring" ou
"universal charstring" */
538. ReturnStatement ::= ReturnKeyword  [Expression]
539. AltConstruct ::= AltKeyword "{" AltGuardList "}"
540. AltKeyword ::= "alt"
541. AltGuardList ::= {GuardStatement | ElseStatement [SemiColon]}
542. GuardStatement ::= AltGuardChar (AltstepInstance [StatementBlock] | GuardOp StatementBlock)
543. ElseStatement ::= "["ElseKeyword "]" StatementBlock
544. AltGuardChar ::= "[" [BooleanExpression] "]"
/* SÉMANTIQUE STATIQUE - La production BooleanExpression doit être conforme aux restrictions
indiquées dans le § 20.1.2 */
545. GuardOp ::= TimeoutStatement |
                ReceiveStatement |
                TriggerStatement |
                GetCallStatement |
                CatchStatement |
                CheckStatement |
                GetReplyStatement |
                DoneStatement |
                KilledStatement
/* SÉMANTIQUE STATIQUE – La production GuardOp utilisée dans la partie d'un module relative à la
commande ne doit contenir que l'instruction timeoutStatement */
546. InterleavedConstruct ::= InterleavedKeyword "{" InterleavedGuardList "}"
547. InterleavedKeyword ::= "interleave"
548. InterleavedGuardList ::= {InterleavedGuardElement [SemiColon]}+
549. InterleavedGuardElement ::= InterleavedGuard InterleavedAction
550. InterleavedGuard ::= "[" "]" GuardOp
551. InterleavedAction ::= StatementBlock
/* SÉMANTIQUE STATIQUE – La production StatementBlock ne peut pas contenir d'instructions de type:
"loop, goto, activate, deactivate, stop, return" ni d'appels à des fonctions */
552. LabelStatement ::= LabelKeyword LabelIdentifier
553. LabelKeyword ::= "label"
554. LabelIdentifier ::= Identifier
555. GotoStatement ::= GotoKeyword LabelIdentifier
556. GotoKeyword ::= "goto"
557. RepeatStatement ::= "repeat"
558. ActivateOp ::= ActivateKeyword "(" AltstepInstance ")"
559. ActivateKeyword ::= "activate"
560. DeactivateStatement ::= DeactivateKeyword ["(" ComponentOrDefaultReference ")"]
561. DeactivateKeyword ::= "deactivate"
```

A.1.6.8 Instructions de base

```
562. BasicStatements ::= Assignment | LogStatement | LoopConstruct | ConditionalConstruct |
                         SelectCaseConstruct
563. Expression ::= SingleExpression | CompoundExpression
/* SÉMANTIQUE STATIQUE – Une expression ne doit pas contenir d'opérations de configuration,
d'opération d'activation ni d'opérations de verdict dans la partie d'un module relative à la
commande */
564. CompoundExpression ::= FieldExpressionList | ArrayExpression
/* SÉMANTIQUE STATIQUE – A l'intérieur de la production CompoundExpression, la production
ArrayExpression peut servir à des types "array, record, record of, set of". */
565. FieldExpressionList ::= "{" FieldExpressionSpec {"," FieldExpressionSpec} "}"
566. FieldExpressionSpec ::= FieldReference AssignmentChar NotUsedOrExpression
567. ArrayExpression ::= "{" [ArrayElementExpressionList] "}"
568. ArrayElementExpressionList ::= NotUsedOrExpression  {"," NotUsedOrExpression}
569. NotUsedOrExpression ::= Expression | NotUsedSymbol
570. ConstantExpression ::= SingleConstExpression | CompoundConstExpression
571. SingleConstExpression ::= SingleExpression
/* SÉMANTIQUE STATIQUE – La production SingleConstExpression ne doit pas contenir de variables ni de
paramètres de module et doit correspondre à une valeur de constante au moment de la compilation*/
572. BooleanExpression ::= SingleExpression
/* SÉMANTIQUE STATIQUE – BooleanExpression doit correspondre à une valeur de type Boolean */
573. CompoundConstExpression ::= FieldConstExpressionList | ArrayConstExpression
/* SÉMANTIQUE STATIQUE – A l'intérieur de la production CompoundConstExpression, la production
ArrayConstExpression peut servir à des types "array, record, record of, set of". */
```

```
Arrays, record, record of and set of types. */
574. FieldConstExpressionList ::= "{" FieldConstExpressionSpec {"," FieldConstExpressionSpec} "}"
575. FieldConstExpressionSpec ::= FieldReference AssignmentChar ConstantExpression
576. ArrayConstExpression ::= "{" [ArrayElementConstExpressionList] "}"
577. ArrayElementConstExpressionList ::= ConstantExpression {"," ConstantExpression}
578. Assignment ::= VariableRef AssignmentChar (Expression | TemplateBody)
/* SÉMANTIQUE STATIQUE – L'expression située dans la partie droite de la production Assignment doit
prendre une valeur explicite d'un type compatible avec celui de la partie gauche pour les variables
de valeur et doit prendre une valeur explicite, de modèle (de litéral ou d'une instance de modèle)
ou d'un mécanisme d'appariement compatible avec le type de la partie gauche pour les variables de
modèle. */
579. SingleExpression ::= XorExpression { "or" XorExpression }
/* SÉMANTIQUE STATIQUE – Si plusieurs productions XorExpression existent, alors la production
XorExpressions doit prendre des valeurs spécifiques de types compatibles. */
580. XorExpression ::= AndExpression { "xor" AndExpression }
/* SÉMANTIQUE STATIQUE – Si plusieurs productions AndExpression existent, alors la production
AndExpressions doit prendre des valeurs spécifiques de types compatibles. */
581. AndExpression ::= NotExpression { "and" NotExpression }
/* SÉMANTIQUE STATIQUE – Si plusieurs productions NotExpressions existent, alors la production
NotExpressions doit prendre des valeurs spécifiques de types compatibles. */
582. NotExpression ::= [ "not" ] EqualExpression
/* SÉMANTIQUE STATIQUE – Les opérandes de l'opérateur Not doivent être de type boléen (TTCN ou
ASN.1) ou des dérivés de type boléen. */
583. EqualExpression ::= RelExpression { EqualOp RelExpression }
/* SÉMANTIQUE STATIQUE – Si plusieurs productions RelExpression existent, alors la production
RelExpressions doit prendre des valeurs spécifiques de types compatibles. */
584. RelExpression ::= ShiftExpression [ RelOp ShiftExpression ]
/* SÉMANTIQUE STATIQUE – Si les deux productions ShiftExpressions existent, alors chaque production
ShiftExpression doit prendre la valeur d'un type spécifique: integer, enumerated ou float (ces
valeurs peuvent être en notation TTCN ou ASN.1) ou de dérivés de ces types. */
585. ShiftExpression ::= BitOrExpression { ShiftOp BitOrExpression }
/* SÉMANTIQUE STATIQUE – Chaque résultat doit correspondre à une valeur spécifique. Si plus d'un
seul résultat existe, l'opérande de la partie droite de type integer ou de types dérivés de
celui-ci. Si l'opérateur de décalage est '« ou »', alors l'opérande de la partie gauche doit
correspondre au type bitstring, hexstring ou octetstring ou à des dérivés de ces types. Si
l'opérateur de décalage est '<@' ou '@>', alors l'opérande de la partie gauche doit être de type
bitstring, hexstring, charstring ou universal charstring ou à des dérivés de ces types.  */
586. BitOrExpression ::= BitXorExpression { "or4b" BitXorExpression }
/* SÉMANTIQUE STATIQUE – Si plus d'une seule production BitXorExpression existe, alors la production
BitXorExpressions doit prendre des valeurs spécifiques de types compatibles. */
587. BitXorExpression ::= BitAndExpression { "xor4b" BitAndExpression }
/* SÉMANTIQUE STATIQUE – Si plus d'une seule production BitAndExpression existe, alors ces
productions BitAndExpression doivent prendre des valeurs spécifiques de types compatibles. */
588. BitAndExpression ::= BitNotExpression { "and4b" BitNotExpression }
/* SEMANTIQUE STATIQUE - Si plus d'une seule production BitNotExpression existe, alors ces
productions BitNotExpression doivent prendre des valeurs spécifiques de types compatibles. */
589. BitNotExpression ::= [ "not4b" ] AddExpression
/* SEMANTIQUE STATIQUE: Si l'opérateur not4b existe, l'opérande doit être de type bitstring,
octetstring ou hexstring ou de types dérivés de ces types.
590. AddExpression ::= MulExpression { AddOp MulExpression }
/* SEMANTIQUE STATIQUE: Chaque production MulExpression doit correspondre à une valeur spécifique.
Si plus d'uneseule production MulExpression existe et si l'opérateur d'addition correspond à
StringOp, alors les productions MulExpression doivent correspondre à un même type qui doit être:
bitstring, hexstring, octetstring, charstring ou universal charstring ou des types dérivés de ces
types. Si plus d'une seule production MulExpression existe et que l'opérateur d'addition ne
corresponde pas à StringOp, alors les productions MulExpression doivent toutes les deux correspondre
à un type integer ou float ou à des types dérivés de ces types.
591. MulExpression ::= UnaryExpression { MultiplyOp UnaryExpression }
/* SÉMANTIQUE STATIQUE – Chaque production UnaryExpression doit correspondre à une valeur
spécifique. Si plus d'une seule production UnaryExpression existe, alors ces productions
UnaryExpression doivent correspondre à un type integer ou float ou à des types dérivés de ces types.
592. UnaryExpression ::= [ UnaryOp ] Primary
/* SÉMANTIQUE STATIQUE – La production Primary doit correspondre à une valeur spécifique de type
"integer" ou "float" ou à des types dérivés de ces types.
593. Primary ::= OpCall | Value | "(" SingleExpression ")"
594. ExtendedFieldReference ::= {(Dot ( StructFieldIdentifier | TypeDefIdentifier))
                              | ArrayOrBitRef }+
/* SÉMANTIQUE STATIQUE – La production TypeDefIdentifier ne doit être utilisée que si le type de la
production VarInstance ou ReferencedValue dans laquelle la référence ExtendedFieldReference est
utilisée est: anytype. */
595. OpCall ::= ConfigurationOps |
                VerdictOps |
                TimerOps |
                TestcaseInstance |
                FunctionInstance |
                TemplateOps |
                ActivateOp
596. AddOp ::= "+" | "-" | StringOp
/* SÉMANTIQUE STATIQUE - Les opérandes des opérateurs "+" ou "-" doivent être de type integer ou
float ou être des dérivés du type integer ou float (c'est-à-dire être une sous-étendue). */
```

```
597. MultiplyOp ::= "*" | "/" | "mod" | "rem"
/* SÉMANTIQUE STATIQUE - Les opérandes des opérateurs "*" , "/", rem ou mod doivent être de type
integer ou float ou être des dérivés du type integer ou float (c'est-à-dire être une sous-étendue.
*/
598. UnaryOp ::= "+" | "-"
/* SÉMANTIQUE STATIQUE - Les opérandes des opérateurs "+" ou "-" doivent être de type integer ou
float ou être des dérivés du type integer ou float (c'est-à-dire être une sous-étendue. */
599. RelOp ::= "<" | ">" | ">=" | "<="
/* SÉMANTIQUE STATIQUE - La priorité des opérateurs est définie dans le Tableau 7. */
600. EqualOp ::= "==" | "!="
601. StringOp ::= "&"
/* SÉMANTIQUE STATIQUE – Les opérandes des opérateurs de chaîne doivent être de type bitstring,
hexstring, octetstring ou character string */
602. ShiftOp ::= "<<" | ">>" | "<@" | "@>"
603. LogStatement ::= LogKeyword "(" LogItem { "," LogItem } ")"
604. LogKeyword ::= "log"
605. LogItem ::= FreeText | TemplateInstance
606. LoopConstruct ::= ForStatement |
                     WhileStatement |
                     DoWhileStatement
607. ForStatement ::= ForKeyword "(" Initial SemiColon Final SemiColon Step ")"
                    StatementBlock
608. ForKeyword ::= "for"
609. Initial ::= VarInstance | Assignment
610. Final ::= BooleanExpression
611. Step ::= Assignment
612. WhileStatement ::= WhileKeyword "(" BooleanExpression ")"
                      StatementBlock
613. WhileKeyword ::= "while"
614. DoWhileStatement ::= DoKeyword StatementBlock
                        WhileKeyword "(" BooleanExpression ")"
615. DoKeyword ::= "do"
616. ConditionalConstruct ::= IfKeyword "(" BooleanExpression ")"
                            StatementBlock
                            {ElseIfClause}[ElseClause]
617. IfKeyword ::= "if"
618. ElseIfClause ::= ElseKeyword  IfKeyword "(" BooleanExpression ")"  StatementBlock
619. ElseKeyword ::= "else"
620. ElseClause ::= ElseKeyword  StatementBlock
621. SelectCaseConstruct ::= SelectKeyword "(" SingleExpression ")" SelectCaseBody
622. SelectKeyword ::= "select"
623. SelectCaseBody ::= "{" { SelectCase }+ "}"
624. SelectCase ::= CaseKeyword ( '(' TemplateInstance {"," TemplateInstance } ')' | ElseKeyword )
                  StatementBlock
625. CaseKeyword ::= "case"
```

A.1.6.9 Productions diverses

```
626. Dot ::= "."
627. Dash ::= "-"
628. Minus ::= Dash
629. SemiColon ::= ";"
630. Colon ::= ":"
631. Underscore ::= "_"
632. AssignmentChar ::= ":="
```

Annexe B

Appariement de valeurs entrantes

B.1 Mécanismes d'appariement de modèles

B.1.0 Généralités

La présente annexe spécifie les mécanismes d'appariement qui peuvent être utilisés dans les modèles de la notation TTCN-3 (et seulement dans ces modèles).

B.1.1 Appariement de valeurs spécifiques

Les valeurs spécifiques sont les mécanismes d'appariement de base des modèles TTCN-3. Les valeurs spécifiques qui sont contenues dans les modèles sont des expressions qui ne contiennent ni mécanismes d'appariement ni structures génériques. Sauf indication contraire, un champ de modèle s'apparie à la valeur de champ entrante correspondante si et seulement si la valeur de champ entrante est exactement identique à la valeur à laquelle l'expression contenue dans le modèle se réduit.

EXEMPLE:

```
// Si l'on a la définition de type de message
type record MyMessageType
{
  integer      field1,
  charstring   field2,
  boolean      field3 optional,
  integer[4]   field4
}

// Un modèle de message utilisant les valeurs spécifiques pourrait être
template MyMessageType MyTemplate:=
{
  field1 := 3+2,              // valeur spécifique de type entier
  field2 := "My string",      // valeur spécifique de type chaîne
                              // de caractères
  field3 := true,             // valeur spécifique de type boléen
  field4 := {1,2,3}           // valeur spécifique de matrice
                              / d'entiers
}
```

B.1.1.1 Omission de valeurs

Le mot clé **omit** indique qu'un champ facultatif de modèle doit être absent. Il peut être utilisé pour les valeurs de tous les types, à condition que le champ de modèle soit facultatif.

EXEMPLE:

```
template Mymessage MyTemplate:=
{ :
  :
  field3 := omit, // omettre ce champ
  :
}
```

B.1.2 Mécanismes d'appariement au lieu de valeurs

B.1.2.0 Généralités

Les mécanismes d'appariement suivants peuvent être utilisé à la place de valeurs explicites.

B.1.2.1 Liste de valeurs

Les listes de valeurs énumèrent des valeurs entrantes acceptables. Elles peuvent être utilisées avec des valeurs de tous types. Un champ de modèle qui utilise une liste de valeurs s'apparie avec le champ entrant correspondant si et seulement si, la valeur de champ entrante correspond à l'une quelconque des valeurs contenues dans la liste de valeurs. Chaque valeur contenue dans la liste de valeurs doit être du type déclaré pour le champ de modèle dans lequel ce mécanisme est utilisé.

EXEMPLE:

```
template Mymessage MyTemplate:=
{
  field1 := (2,4,6),               // liste de valeurs "integer"
  field2 := ("String1", "String2"),// liste de valeurs "charstring"
  :
  :
}
```

B.1.2.2 Liste de valeurs complémentées

Le mot clé **complement** indique une liste de valeurs qui ne seront pas acceptées en tant que valeurs entrantes (c'est-à-dire qu'il s'agit du complément à une liste de valeurs). Ce mot clé peut être utilisé pour toutes valeurs de tous types.

Chaque valeur contenue dans la liste doit être du type déclaré pour le champ de modèle dans lequel le complément est utilisé. Un champ de modèle qui utilise un complément s'apparie avec le champ entrant correspondant si et seulement si le champ entrant ne s'apparie avec aucune des valeurs énumérées dans la liste de valeurs. La liste de valeurs peut être une valeur isolée, évidemment.

EXEMPLE:

```
template Mymessage MyTemplate:=
{
  complement (1,3,5),  // liste de valeurs inacceptables d'entier
  :
  field3 not(true)     // s'appariera avec la valeur false
  :
}
```

B.1.2.3 Valeur quelconque

Le symbole d'appariement "?" (*AnyValue*) sert à indiquer que toute valeur entrante valide est acceptable. Il peut être utilisé avec des valeurs de tous types. Un champ de modèle qui utilise le mécanisme de valeur quelconque s'apparie avec le champ entrant correspondant si et seulement si, ce champ entrant se réduit à un élément isolé du type spécifié.

EXEMPLE:

```
template Mymessage MyTemplate:=
{
  field1 := ?,   // s'appariera avec tout entier
  field2 := ?,   // s'appariera avec toute valeur "charstring" non vide
  field3 := ?,   // s'appariera avec la valeur "true" ou "false"
  field4 := ?    // s'appariera avec toute séquence d'entiers
}
```

B.1.2.4 Valeur quelconque ou inexistante

Le symbole d'appariement "*" (*AnyValueOrNone*) sert à indiquer que toute valeur entrante valide, y compris l'omission de cette valeur, est acceptable. Il peut être utilisé avec des valeurs de tous types, à condition que le champ de modèle soit déclaré comme facultatif.

Un champ de modèle qui utilise ce symbole s'apparie avec le champ entrant correspondant si et seulement si, soit le champ entrant se réduit à un élément quelconque du type spécifié ou le champ entrant est absent.

EXEMPLE:

```
template Mymessage MyTemplate:=
{ :
  field3 := *,   // s'appariera avec true ou false ou avec un champ omis
  :
}
```

B.1.2.5 Etendue de valeurs

Les étendues indiquent un domaine borné de valeurs acceptables, quand elles sont utilisées pour des valeurs de type **integer** ou **float** (et leurs sous-types). A valeur limite doit être soit:

a) +l'infini ou –l'infini;

b) une expression qui se réduit à un entier ou un nombre en virgule flottante spécifique.

La limite inférieure doit être mise dans la partie gauche de l'étendue, la limite supérieure dans la partie droite. La limite inférieure doit être inférieure à la limite supérieure. Un champ de modèle qui utilise une étendue s'apparie avec le

champ entrant correspondant si et seulement si la valeur de champ entrante est égale à une des valeurs contenues dans l'étendue.

Quand elles sont utilisées dans des modèles ou dans des champs de modèle de type **`charstring`** ou **`universal charstring`**, les limites doivent prendre des positions de caractère valides, conformément à la (aux) table(s) de jeu codé de caractères du type (par exemple, la position indiquée ne doit pas être vide). Les positions vides entre les limites inférieure et supérieure ne sont pas considérées comme étant des valeurs valides de l'étendue spécifiée.

EXEMPLE:

```
template Mymessage MyTemplate:=
{
   field1 := (1 .. 6),  // étendue de type entier (integer)
   :
   :
   :
}
// d'autres entrées pour le champ field1 pourraient être (-infinity jusqu'à
// 8) ou (12 jusqu'à infinity)
```

B.1.2.6 Opération "SuperSet"

L'opération "SuperSet" est une opération d'appariement qui ne doit être appliquée qu'à des valeurs de type **`set of`**. L'opération **`superset`** est indiquée par le mot clé **`superset`**. Un champ qui utilise l'opération **`superset`** s'apparie avec le champ entrant correspondant si et seulement si ce champ entrant contient au moins tous les éléments définis dans le super-ensemble et s'il peut en contenir d'autres. L'argument de l'opération "SuperSet" doit être du type qui a été déclaré pour le champ dans lequel le mécanisme de surensemble est utilisé.

EXEMPLE:

```
type set of integer MySetOfType;

template MySetOfType MyTemplate1 := superset ( 1, 2, 3 );
// toute séquence d'entiers s'apparie avec celle qui contient au moins une
// occurrence des nombres 1, 2 et 3 dans un ordre ou emplacement quelconque
```

B.1.2.7 Opération "SubSet"

SubSet est une opération d'appariement qui ne peut être appliquée qu'à des valeurs de type **`set of`**. L'opération SubSet est indiquée par le mot clé **`subset`**.

Un champ qui utilise l'opération "SubSet" s'apparie avec le champ entrant correspondant si et seulement si ce champ entrant contient seulement les éléments définis dans le sous-ensemble et s'il peut en contenir moins. L'argument de l'opération "SubSet" doit être du type déclaré pour le champ dans lequel le mécanisme de sous-ensemble est utilisé.

EXEMPLE:

```
template MySetOfType MyTemplate1:= subset ( 1, 2, 3 );
// toute séquence d'entiers s'apparie avec celle qui contient zéro ou une
// seule occurrence des nombres 1, 2 et 3 dans un ordre ou emplacement
// quelconque
```

B.1.3 Mécanismes d'appariement à l'intérieur de valeurs

B.1.3.0 Généralités

Les mécanismes d'appariement suivants peuvent être utilisés à l'intérieur de valeurs explicites des types: string, record, record of, set, set of et des matrices.

B.1.3.1 Elément quelconque

Le symbole d'appariement "?" (*AnyElement*) sert à indiquer qu'il remplace des éléments isolés d'une chaîne (sauf les chaînes de caractères) ou d'un type **`record of, set of`** ou d'une matrice. Il ne doit être utilisé qu'à l'intérieur de valeurs de type **`string`**, **`record of`**, **`set of`** et des matrices.

EXEMPLE:

```
template Mymessage MyTemplate:=
{  :
   field2 := "abcxyz",
   field3 := '10???'B,  // où chaque "?" peut être 0 ou 1
   field4 := {1, ?, 3}  // où ? peut être toute valeur d'entier
}
```

NOTE – Le symbole "?" dans le champ `field4` peut être interprété comme une valeur quelconque (*AnyValue*), comme une valeur d'entier, ou comme un élément quelconque (*AnyElement*) à l'intérieur d'un type **`record of`**, **`set of`**, ou d'une matrice. Etant donné que les deux interprétations conduisent au même appariement, aucun problème ne se pose.

B.1.3.1.1 Utilisation de structures génériques à caractère unique

S'il est nécessaire d'exprimer la structure générique "?" sous forme de chaînes de caractères, cela doit être effectué au moyen de structures alphanumériques (voir § B.1.5). Par exemple, les chaînes "abcdxyz", "abccxyz", "abcxxyz", etc. correspondront toutes à la structure séquentielle "abc?xyz". Ce ne sera cependant pas le cas des chaînes "abcxyz", "abcdefxyz", etc.

B.1.3.2 Nombre quelconque d'éléments ou absence d'élément

Le symbole d'appariement "*" (*AnyElementsOrNone*) sert à indiquer qu'il remplace zéro ou *n* éléments consécutifs d'une chaîne (sauf les chaînes de caractères), ou d'un type **`record of, set of`** ou d'une matrice. Le symbole "*" correspond à la plus longue séquence d'éléments possible, conformément à la structure séquentielle qui est spécifiée par les symboles qui entourent le symbole "*".

EXEMPLE:

```
template Mymessage MyTemplate:=
{ :
   field2 := "abcxyz",
   field3 := '10*11'B,       // où "*" peut être toute séquence de bits (éventuellement vide)
   field4 := {*, 2, 3}       // où "*" peut être un nombre quelconque de valeurs d'entier
                              // ou être omis
}

var charstring MyStrings[4];
MyPCO.receive(MyStrings:{"abyz", *, "abc" });
```

Si un symbole "*" apparaît au niveau le plus élevé à l'intérieur d'une chaîne d'un type **`record of`**, **`set of`**, ou d'une matrice, il doit être interprété comme une structure *AnyElementsOrNone*.

NOTE – Cette règle empêche l'interprétation (sinon possible) de "*" en tant que structure *AnyValueOrNone* qui remplace un élément à l'intérieur d'une chaîne, d'un type **`record of`**, **`set of`**, ou d'une matrice.

B.1.3.2.1 Utilisation de structures génériques à caractères multiples

S'il est nécessaire d'exprimer une structure générique "*" dans des chaînes de caractères, cela doit être effectué au moyen de structures alphanumériques (voir § B.1.5). Par exemple, les chaînes "abcxyz", "abcdefxyz", "abcabcxyz", etc. correspondront toutes à la structure "abc*xyz".

B.1.3.3 Permutation

La permutation est une opération d'appariement qui ne doit être utilisée que pour les valeurs des types **`record of`**. La permutation est indiquée par le mot clé **`permutation`**. Les expressions et les symboles d'appariement *AnyElement* et *AnyElementsOrNone* sont autorisés comme éléments de permutation. Chaque élément énuméré dans la permutation doit être du type dupliqué par le type **`record of`**.

La permutation à la place d'un élément isolé signifie que toute série d'éléments est acceptable, à condition qu'elle contienne les mêmes éléments que la liste des valeurs contenues dans la permutation, quoique éventuellement dans un ordre différent. Si la permutation et le symbole d'appariement *AnyElementsOrNone* sont tous deux utilisés à l'intérieur d'une valeur, ils doivent être évalués conjointement.

Le symbole d'appariement *AnyElementsOrNone* utilisé à l'intérieur de la permutation remplace zéro ou un nombre quelconque d'éléments à l'intérieur du segment de la valeur **`record of`** appariée par permutation. Le symbole d'appariement *AnyElementsOrNone* utilisé à l'intérieur d'une permutation doit être évalué en dernier (une fois que tous les autres éléments de la liste de permutation ont été appariés avec un élément déjà contenu dans la liste évaluée).

NOTE 1 – Le symbole d'appariement *AnyElementsOrNone* utilisé à l'intérieur de la permutation a un effet différent que le symbole d'appariement *AnyElementsOrNone* utilisé conjointement avec la permutation, car dans ce dernier cas *AnyElementsOrNone* ne remplace que des éléments consécutifs. Ainsi, {**`permutation`**(1,2,*)} est équivalent à ({*,1,*,2,*},{*,2,*,1,*}), alors que {**`permutation`**(1,2),*} est équivalent à ({1,2},{2,1},*).

NOTE 2 – Quand *AnyElementsOrNone* est utilisé conjointement avec la permutation, un attribut de longueur peut être appliqué à *AnyElementsOrNone* afin de limiter le nombre des éléments appariés par *AnyElementsOrNone* (voir aussi § B.1.4.1). Inversement, aucun attribut de longueur ne doit être ajouté à *AnyElementsOrNone* quand ce symbole d'appariement est utilisé à l'intérieur d'une permutation (mais peut être appliqué en revanche à toute la permutation).

EXEMPLE:

```
type record of integer MySequenceOfType;

template MySequenceOfType MyTemplate1 := { permutation ( 1, 2, 3 ), 5 };
// s'apparie avec l'une quelconque des séquences suivantes de 4 entiers:
// 1,2,3,5; 1,3,2,5; 2,1,3,5; 2,3,1,5; 3,1,2,5; or 3,2,1,5

template MySequenceOfType MyTemplate1 := { permutation ( 1, 2, 3 ), 5 };
 // s'apparie avec l'une quelconque des séquences suivantes de 4 entiers:
// 2,3,1,5; 3,1,2,5; or 3,2,1,5

template MySequenceOfType MyTemplate2 := { permutation ( 1, 2, ? ), 5 };
 // s'apparie avec toute séquence de 4 entiers qui se termine par le chiffre 5
 // et qui contienne les chiffres 1 et 2 au moins une fois en d'autres
 // positions

template MySequenceOfType MyTemplate3 := { permutation ( 1, 2, 3 ), * };
 // s'apparie avec toute séquence d'entiers commençant par 1,2,3; 1,3,2;
 // 2,1,3; 2,3,1; 3,1,2 ou 3,2,1

template MySequenceOfType MyTemplate4 := { *, permutation ( 1, 2, 3 )};
// s'apparie avec toute séquence d'entiers commençant par 1,2,3; 1,3,2;
// 2,1,3; 2,3,1; 3,1,2 ou 3,2,1

template MySequenceOfType MyTemplate5 := { *, permutation ( 1, 2, 3 ),* };
 // s'apparie avec toute séquence d'entiers contenant l'une quelconque des
 // sous-chaînes suivantes à quelque position que ce soit:
 // 1,2,3; 1,3,2; 2,1,3; 2,3,1; 3,1,2 ou 3,2,1

template MySequenceOfType MyTemplate6 := { permutation ( 1, 2, * ), 5 };
 // s'apparie avec toute séquence d'entiers qui se termine par le chiffre 5 et
 // qui contienne les chiffres 1 et 2 au moins une fois en d'autres positions

template MySequenceOfType MyTemplate7 := { permutation ( 1, 2, 3 ), * length (0..5)};
 // (0..5)}; s'apparie avec toute séquence de trois à huit entiers commençant
 // par 1,2,3; 1,3,2; 2,1,3; 2,3,1; 3,1,2 ou 3,2,1

template MySequenceOfType MyTemplate9 := { permutation ( 1, 2, *) length (3..5), 5 };
 // (3..5), 5 }; s'apparie avec toute séquence de quatre à six entiers qui se
 // termine par le chiffre 5 et qui contienne les chiffres 1 et 2 au moins une
 // fois en une autre position
```

B.1.4 Attributs d'appariement de valeurs

B.1.4.0 Généralités

Les attributs suivants peuvent être associés à des mécanismes d'appariement.

B.1.4.1 Restrictions de longueur

L'attribut de restriction de longueur sert à limiter la longueur de valeurs de chaîne et le nombre d'éléments contenus dans une structure matricielle de type **`set of, record of, array.`** Il ne doit être utilisé que comme attribut des mécanismes suivants: Complément, *AnyValue*, *AnyValueOrNone*, *AnyElement* et *AnyElementsOrNone* (mais pas à l'intérieur du mécanisme permutation), permutation, super-ensemble et sous-ensemble. Il peut également être utilisé conjointement avec le mécanisme d'appariement complément et avec l'attribut **`ifpresent`**. La syntaxe de l'attribut **`length`** peut être trouvée dans les § 6.2.3 et 6.3.3.

NOTE – Quand les deux mécanismes d'appariement *complément* et *restriction de longueur* sont utilisés pour un modèle ou un champ de modèle, les restrictions découlant de ces mécanismes doivent s'appliquer indépendamment au modèle ou au champ de modèle.

Les unités de longueur doivent être interprétées conformément au Tableau 4 figurant dans le corps du texte de la présente Recommandation dans le cas des valeurs de chaîne. Pour les types **`set of, record of`** et les matrices, l'unité de longueur est le type dupliqué. Les limites doivent être indiquées par des expressions qui se réduisent à des valeurs spécifiques non négatives, de type **`integer`**. Facultativement, le mot clé **`infinity`** peut être utilisé comme valeur pour la limite supérieure afin d'indiquer qu'il n'y a aucune limite supérieure de longueur.

Les spécifications de longueur pour le modèle ne doivent pas entrer en conflit avec les (éventuelles) restrictions de longueur du type correspondant. Un champ de modèle qui utilise *length* comme attribut d'un symbole s'apparie avec le champ entrant correspondant si et seulement si ce champ entrant s'apparie à la fois avec le symbole et avec son attribut associé. L'attribut de longueur s'apparie si la longueur du champ entrant est supérieure ou égale à la limite inférieure spécifiée et inférieure ou égale à la limite supérieure spécifiée. Dans le cas d'une seule valeur de longueur, l'attribut de longueur ne s'apparie que si la longueur du champ reçu est exactement égale à la valeur spécifiée.

Il est permis d'utiliser une restriction de longueur conjointement avec la valeur spéciale **omit**. Toutefois, dans ce cas, l'attribut de longueur n'a aucun effet (c'est-à-dire qu'il est redondant avec la structure d'omission **omit**). Avec les structures *AnyValueOrNone* et **ifpresent,** il impose une restriction à l'éventuelle valeur entrante.

EXEMPLE:

```
template Mymessage MyTemplate:=
{
  field1 := complement ({4,5},{1,4,8,9}) length (1 .. 6),
  // toute valeur contenant 1, 2, 3, 4, 5 ou 6 éléments est acceptée, à
  // condition qu'elle diffère de {4,5} ou de {1,4,8,9}
  // field2 := "ab*ab" length(13) // la longueur max de la chaîne AnyElementsOrNone
  // est de 9 caractères
  :
}
```

B.1.4.2 L'indicateur "IfPresent"

L'indicateur **ifpresent** signale qu'un appariement peut être effectué si un champ facultatif est présent (c'est-à-dire non omis). Cet attribut peut être utilisé avec tous les mécanismes d'appariement, à condition que le type soit déclaré comme facultatif.

Un champ de modèle qui utilise l'indicateur **ifpresent** s'apparie avec le champ entrant correspondant si et seulement si ce champ entrant s'apparie conformément au mécanisme d'appariement associé, ou si ce champ entrant est absent.

EXEMPLE:

```
template Mymessage:MyTemplate:=
{ :
  field2 := "abcd" ifpresent, // ce champ correspond à "abcd"
  si non omis
  :
  :
}
```

NOTE – *AnyValueOrNone* a exactement la même signification que **? ifpresent**.

B.1.5 Appariement de séquences de caractères

B.1.5.0 Généralités

Des structures alphanumériques peuvent être utilisées dans des modèles afin de définir le format d'une chaîne requise de caractères à recevoir. Les structures alphanumériques peuvent servir à apparier des valeurs de type **charstring** et **universal charstring**. En plus des caractères littéraux, les structures alphanumériques permettent l'utilisation des métacaractères (par exemple, dans une structure alphanumérique, ? et * indiquent respectivement l'appariement avec un caractère quelconque et avec un nombre quelconque de caractères quelconques).

EXEMPLE 1:

```
template charstring MyTemplate:= pattern "ab??xyz*0";
```

Ce modèle s'appariera avec toute chaîne de caractères qui se composerait des caractères "ab", suivis par deux caractères quelconques, suivis par les caractères "xyz", suivis par un nombre quelconque de caractères quelconques (y compris un nombre quelconque de zéros ("0")) avant le caractère "0" de fin de séquence..

S'il est requis d'interpréter littéralement un métacaractère, celui-ci devrait être précédé du métacaractère "\".

EXEMPLE 2:

```
template charstring MyTemplate:= pattern "ab?\?xyz*";
```

Ce modèle s'appariera avec toute chaîne de caractères qui se composerait des caractères 'ab', suivis d'un caractère quelconque, lui-même suivi des caractères '?xyz', suivis d'un nombre quelconque de caractères quelconques.

La liste des métacaractères pour les structures de la notation TTCN-3 est représentée dans le Tableau B.1. Les métacaractères ne doivent pas contenir de blancs (*whitespaces*), sauf s'il s'agit d'une blanc précédé par un caractère d'interligne (*newline*) avant ou à l'intérieur d'une expression d'un ensemble.

Tableau B.1/Z.140 – Liste des métacaractères des structures TTCN-3

Métacaractère	Description
?	S'apparie avec tout caractère (voir Notes 1 et 2)
*	S'apparie avec tout caractère zéro ou plusieurs fois; doit s'apparier avec le nombre de caractères le plus long possible (voir exemple 1 ci-dessus) (voir Notes 1 et 2)
\	Provoque l'interprétation du métacaractère suivant comme un littéral (voir Note 3). Quand il précède un caractère non défini comme ayant la signification d'un métacaractère, le métacaractère '\' et le caractère qui lui est associé s'apparient au caractère qui suit le métacaractère '\' (voir Note 4)
[ ]	S'apparie avec tout caractère contenu dans l'ensemble spécifié. Voir § B.1.5.1 pour plus de détails
-	A la signification d'un métacaractère seulement à l'intérieur d'une paire de crochets ("[" et "]"), sauf les première et dernière positions à l'intérieur des paramètres. Permet de spécifier une étendue de caractères. Voir § B.1.5.1 pour plus de détails
^	A la signification d'un métacaractère en tant que premier caractère qui suit le crochet d'ouverture seulement à l'intérieur d'une paire de crochets ("[" et "]") et provoque l'appariement avec tout caractère complétant l'ensemble des caractères qui suivent ce métacaractère. Voir § B.1.5.1 pour plus de détails
\q{ groupe, plan, rangée, cellule }	S'apparie avec le caractère universel spécifié par le quadruplet
{référence}	Insère la chaîne définie par l'utilisateur qui est indiquée et l'interprète comme une expression régulière. Voir § B.1.5.2 pour plus de détails
\N{référence}	S'apparie avec tout caractère du jeu de caractères, ce jeu étant défini par la définition référencée; voir § B.1.5.4 pour plus de détails
\d	S'apparie avec tout chiffre numérique (équivalent à [0-9])
\w	S'apparie avec tout caractère alphanumérique (équivalent à [0-9a-zA-Z])
\t	S'apparie avec le caractère de commande HT (9) du jeu C0 (voir ISO/CEI 6429 [11])
\n	S'apparie avec l'un quelconque des caractères de commande du jeu C0 suivants: LF(10), VT(11), FF(12), CR(13) (voir ISO/CEI 6429 [11]) (conjointement appelés caractères d'interligne)
\r	S'apparie avec le caractère de commande CR du jeu C0 (voir ISO/CEI 6429 [11])
\s	S'apparie avec l'un quelconque des caractères de commande du jeu C0 suivants: HT(9), LF(10), VT(11), FF(12), CR(13), SP(32) (voir ISO/CEI 6429 [11], Rec. UIT-T T.50 [9]) (conjointement appelés caractères blancs)
\b	S'apparie avec une limite de mot (tout caractère graphique, sauf SP ou DEL, est précédé ou suivi par l'un quelconque des caractères blancs ou d'interligne)
\"	S'apparie avec le caractère de guillemet droit
" "	S'apparie avec le caractère de guillemet droit
|	Utilisé afin d'indiquer un choix entre deux expressions
( )	Utilisé afin de grouper une expression

Tableau B.1/Z.140 – Liste des métacaractères des structures TTCN-3

Métacaractère	Description
`#(n, m)`	S'apparie avec l'expression précédente au moins n fois mais au plus m fois (suffixe). Voir § B.1.5.3 pour plus de détails
`#n`	S'apparie avec l'expression précédente exactement n fois (n correspondant à un seul chiffre) (suffixe); identique à #(n)
`+`	S'apparie avec l'expression précédente une ou plusieurs fois (suffixe); identique à #(1,)
NOTE 1 – Les métacaractères ? et * peuvent s'apparier avec l'un quelconque des caractères du jeu de caractères du type racine du modèle ou du champ de modèle dans lequel ils sont utilisés (c'est-à-dire sans tenir compte des contraintes de type appliquées). Toutefois, on ne doit pas oublier que les opérations de réception obligent à vérifier le type du message avant de tenter de l'apparier. Par conséquent, les valeurs reçues qui ne sont pas conformes à la spécification du sous-type du modèle ou du champ de modèle ne sont jamais fournies pour appariement.	
NOTE 2 – Dans d'autres langages/notations, les métacaractères ? et * ont une signification différente. Toutefois, dans la notation TTCN, ces caractères sont communément utilisés pour l'appariement tel que spécifié dans le présent tableau.	
NOTE 3 – Par conséquent, le caractère de barre oblique inverse peut être apparié par une paire de caractères de barre oblique inverse sans espace entre eux (\\), par exemple, la structure '\\d' s'appariera avec la chaîne '\d'; les crochets d'ouverture et de fermeture peuvent être appariés par '\[' et '\]' respectivement, etc.	
NOTE 4 – Cette utilisation du métacaractère '\' est déconseillée car d'autres métacaractères pourront être définis ultérieurement.	

B.1.5.1 Expression d'un ensemble

Une liste de caractères englobés dans une paire de crochet '[' et ']' s'apparie avec tout caractère isolé de cette liste. L'expression d'un ensemble est délimitée par les symboles '[' ']'. En plus des caractères littéraux, il est possible de spécifier des séries de caractères au moyen du caractère tiret '-' comme séparateur. La série comprend le caractère immédiatement avant le séparateur, le caractère immédiatement après celui-ci et tous les caractères avec un code de caractère entre les codes des premier et dernier caractères de la série. Un caractère tiret '-' à l'intérieur de la liste mais non précédé ni suivi d'un caractère perd sa signification particulière.

L'expression d'un ensemble peut également être rendue négative par insertion du caractère accent circonflexe '^' comme premier caractère après le crochet d'ouverture. Cette expression négative a priorité sur les séries de caractères. Par conséquent, un tiret '-' immédiatement après un accent circonflexe '^' de négation doit être traité comme un caractère littéral.

Une liste vide et une liste vide rendue négative ne sont pas autorisées. Par conséquent, un crochet de fermeture ']' immédiatement après un crochet d'ouverture '[' ou un accent circonflexe après le crochet d'ouverture '[' et immédiatement suivi par un crochet de fermeture ']' doit être traité comme des caractères littéraux.

Tous les métacaractères, à l'exception de ceux énumérés ci-dessous, perdent leur signification particulière à l'intérieur de la liste:

- ']' pas en première position et pas immédiatement après un accent circonflexe '^' en première position;
- '-' pas en première ni en dernière position dans la liste;
- '^'en première position dans la liste sauf quand il est immédiatement suivi par un crochet de fermeture;
- '\', \d, \t, \w, \r, \n, \s et \b;
- \q{groupe,plan,rangée,cellule};
- **\N**{référence}.

NOTE 1 – Les listes imbriquées ne sont pas autorisées (par exemple, dans la structure '[ab[r-z]]', le deuxième crochet d'ouverture '[' indique un caractère littéral '[', le premier crochet de fermeture ']' ferme la liste et le deuxième crochet de fermeture ']' provoque une erreur en tant que crochet d'ouverture étranger dans la structure).

NOTE 2 – Pour inclure un caractère littéral accent circonflexe "^", l'insérer n'importe où sauf en première position, ou le faire précéder d'une barre oblique inverse. Pour inclure un caractère littéral tiret "-", l'insérer en première ou dernière position dans la liste, ou le faire précéder d'une barre oblique inverse. Pour inclure un caractère littéral crochet de fermeture "]", l'insérer en première position ou le faire précéder d'une barre oblique inverse. Si le premier caractère de la liste est l'accent circonflexe "^", alors les caractères "-" et "]" s'apparient également l'un avec l'autre lorsqu'ils suivent immédiatement ledit accent circonflexe.

EXEMPLE:

```
template charstring RegExp1:= pattern '[a-z]';  // s'appariera avec tout caractère de a à z

template charstring RegExp2:= pattern '[^a-z]'; // s'appariera avec tout caractère sauf de a à z

template charstring RegExp3:= pattern '[AC-E][0-9][0-9][0-9]YKE';
```

```
// RegExp3 s'appariera avec une chaîne qui commence par la lettre A ou une
// lettre comprise entre C et E (mais pas p.ex. le B), puis a trois chiffres
// et les lettres YKE
```

B.1.5.2 Expression de référence

En plus des valeurs de chaîne directes, il est également possible d'utiliser, dans la structure, des références à des modèles, constantes, variables ou paramètres de module existants. La référence est englobée dans les caractères "{" "}" et la référence soit se réduire à un des types de chaîne de caractères. Le contenu des modèles, constantes ou variables référencés doit être traité comme une expression régulière. Chaque expression ne doit être déréférencée qu'une seule fois.

EXEMPLE:

```
const charstring MyString:= "ab?";

template charstring MyTemplate:= pattern '{MyString}';
```

Ce modèle s'appariera avec toute chaîne de caractères qui se compose des caractères 'ab', suivis d'un caractère quelconque. En effet, toute chaîne de caractères suivant le mot clé **`pattern`**, soit explicitement ou par référence, sera interprété suivant les règles définies dans le présent paragraphe.

```
template universal charstring MyTemplate1:= pattern '{MyString}de\q{1, 1, 13, 7}';
```

Ce modèle s'appariera avec toute chaîne de caractères qui se compose des caractères 'ab', suivis d'un caractère quelconque, suivi par les caractères 'de', suivis par le caractère contenu dans l'ISO/CEI 10646 avec groupe=1, plan=1, rangée=13 et cellule=7.

Si une expression de référence fait référence à un modèle, une constante ou une variable qui contient une ou plusieurs expressions de référence, alors les références contenues dans le modèle, la constante ou la variante référencés doivent être déréférencées récursivement avant que leur contenu soit inséré dans la structure faisant référence.

EXEMPLE:

```
const charstring MyConst2 := pattern "ab";
template charstring RegExp1 := pattern "{MyConst2}";
   // s'apparie avec la chaîne "ab"
template charstring RegExp2 := pattern "{RegExp1}{RegExp1}";
   // s'apparie avec la chaîne "abab"
template charstring RegExp3 := pattern "c{RegExp2}d";
   // s'apparie avec la chaîne "cababd"

template charstring RegExp4 := pattern "{Reg";
template charstring RegExp5 := pattern "Exp1}";
template charstring RegExp6 := pattern "{RegExp4}{RegExp5}";
  // ne s'apparie qu'avec la chaîne {RegExp1} c'est-à-dire, ne doit pas
  // être traité comme une expression de référence faisant
  // référence au modèle RegExp1
```

B.1.5.3 Appariement n fois d'une expression

Afin de spécifier que l'expression précédente devrait être appariée un certain nombre de fois, une des syntaxes suivantes doit être utilisée: '#(n, m)', '#(n,)', '#(, m)', '#(n)', '#n' ou '+'. La forme '#(n, m)' spécifie que l'expression précédente doit impérativement être appariée au moins n fois mais pas plus de m fois. Le métacaractère suffixe '#(, m)' spécifie que l'expression précédente doit impérativement être appariée au moins n fois, alors que '#(, m)' indique que l'expression précédente ne doit pas être appariée plus de m fois. Les métacaractères (suffixes) '#(n)' et '#n' spécifient que l'expression précédente doit impérativement être appariée exactement n fois (ils sont équivalents à '#(n,n)'). Dans la forme '#n', n doit être constitué d'un seul chiffre. Le métacaractère suffixe '+' indique que l'expression précédente doit impérativement être appariée au moins une fois (il est équivalent à '#(1,)').

EXEMPLE:

```
template charstring RegExp4:= pattern '[a-z]#(9, 11)'; // correspond à au moins 9 mais pas plus
                                                       // que 11 caractères de a à z
template charstring RegExp5a:= pattern '[a-z]#(9)';    // correspond à exactement
                                                       // 9 caractères de a à z
template charstring RegExp5b:= pattern '[a-z]#9';      // correspond à caractères de a à z
  template charstring RegExp6:= pattern '[a-z]#(9, )'; // correspond à au moins
                                                       // 9 caractères de a à z
template charstring RegExp7:= pattern '[a-z]#(, 11)';  // correspond à pas plus
                                                       // que 11 caractères de a à z
template charstring RegExp8:= pattern '[a-z]+';        // correspond à au moins
                                                       // 1 caractère de a à z,
```

B.1.5.4 Appariement d'un jeu de caractères référencé

Une notation de la forme "**\N{**reference**}**", où "reference" indique un modèle, une constante, une variable ou un paramètre de module d'une longueur d'un caractère, s'apparie avec le caractère contenu dans la valeur ou le modèle référencé.

La référence à un modèle, une constante, une variable ou un paramètre de module d'une longueur autre que d'un caractère doit provoquer une erreur.

Une notation de la forme "**\N{**typereference**}**", où "typereference" est une référence à un type **charstring** ou **universal charstring**, s'apparie avec tout caractère du jeu de caractères indiqué par le type référencé.

NOTE 1 – Les cas dans lesquels le jeu de caractères référencé n'est pas un sous-ensemble vrai des valeurs autorisées par la définition de type du modèle ou du champ de modèle pour lequel la séquence de caractères est utilisée ne doivent pas être considérés comme une erreur (mais par exemple, l'appariement ne peut jamais se produire si les deux jeux ne se recoupent pas.

NOTE 2 – \N{**charstring**} est équivalent à ? quand ce dernier est appliqué à un modèle ou un champ de modèle de type **charstring** et \N{**universal charstring**} est équivalent à ? quand ce dernier est appliqué à un modèle ou un champ de modèle de type **universal charstring** (mais provoque une erreur s'il est appliqué à un modèle ou un champ de modèle de type **charstring**).

EXEMPLE:

```
type charstring MyCharRange ('a'..'z');
type charstring MyCharList ('a', 'z');
const MyCharRange myCharR := 'r';

template charstring myTempPatt1 := pattern '\N { myCharR }';
// myTempPatt1 ne doit s'apparier qu'avec la chaîne 'r'

template charstring myTempPatt2 := pattern '\N { MyCharRange }';
// myTempPatt2 doit s'apparier avec toute chaîne contenant
// un seul caractère de a à z

template MyCharRange myTempPatt3 := pattern '\N { MyCharList }';
// myTempPatt3 ne doit s'apparier qu'avec les chaînes 'a' et 'r'

template MyCharList myTempPatt4 := pattern '\N { MyCharRange }';
// myTempPatt4 ne doit s'apparier qu'avec les chaînes 'a' et 'r'
```

B.1.5.5 Règles de compatibilité de type pour les structures

Pour les structures référencées (voir § B.1.5.2) et les jeux de caractères référencés (voir § B.1.5.4), des règles de compatibilité de type spécifiques s'appliquent: un type, un modèle, une constante, une variable ou un paramètre de module référencé du type **charstring** peut toujours être utilisé dans la spécification de la structure d'un modèle ou d'un champ de modèle de type **universal charstring**; un type, un modèle ou une valeur référencé du type **universal charstring** peut être utilisé dans la spécification de la structure d'un modèle ou d'un champ de modèle de type **charstring** si tous les caractères utilisés dans le modèle ou la valeur référencé et le jeu de caractères autorisés par le type référencé ont leurs caractères correspondants dans le type **charstring** (voir la définition des caractères correspondants dans le § 6.7.1).

Annexe C

Fonctions de notation TTCN-3 prédéfinies

La présente annexe définit les fonctions TTCN-3 prédéfinies.

C.0 Procédures générales de traitement d'exception

Les situations d'erreur (par exemple, paramètre d'entrée en dehors de l'étendue autorisée, paramètre d'entrée d'un type erroné, valeur d'entrée contenant un caractère inapproprié, etc.), pour lesquelles aucune règle explicite de traitement d'exception n'est définie dans les paragraphes pertinents de la présente annexe doivent provoquer une erreur au moment de la compilation ou de l'exécution en notation TTCN-3. La possibilité de différencier les situations d'erreur qui provoquent une erreur au moment de la compilation et celles qui provoquent une erreur au moment de l'exécution est offerte en option en fonction de l'utilitaire implémenté.

C.1 Conversion d'entier en caractère

```
int2char(integer value) return charstring
```

Cette fonction convertit une valeur de type **`integer`** dans l'étendue de 0 à 127 (codage sur 8 éléments binaires) en valeur de type **`charstring`** d'une longueur d'un seul caractère. La valeur d'entier décrit le codage sur 8 éléments binaires du caractère.

C.2 Conversion de caractère en entier

```
char2int(charstring value) return integer
```

Cette fonction convertit une valeur de type **`charstring`** d'une longueur d'un seul caractère en valeur d'entier dans l'étendue de 0 à 127. La valeur d'entier décrit le codage sur 8 éléments binaires du caractère.

C.3 Conversion d'entier en caractère universel

```
int2unichar(integer value) return universal charstring
```

Cette fonction convertit une valeur de type **`integer`** dans l'étendue de 0 à 2 147 483 647 (codage sur 32 éléments binaires) en valeur de type **`universal charstring`** d'une longueur d'un seul caractère. La valeur d'entier décrit le codage sur 32 éléments binaires du caractère.

C.4 Conversion de caractère universel en entier

```
unichar2int(universal charstring value) return integer
```

Cette fonction convertit une valeur de type **`universal charstring`** d'une longueur d'un seul caractère en valeur d'entier dans l'étendue de 0 à 2 147 483 647. La valeur d'entier décrit le codage sur 32 éléments binaires du caractère.

C.5 Conversion de chaîne binaire en entier

```
bit2int(bitstring value) return integer
```

Cette fonction convertit une valeur isolée de type **`bitstring`** en valeur isolée de type **`integer`**.

Pour les besoins de cette conversion, un type **`bitstring`** doit être interprété comme une valeur positive en base 2 de type **`integer`**. Le bit de droite est de plus faible poids, le bit de gauche est de plus fort poids. Les bits 0 et 1 représentent respectivement les valeurs décimales 0 et 1.

C.6 Conversion de chaîne hexadécimale en entier

```
hex2int(hexstring value) return integer
```

Cette fonction convertit une valeur isolée de type **`hexstring`** en valeur isolée de type **`integer`**.

Pour les besoins de cette conversion, un type **hexstring** doit être interprété comme une valeur positive en base 16 de type **integer**. Le chiffre hexadécimal de droite est de plus faible poids, le chiffre hexadécimal de gauche est de plus fort poids. Les chiffres hexadécimaux 0 à F représentent respectivement les valeurs décimales 0 à 15.

C.7 Conversion de chaîne d'octets en entier

```
oct2int(octetstring value) return integer
```

Cette fonction convertit une valeur isolée de type **octetstring** en valeur isolée de type **integer**.

Pour les besoins de cette conversion, un type **octetstring** doit être interprété comme une valeur positive en base 16 de type **integer**. Le chiffre hexadécimal de droite est de plus faible poids, le chiffre hexadécimal de gauche est de plus fort poids. Le nombre de chiffres hexadécimaux fourni doit être un multiple de 2 étant donné qu'un seul octet est composé de deux chiffres hexadécimaux. Les chiffres hexadécimaux 0 à F représentent respectivement les valeurs décimales 0 à 15.

C.8 Conversion de chaîne de caractères en entier

```
str2int(charstring value) return integer
```

Cette fonction convertit une valeur de type **charstring** représentant une valeur de type **integer** en valeur équivalente de type **integer**.

EXEMPLE:

```
str2int("66")     // retournera la valeur de type integer 66

str2int("-66")    // retournera la valeur de type integer -66

str2int("abc")    // produira une erreur de compilation ou de test
                  // élémentaire

str2int("0")      // retournera la valeur de type integer 0
```

C.9 Conversion d'entier en chaîne binaire

```
int2bit(in integer value, in integer length) return bitstring
```

Cette fonction convertit une valeur isolée de type **integer** en valeur isolée de type **bitstring**. La chaîne résultante a une longueur en bits égale au paramètre `length`.

Pour les besoins de cette conversion, un type **bitstring** doit être interprété comme une valeur positive en base 2 de type **integer**. Le bit de droite est de plus faible poids, le bit de gauche est de plus fort poids. Les bits 0 et 1 représentent respectivement les valeurs décimales 0 et 1 . Si la conversion produit une valeur contenant moins de bits que spécifié dans le paramètre `length`, alors le type **bitstring** doit être justifié à gauche avec des zéros.

C.10 Conversion d'entier en chaîne hexadécimale

```
int2hex(in integer value, in integer length) return hexstring
```

Cette fonction convertit une valeur isolée de type **integer** en valeur isolée de type **hexstring**. La chaîne résultante a une longueur égale, en chiffres hexadécimaux, au paramètre `length`.

Pour les besoins de cette conversion, un type **hexstring** doit être interprété comme une valeur positive en base 16 de type **integer**. Le chiffre hexadécimal de droite est de plus faible poids, le chiffre hexadécimal de gauche est de plus fort poids. Les chiffres hexadécimaux 0 à F représentent respectivement les valeurs décimales 0 à 15. Si la conversion produit une valeur avec un moins grand nombre de chiffres hexadécimaux que spécifié dans le paramètre `length`, alors le type **hexstring** doit être justifié à gauche avec des zéros.

C.11 Conversion d'entier en chaîne d'octets

```
int2oct(in integer value, in integer length) return octetstring
```

Cette fonction convertit une valeur isolée de type **integer** en valeur isolée de type **octetstring** . La chaîne résultante a une longueur en octets égale au paramètre `length`.

Pour les besoins de cette conversion, un type **octetstring** doit être interprété comme une valeur positive en base 16 de type **integer**. Le chiffre hexadécimal de droite est de plus faible poids, le chiffre hexadécimal de gauche est de plus fort poids. Le nombre de chiffres hexadécimaux fourni doit être un multiple de 2 étant donné qu'un seul octet est composé de deux chiffres hexadécimaux. Les chiffres hexadécimaux 0 à F représentent respectivement les valeurs décimales 0 à 15. Si la conversion produit une valeur avec un moins grand nombre de chiffres hexadécimaux que spécifié dans le paramètre `length`, alors le type **hexstring** doit être justifié à gauche avec des zéros.

C.12 Conversion d'entier en chaîne de caractères

```
int2str(integer value) return charstring
```

Cette fonction convertit la valeur d'entier en son équivalent de type "string" (la base de la chaîne de retour est toujours décimale).

EXEMPLE:

```
int2str(66)    // retournera la valeur charstring "66"

int2str(-66)   // retournera la valeur charstring "-66"

int2str(0)     // retournera la valeur charstring "0"
```

C.13 Longueur de type chaîne

```
lengthof(any_string_type value) return integer
```

Cette fonction retourne la longueur d'une valeur qui est de type **bitstring**, **hexstring**, **octetstring**, ou toute chaîne de caractères. Les unités de longueur pour chaque type chaîne sont définies dans le Tableau 4.

La longueur d'un type **universal charstring** doit être calculée par comptage chaque caractère de combinaison et chaque caractère syllabique du système Han-geul (y compris les éléments de remplissage) tour à tour (voir l'ISO/CEI 10646 [10], paragraphes 23 et 24).

EXEMPLE:

```
lengthof('010'B) // retourne 3

lengthof('F3'H) // retourne 2

lengthof('F2'O) // retourne 1

lengthof (universal charstring : "Length_of_Example") // retourne 17
```

C.14 Nombre d'éléments contenus dans une valeur structurée

```
sizeof(any_type value) return integer
```

Cette fonction retourne le nombre effectif d'éléments d'un paramètre, d'une constante, d'une variable, d'un **template** d'un module, de type **record**, **record of**, **set**, **set of** ou de type matriciel (voir Note). Dans le cas de valeurs de type **record of** et **set of,** de modèles ou de matrices, la valeur effective à retourner est le numéro séquentiel du dernier élément défini (c'est-à-dire l'indice de cet élément plus 1).

NOTE – Seuls sont calculés les éléments de l'objet TTCN-3 qui est le paramètre de la fonction; c'est-à-dire qu'aucun élément des types/valeurs imbriqués n'est pris en considération lors de la détermination de la valeur de retour.

EXEMPLE:

```
// si l'on a
type record MyPDU
  {  boolean field1  optional,
     integer field2
  };

template MyPDU  MyTemplate
  { field1  omit,
    field2 5
  };

var integer numElements;

// alors

numElements := sizeof(MyTemplate);  // retourne 1
```

```
// si l'on a
      type record length(0..10) of integer MyList;
      var MyList MyRecordVar;
      MyRecordVar := { 0, 1, omit, 2, omit };


      // alors
      numElements := sizeof(MyRecordVar);
      // retourne 4 sans tenir compte du fait que l'élément MyRecordVar[2] est
      // indéfini
```

C.15 La fonction "IsPresent"

```
ispresent(any_type value) return boolean
```

Cette fonction n'est autorisée que pour les types **record** et **set** et retourne la valeur **`true`** si et seulement si la valeur du champ référencé est présente dans l'instance effective de l'objet de données référencé. L'argument de la fonction **`ispresent`** doit être une référence à un champ de type **record** ou **set**.

```
// si l'on a
type record MyRecord
  {  boolean  field1 optional,
     integer field2
  }
// et compte tenu du fait que MyPDU est un modèle de type MyRecord
// et du fait que l'élément received_PDU est également de type MyRecord,
// alors
MyPort.receive(MyPDU) -> value received_PDU
ispresent(received_PDU.field1)
// retourne la valeur "true" si le champ field1 est présent dans l'instance
// effective de MyPDU
```

C.16 La fonction "IsChosen"

```
ischosen(any_type value) return boolean
```

Cette fonction retourne la valeur **`true`** si et seulement si la référence de l'objet de données spécifie la variante du type **`union`** qui est effectivement choisie pour un certain objet de données.

EXEMPLE:

```
// si l'on a
type union MyUnion
  {  PDU_type1   p1,
     PDU_type2   p2,
     PDU_type p3
  }

// et étant donné que l'unité MyPDU est un modèle du type MyUnion
// et que l'unité received_PDU est également du type MyUnion
// alors
MyPort.receive(MyPDU) -> value received_PDU
ischosen(received_PDU.p2)
// retourne la valeur "true" si l'instance effective de MyPDU achemine une
// unité PDU du type PDU_type2
```

C.17 La fonction "Regexp"

```
regexp (any_character_string_type instr, charstring expression, integer groupno) return
character_string_type
```

Cette fonction retourne la sous-chaîne de la chaîne d'entrée de caractères `instr`, qui est le contenu du nième groupe correspondant à l'`expression`. La chaîne d'entrée `instr` peut contenir tout type de chaîne de caractères. Le type de la chaîne de caractères retournée est le type radical de la chaîne `instr`. L'expression est une structure de caractères comme décrit dans le § B.1.5. Le numéro du groupe à retourner est spécifié par `groupno`, qui doit être un entier positif. Les numéros de groupe sont affectés par l'ordre des occurrences du crochet d'ouverture d'un groupe et comptés de 0 par échelons unitaires. Si aucune sous-chaîne remplissant toutes les conditions (c'est-à-dire structure et numéro de groupe) n'est trouvée dans la chaîne d'entrée, une chaîne vide est retournée.

EXEMPLE:

```
// si l'on a
var charstring mypattern2 := "
var charstring  myinput := '      date: 2001-10-20 ;  msgno: 17; exp  '
var charstring mypattern := '[ /t]#(,)date:[ \d\-]#(,);[ /t]#(,)msgno: (\d#(1,3)); [exp]#(0,1)'

// alors l'expression
var charstring mystring := regexp(myinput, mypattern,1)
// retournera la valeur "17"
```

C.18 Conversion de chaîne binaire en chaîne de caractères

bit2str (bitstring value**) return charstring**

Cette fonction convertit une valeur isolée de type **bitstring** en valeur isolée de type **charstrin**g. Le type résultant **charstring** a la même longueur que le type **bitstring** et contient seulement les caractères '0' et '1'.

Aux fins de cette conversion, un type **bitstring** devrait être converti en un type **charstrin**g. Chaque bit du type **bitstring** est converti en un caractère '0' ou '1' selon la valeur 0 ou 1 du bit. L'ordre séquentiel des caractères dans le type résultant **charstring** est le même que l'ordre des bits dans le type **bitstring**.

EXEMPLE:

bit2str ('1110101'B) will return "1110101"

C.19 Conversion de chaîne hexadécimale en chaîne de caractères

hex2str (hexstring value**) return charstring**

Cette fonction convertit une valeur isolée de type chaîne hexadécimale en valeur isolée de type chaîne de caractères. Le type résultant charstring a la même longueur que le type hextring et contient seulement les caractères '0' à '9' et 'A' à 'F'.

Aux fins de cette conversion, un type **hexstring** devrait être converti en type **charstring**. Chaque chiffre hexadécimal du type **hexstring** est converti en un caractère '0' à '9' et 'A' à 'F' selon la valeur 0 à 9 ou A à F du chiffre hexadécimal. L'ordre séquentiel des caractères dans le type résultant **charstring** est le même que l'ordre des chiffres dans le type **hexstring**.

EXEMPLE:

hex2str ('AB801'H) will return "AB801"

C.20 Conversion de chaîne d'octets en chaîne de caractères

oct2str (octetstring invalue**) return charstring**

Cette fonction convertit une valeur "in" de type **octetstring** en valeur de type **charstring** représentant la chaîne équivalente de la valeur d'entrée. Le type résultant **charstring** doit avoir la même longueur que le type entrant **octetstring**.

Aux fins de cette conversion, chaque chiffre hexadécimal de valeur "in" `invalue` est converti en un caractère '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E' ou 'F' retournant la valeur du chiffre hexadécimal. L'ordre séquentiel des caractères dans le type résultant **charstring** est le même que l'ordre des chiffres hexadécimaux dans le type **octetstring**.

EXEMPLE:

oct2str ('4469707379'O) = "4469707379"

C.21 Conversion de chaîne de caractères en chaîne d'octets

str2oct (charstring invalue**) return octetstring**

Cette fonction convertit une chaîne de type **charstring** en chaîne de type **octetstring**. La chaîne `invalue` doit contenir un nombre pair de caractères, chacun de ceux-ci devant correspondre à un seul des caractères graphiques '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e' 'f', 'A', 'B', 'C', 'D', 'E' ou 'F'. Le type résultant **octetstring** aura la même longueur que le type entrant **charstring**.

EXEMPLE:

```
str2oct ("54696E6B792D57696E6B79") = '54696E6B792D57696E6B79'O
```

C.22 Conversion de chaîne binaire en chaîne hexadécimale

```
bit2hex (bitstring value) return hexstring
```

Cette fonction convertit une valeur isolée de type **bitstring** en valeur isolée de type **hexstring**. Le type résultant **hexstring** représente la même valeur que le type **bitstring**.

Aux fins de cette conversion, une chaîne binaire doit être convertie en chaîne hexadécimale, où la chaîne binaire est subdivisée en groupes de quatre bits à partir du bit de droite. Chaque groupe de quatre bits est converti en chiffre hexadécimal comme suit:

'0000'B → '0'H, '0001'B → '1'H,'0010'B → '2'H, '0011'B → '3'H, '0100'B → '4'H, '0101'B → '5'H,
'0110'B → '6'H, '0111'B → '7'H,'1000'B → '8'H, '1001'B → '9'H, '1010'B → 'A'H, '1011'B → 'B'H,
'1100'B → 'C'H, '1101'B → 'D'H,'1110'B → 'E'H, and '1111'B → 'F'H.

Quand le groupe de bits de gauche contient effectivement moins de 4 bits, ce groupe est rempli avec '0'B à partir de la gauche jusqu'à ce qu'il contienne exactement 4 bits; puis il est converti. L'ordre séquentiel des chiffres hexadécimaux dans la chaîne hexadécimale résultante est le même que l'ordre des groupes de 4 bits dans la chaîne binaire.

EXEMPLE:

```
bit2hex ('111010111'B)= '1D7'H
```

C.23 Conversion de chaîne hexadécimale en chaîne d'octets

```
hex2oct (hexstring value) return octetstring
```

Cette fonction convertit une valeur isolée de type **hexstring** en valeur isolée de type **octetstring**. Le type résultant **octetstring** représente la même valeur que le type **hexstring**.

Aux fins de cette conversion, un type **hexstring** doit être converti en type **octetstring**, où le type **octetstring** contient la même séquence de chiffres hexadécimaux que le type **hexstring** quand la longueur du type **hexstring** modulo 2 est 0. Sinon, le type résultant **octetstring** contient 0 comme chiffre hexadécimal de gauche, suivi par la même séquence de chiffres hexadécimaux que dans le type **hexstring**.

EXEMPLE:

```
hex2oct ('1D7'H)= '01D7'O
```

C.24 Conversion de chaîne binaire en chaîne d'octets

```
bit2oct (bitstring value) return octetstring
```

Cette fonction convertit une valeur isolée de type **bitstring** en valeur isolée de type **octetstring**. Le type résultant **octetstring** représente la même valeur que le type **bitstring**.

Pour la conversion, ce qui suit est vrai: bit2oct(value)=hex2oct(bit2hex(value)).

EXEMPLE:

```
bit2oct ('111010111'B)= '01D7'O
```

C.25 Conversion de chaîne hexadécimale en chaîne binaire

```
hex2bit (hexstring value) return bitstring
```

Cette fonction convertit une valeur isolée de type **hexstring** en valeur isolée de type **bitstring**. Le type résultant **bitstring** représente la même valeur que le type **hexstring**.

Aux fins de cette conversion, un type **hexstring** doit être converti en type **bitstring**, où les chiffres hexadécimaux du type **hexstring** sont convertis en groupes de bits comme suit:

'0'H → '0000'B, '1'H → '0001'B,'2'H → '0010'B, '3'H → '0011'B, '4'H → '0100'B, '5'H → '0101'B,
'6'H → '0110'B, '7'H → '0111'B,'8'H → '1000'B, '9'H → '1001'B, 'A'H → '1010'B, 'B'H → '1011'B,
'C'H → '1100'B, 'D'H → '1101'B,'E'H → '1110'B, and 'F'H → '1111'B.

L'ordre séquentiel des groupes de 4 bits dans le type résultant **bitstring** est le même que l'ordre des chiffres hexadécimaux dans le type **hexstring**.

EXEMPLE:

```
hex2bit ('1D7'H)= '000111010111'B
```

C.26 Conversion de chaîne d'octets en chaîne hexadécimale

```
oct2hex (octetstring value) return hexstring
```

Cette fonction convertit une valeur isolée de type **octetstring** en valeur isolée de type **hexstring**. Le type résultant **hexstring** représente la même valeur que le type **octetstring**.

Aux fins de cette conversion, un type **octetstring** doit être converti en type **hexstring** contenant la même séquence de chiffres hexadécimaux que le type **octetstring**.

EXEMPLE:

```
oct2hex ('1D74'O)= '1D74'H
```

C.27 Conversion de chaîne d'octets en chaîne binaire

```
oct2bit (octetstring value) return bitstring
```

Cette fonction convertit une valeur isolée de type **octetstring** en valeur isolée de type **bitstring**. Le type résultant **bitstring** représente la même valeur que le type **octetstring**.

Pour la conversion, ce qui suit est vrai: oct2bit(value)=hex2bit(oct2hex(value)).

EXEMPLE:

```
oct2bit ('01D7'O)='0000000111010111'B
```

C.28 Conversion d'entier en nombre à virgule flottante

```
int2float (integer value) return float
```

Cette fonction convertit une valeur de type **integer** en valeur de type **float**.

EXEMPLE:

```
int2float(4) = 4.0
```

C.29 Conversion de nombre à virgule flottante en entier

```
float2int (float value) return integer
```

Cette fonction convertit une valeur de type **float** en valeur de type **integer** par suppression de la partie fractionnaire de l'argument et retour du type résultant **integer**.

EXEMPLE:

```
float2int(3.12345E2) = float2int(312.345) = 312
```

C.30 La fonction de générateur de nombre aléatoire

```
rnd ([float seed]) return float
```

La fonction **rnd** retourne un (pseudo-) nombre aléatoire inférieur à 1 mais supérieur ou égal à 0. Le générateur de nombre aléatoire est initialisé au moyen d'une valeur d'initialisation facultative. Ensuite, si aucune nouvelle valeur d'initialisation n'est fournie, le dernier nombre produit sera utilisé comme valeur d'initialisation pour le prochain nombre aléatoire. Sans initialisation préalable, une valeur calculée à partir de l'horloge du système sera utilisée comme valeur d'initialisation quand la fonction **rnd** est utilisée pour la première fois.

NOTE – Chaque fois que la fonction **rnd** est initialisée avec la même valeur d'initialisation, cette fonction doit répéter la même séquence de nombres aléatoires.

Afin de produire un entier aléatoire dans une certaine étendue, la formule suivante peut être utilisée:

```
float2int(int2float(upperbound - lowerbound +1)*rnd()) + lowerbound
// Ici, limitesupérieure et limiteinférieure indiquent le nombre le plus élevé et
// le nombre le moins élevé de l'étendue.
```

C.31 La fonction de sous-chaîne

```
substr (any_string_type value, in integer index, in integer returncount)
return    input_string_type
```

Cette fonction retourne une sous-chaîne à partir d'une valeur qui est de type **`bitstring`**, **`hexstring`**, **`octetstring`**, ou qui est une chaîne de caractères quelconque. Le type de la sous-chaîne est le type radical de la valeur d'entrée. Le point de départ de la sous-chaîne à retourner est défini par le second paramètre "in" (indice). L'indexation commence à partir de zéro. Le troisième paramètre d'entrée définit la longueur de la sous-chaîne à retourner. Les unités de longueur sont conformes à la définition du Tableau 4.

EXEMPLE:

```
substr ('00100110'B, 3, 4)      // retourne '0011'B

substr ('ABCDEF'H, 2, 3)        // retourne 'CDE'H

substr ('01AB23CD'O, 1, 2)      // retourne 'AB23'O

substr ("My name is JJ", 11, 2) // retourne "JJ"
```

C.32 Nombre d'éléments dans un type structuré

```
sizeoftype(any_type value) return integer
```

Cette fonction retourne le nombre déclaré d'éléments d'un paramètre de module, d'une constante, d'une variable ou d'un modèle (**`template`**) d'un type **`record of`** ou **`set of`** ou d'une matrice (voir Note). Cette fonction doit être appliquée aux valeurs des types avec restriction de longueur. Le nombre effectif à retourner est le numéro séquentiel du dernier élément, indépendamment du fait que sa valeur soit ou non définie (c'est-à-dire l'indice supérieur de longueur de la définition de type sur laquelle le paramètre de la fonction est fondé, plus 1).

NOTE – Seuls sont calculés les éléments de l'objet TTCN-3 qui est le paramètre de la fonction; c'est-à-dire qu'aucun élément des types/valeurs imbriqués n'est pris en considération lors de la détermination de la valeur de retour.

EXEMPLE:

```
// si l'on a
type record of integer MyPDU1;
type set length(1..8) of integer MyPDU2;
type record length(10) of integer MyPDU3;

var MyPDU1 MyRecordOfVar1;
var MyPDU2 MyRecordOfVar2;
var MyPDU3 MyRecordOfVar3;

var integer numElements;

// alors
numElements := sizeoftype(MyRecordOfVar1);  // retourne une erreur car MyPDU1
                                            // n'est pas contraint
numElements := sizeoftype(MyRecordOfVar2);  // retourne 8
numElements := sizeoftype(MyRecordOfVar3);  // retourne 10
```

C.33 Conversion de chaîne de caractères en nombre à virgule flottante

```
str2float (charstring value) return float
```

Cette fonction convertit une valeur de type **`charstring`** comportant un nombre à virgule flottante en valeur de type **`float`**. Le format du nombre contenu dans la valeur de type **`charstring`** doit suivre les règles indiquées dans le § 6.1.0 avec les exceptions suivantes:

- les zéros de gauche sont autorisés,
- le signe '+' de gauche avant des valeurs positives est autorisé,
- '–0,0' est autorisé.

EXEMPLE:

```
str2float('12345.6') //  est identique à str2float('123.456E+02')
```

C.34 La fonction "Replace" (remplacement)

```
replace (in any_string_type str, in integer ind, in integer len, in any_string_type repl)
return any_string_type
```

Cette fonction remplace la sous-chaîne de valeur **str** à l'indice **ind** de longueur **len** par la valeur de chaîne **repl** et retourne la chaîne résultante. La valeur **str** ne doit pas être modifiée. Si **len** est égal à **0,** la chaîne **repl** est insérée. Si **ind** est égal à **0**, **repl** est inséré au début de **str.** Si **ind** est égal à **lengthof(str)**, **repl** est inséré à la fin de **str**. Les valeurs **str** et **repl** doivent être du même type de chaîne et doivent avoir comme type de base **bitstring**, **hexstring**, **octetstring** ou toute chaîne de caractères. La chaîne retournée est du même type que **str** et **repl**. Il est à noter que l'indexation dans les chaînes commence à partir de zéro.

Les cas d'erreur suivants conduiront à une erreur au moment de la compilation ou de l'exécution:

- **str** ou **repl** ne sont pas du type chaîne;
- **str** et **repl** sont de type différent;
- **ind** est inférieur à **0** ou supérieur à **lengthof(str)**;
- **len** est inférieur à **0** ou supérieur à **lengthof(str)**;
- **ind+len** est supérieur à **lengthof(str)**.

EXEMPLE:

```
replace ('00000110'B, 1, 3, '111'B)     // retourne '01110110'B

replace ('ABCDEF'H, 0, 2, '123'H)       // retourne '123CDEF'H

replace ('01AB23CD'O, 2, 1, 'FF96'O)    // retourne '01ABFF96CD'O

replace ("My name is JJ", 11, 1, "xx")  // retourne "My name is xxJ"

replace ("My name is JJ", 11, 0, "xx")  // retourne "My name is xxJJ"

replace ("My name is JJ", 2, 2, "x")    // retourne "Myxame is JJ",

replace ("My name is JJ", 12, 2, "xx")  // produit une erreur de test élémentaire

replace ("My name is JJ", 13, 2, "xx")  // produit une erreur de test élémentaire

replace ("My name is JJ", 13, 0, "xx")  // retourne "My name is JJxx"
```

C.35 Conversion de chaîne d'octets en chaîne de caractères

```
oct2char (octetstring invalue) return charstring
```

Cette fonction convertit une valeur "`invalue`" de type **octetstring** en valeur de type **charstring**. La valeur "`invalue`" du paramètre d'entrée ne doit pas contenir de valeurs supérieures à 7F. Le type résultant **charstring** doit avoir la même longueur que le type d'entrée **octetstring.** Les octets sont interprétés comme des codes de la Rec. UIT-T T.50 [9] (conformément à la version internationale de référence) et les caractères résultants sont ajoutés à la valeur retournée.

EXEMPLE

```
oct2char ('4469707379'O) = "Dipsy"
```

NOTE – La chaîne de caractères retournée peut contenir des caractères non graphiques, qui ne peuvent pas être présentés entre les guillemets droits.

C.36 Conversion de chaîne de caractères en chaîne d'octets

```
char2oct (charstring invalue) return octetstring
```

Cette fonction convertit une valeur "`invalue`" de type **charstring** en valeur de type **octetstring**. Chaque octet du type **octetstring** contiendra les codes de la Rec. UIT-T T.50 [9] (conformément à la version internationale de référence) des caractères appropriés de la valeur "`invalue`".

EXEMPLE

```
char2oct ("Tinky-Winky") = '54696E6B792D57696E6B79'O
```

Annexe D (informative)

Vide

NOTE – Le contenu de la présente annexe a été transféré dans la Rec. UIT-T Z.146 [6].

Annexe E (informative)

Bibliothèque de types utiles

E.1 Limitations

Les noms de type ajoutés à cette bibliothèque devraient être uniques dans l'ensemble du langage et dans la bibliothèque (c'est-à-dire qu'ils ne devraient pas faire partie des noms définis dans l'Annexe C). Les noms définis dans cette bibliothèque ne devraient pas être utilisés par les utilisateurs de la notation TTCN-3 comme identificateurs de définitions autres qu'indiquées dans la présente annexe.

NOTE – Les définitions de type figurant dans la présente annexe peuvent donc être répétées dans des modules TTCN-3 mais aucun type distinct de celui qui est spécifié dans la présente annexe ne peut être défini avec un des identificateurs utilisés dans la présente annexe.

E.2 Types utiles en notation TTCN-3

E.2.1 Simples types utiles de base

E.2.1.0 Entiers signés et non signés d'un seul octet

Ces types prennent en charge les valeurs d'entier contenues dans l'étendue de –128 à 127 pour le type signé et de 0 à 255 pour le type non signé. La notation de valeur pour ces types est la même que la notation de valeur pour le type entier. Les valeurs de ces types doivent être codées et décodées comme si elles étaient représentées par un seul octet dans le système, indépendamment de la forme de représentation effectivement utilisée.

NOTE – Le codage des valeurs de ces types peut être le même ou peut différer d'une valeur à l'autre et peut différer du codage du type integer (le type radical de ces types utiles) selon les règles de codage effectivement utilisées. Les détails des règles de codage sont hors du domaine d'application de la présente Recommandation.

Les définitions de ces types sont les suivantes:

```
type integer   byte     (-128 .. 127)     with { variant "8 bit" };

type integer   unsignedbyte (0 .. 255)    with { variant "unsigned 8 bit" };
```

E.2.1.1 Entiers courts, signés et non signés

Ces types prennent en charge les valeurs d'entier dans l'étendue de –32 768 à 32 767 pour le type signé et de 0 à 65 535 pour le type non signé. La notation de valeur pour ces types est la même que la notation de valeur pour le type entier. Les valeurs de ces types doivent être codées et décodées comme si elles étaient représentées par deux octets dans le système, indépendamment de la forme de représentation effectivement utilisée.

NOTE – Le codage des valeurs de ces types peut être le même ou peut différer d'une valeur à l'autre et peut différer du codage du type entier (integer) (le type radical de ces types utiles) selon les règles de codage effectivement utilisées. Les détails des règles de codage sont hors du domaine d'application de la présente Recommandation.

Les définitions de ces types sont les suivantes:

```
type integer   short        (-32768 .. 32767)   with { variant "16 bit" };

type integer   unsignedshort  (0 .. 65535)      with { variant "unsigned 16 bit" };
```

E.2.1.2 Entiers longs, signés et non signés

Ces types prennent en charge les valeurs d'entier dans l'étendue de –2 147 483 648 à 2 147 483 647 pour le type signé et de 0 à 4 294 967 295 pour le type non signé. La notation de valeur pour ces types est la même que la notation de valeur pour le type entier. Les valeurs de ces types doivent être codées et décodées comme si elles étaient représentées par quatre octets dans le système, indépendamment de la forme de représentation effectivement utilisée.

NOTE – Le codage des valeurs de ces types peut être le même ou peut différer d'une valeur à l'autre et peut différer du codage du type entier (integer) (le type radical de ces types utiles) selon les règles de codage effectivement utilisées. Les détails des règles de codage sont hors du domaine d'application de la présente Recommandation.

Les définitions de ces types sont les suivantes:

```
type integer   long (-2147483648 .. 2147483647)
               with { variant "32 bit" };

type integer   unsignedlong (0 .. 4294967295)
               with { variant "unsigned 32 bit" };
```

E.2.1.3 Entiers à double extension, signés et non signés

Ces types prennent en charge les valeurs d'entier dans l'étendue de –9 223 372 036 854 775 808 à 9 223 372 036 854 775 807 pour le type signé et de 0 à 18 446 744 073 709 551 615 pour le type non signé. La notation de valeur pour ces types est la même que la notation de valeur pour le type entier. Les valeurs de ces types doivent être codées et décodées comme si elles étaient représentées par huit octets dans le système, indépendamment de la forme de représentation effectivement utilisée.

NOTE – Le codage des valeurs de ces types peut être le même ou peut différer d'une valeur à l'autre et peut différer du codage du type entier (integer) (le type radical de ces types utiles) selon les règles de codage effectivement utilisées. Les détails des règles de codage sont hors du domaine d'application de la présente Recommandation.

Les définitions de ces types sont les suivantes:

```
type integer   longlong (-9223372036854775808 .. 9223372036854775807)
               with { variant "64 bit" };

type integer   unsignedlonglong (0 .. 18446744073709551615)
               with { variant "unsigned 64 bit" };
```

E.2.1.4 Nombres en virgule flottante selon l'IEEE 754

Ces types prennent en charge la norme 754 de l'ANSI/IEEE (voir Bibliographie) pour l'arithmétique binaire à virgule flottante. Le type "float" selon l'IEEE 754 prend en charge les nombres en virgule flottantes en base 10, un exposant de grandeur 8, une mantisse de grandeur 23 et un bit de signe. Le type IEEE 754 double prend en charge les nombres en virgule flottante en 10, un exposant de grandeur 11, une mantisse de grandeur 52 et un bit de signe. Le type IEEE 754 `extfloat` prend en charge les nombres en virgule flottante en base 10, un exposant minimal de grandeur 11, une mantisse minimale de grandeur 32 et un bit de signe. Le type IEEE 754 `extdouble` prend en charge les nombres en virgule flottante en base 10, un exposant minimal de grandeur 15, une mantisse minimale de grandeur 64 et un bit de signe.

Les valeurs de ces types doivent être codées et décodées conformément au IEEE 754 définitions. La notation de valeur pour ces types est la même que la notation de valeur pour le type "float" (base 10).

NOTE – Le codage précis des valeurs de ce type dépend des règles effectives de codage utilisé. Les détails des règles de codage sont hors du domaine d'application de la présente Recommandation.

Les définitions de ces types sont les suivantes:

```
type  float   IEEE754float       with { variant "IEEE754 float" };

type  float   IEEE754double      with { variant "IEEE754 double" };

type  float   IEEE754extfloat    with { variant "IEEE754 extended float" };

type  float   IEEE754extdouble   with { variant "IEEE754 extended double" };
```

E.2.2 Types utiles de chaîne de caractères

E.2.2.0 Chaîne de caractères UTF-8 "utf8string"

Ce type prend en charge l'ensemble du jeu de caractères du type TTCN-3 **`universal charstring`** (voir alinéa d du § 6.1.1). Ses valeurs distinctives sont zéro, un, ou plusieurs caractères extraits de ce jeu. Les valeurs de ce type doivent entièrement (par exemple, chaque caractère de la valeur individuellement) être codées et décodées conformément au format de codage du jeu UCS 8 (UTF-8) comme défini dans l'Annexe R de l'ISO/CEI 10646 [10]. La notation de valeur pour ce type est la même que la notation de valeur pour le type **`universal charstring`**.

La définition de ce type est:

```
type universal charstring utf8string with { variant "UTF-8" };
```

E.2.2.1 Chaîne de caractères BMP "bmpstring"

Ce type prend en charge le jeu de caractères de la Table multilingue de base (BMP, *basic multilingual plane*) de l'ISO/CEI 10646 [10]. La table BMP représente tous les caractères du plan 00 et du groupe 00 du jeu universel de caractères codés sur plusieurs octets. Ses valeurs distinctives sont zéro, un, ou plusieurs caractères extraits de la table BMP. Les valeurs de ce type doivent entièrement (par exemple, chaque caractère de la valeur individuellement) être codées et décodées conformément au format de représentation codée du jeu UCS-2 (voir § 14.1 de l'ISO/CEI 10646 [10]). La notation de valeur pour ce type est la même que la notation de valeur pour le type **`universal charstring`**.

NOTE – Le type "bmpstring" prend en charge un sous-ensemble du type TTCN-3 **`universal charstring`**.

La définition de ce type est:

```
type universal charstring bmpstring ( char ( 0,0,0,0 ) .. char ( 0,0,255,255) )
                        with { variant "UCS-2" };
```

E.2.2.2 Chaîne de caractères UTF-16 "utf16string"

Ce type prend en charge tous les caractères des plans 00 à 16 du groupe 00 du jeu universel de caractères codés sur plusieurs octets (voir ISO/CEI 10646 [10]). Ses valeurs distinctives sont zéro, un, ou plusieurs caractères extraits de ce jeu. Les valeurs de ce type doivent entièrement (par exemple, chaque caractère de la valeur individuellement) être codées et décodées conformément au format de codage du jeu UCS 16 (UTF-16) comme défini dans l'Annexe Q de l'ISO/CEI 10646 [10]. La notation de valeur pour ce type est la même que la notation de valeur pour le type **`universal charstring`**.

NOTE – Le type "utf16string" prend en charge un sous-ensemble du type TTCN-3 **`universal charstring`**.

La définition de ce type est:

```
type universal charstring utf16string  ( char ( 0,0,0,0 ) .. char ( 0,16,255,255) )
                         with { variant "UTF-16" };
```

E.2.2.3 Chaîne de caractères ISO/CEI 8859 "iso8859string"

Ce type prend en charge tous les caractères contenus dans tous les alphabets définis dans le norme multilatérale ISO/CEI 8859 (voir Bibliographie). Ses valeurs distinctives sont zéro, un, ou plusieurs caractères extraits du jeu de caractères ISO/CEI 8859. Les valeurs de ce type doivent entièrement (par exemple, chaque caractère de la valeur individuellement) être codées et décodées conformément à la représentation codée qui est spécifiée dans l'ISO/CEI 8859 (codage sur 8 éléments binaires). La notation de valeur pour ce type est la même que la notation de valeur pour le type **`universal charstring`**.

NOTE 1 – Le type "iso8859string" prend en charge un sous-ensemble du type TTCN-3 **`universal charstring`**.

NOTE 2 – Dans chaque alphabet ISO/CEI 8859, la partie inférieure de la table contenant le jeu de caractères (positions 02/00 à 07/14) est compatible avec le jeu de caractères de la Rec. UIT-T T.50 [9]. Tous les caractères complémentaires qui sont propres à une langue ne sont donc définis que pour la partie supérieure de la table de caractères (positions 10/00 à 15/15). Comme le type "iso8859string" est défini comme un sous-ensemble du type TTCN-3 "universal charstring", toute représentation codée de caractères de l'un quelconque des alphabets ISO/CEI 8859 peut être mappée à un caractère équivalent (ayant la même représentation codée lors d'un codage sur 8 bits) extrait des tables de caractères Latin de base ou Latin-1 (Supplément) de l'ISO/CEI 10646 [10].

La définition de ce type est:

```
type universal charstring iso8859string ( char ( 0,0,0,0 ) .. char ( 0,0,0,255) )
                         with { variant "8 bit" };
```

E.2.3 Types utiles structurés

E.2.3.0 Littéral décimal en virgule fixe

Ce type prend en charge l'utilisation d'un littéral décimal en virgule fixe comme défini dans la syntaxe et la sémantique du langage IDL, version 2.6 (voir Bibliographie). Il est spécifié par une partie d'entier, une virgule décimale et une partie fractionnaire. Les deux parties – entier et fraction – se composent d'une séquence de chiffres décimaux (en base 10). Le nombre de chiffres est mémorisé dans l'élément "digits" et la grandeur de la partie fractionnaire est indiquée dans l'élément "scale". Les chiffres proprement dits sont mémorisés dans l'élément "value_". La notation de valeur pour ce type est la même que la notation de valeur pour le type d'enregistrement. Les valeurs de ce type doivent être codées et décodées comme des valeurs décimales en virgule fixe du langage IDL.

NOTE – Le codage précis des valeurs de ce type dépend des règles effectives de codage utilisées. Les détails des règles de codage sont hors du domaine d'application de la présente Recommandation.

La définition de ce type est:

```
type record IDLfixed {
              unsignedshort  digits,
              short   scale,
              charstring  value_
                 }
            with { variant "IDL:fixed FORMAL/01-12-01 v.2.6" };
```

E.2.4 Types utiles de chaîne atomique

E.2.4.1 Type de caractères uniques IRV

Type dont les valeurs distinctives sont les caractères uniques de la Rec. UIT-T T.50 [9] conformes à la Version internationale de référence (IRV, *international reference version*) spécifiée dans le § 8.2/T.50 [9] (voir aussi la Note 2 du § 6.1.1).

La définition de ce type est:

```
type charstring char length (1);
```

NOTE 1 – Le nom de ce type utile est le même que le mot clé TTCN-3 utilisé pour indiquer les valeurs de type **universal charstring** dans la forme quadruplet. En règle générale, il est interdit d'utiliser les mots clés TTCN-3 comme identificateurs. Seul fait exception à cette règle le type utile "char", dont l'utilisation n'est autorisée que pour assurer la rétrocompatibilité avec les versions antérieures de la norme TTCN-3.

NOTE 2 – La chaîne spéciale de l'attribut "8 bit" définie dans le § 28.2.3 peut être utilisée avec ce type pour spécifier tel ou tel codage pour ces valeurs. En outre, l'utilisation des mécanismes des attributs permet de modifier d'autres propriétés du type de base.

E.2.4.2 Type de caractères universels uniques

Type dont les valeurs distinctives sont des caractères uniques de l'ISO CEI 10646 [10].

La définition de ce type est:

```
type universal charstring uchar length (1);
```

NOTE – Les chaines spéciales définies dans le § 28.2.3, à l'exception de l'attribut "8 bit", peuvent être utilisées avec ce type pour spécifier tel ou tel codage pour ces valeurs. En outre, l'utilisation des mécanismes des attributs permet de modifier d'autres propriétés du type de base.

E.2.4.3 Type de bits uniques

Type dont les valeurs distinctives sont des chiffres binaires uniques.

La définition de ce type est:

```
type bitstring bit length (1);
```

E.2.4.4 Type de chiffres hexadécimaux uniques

Type dont les valeurs distinctives sont des chiffres hexadécimaux uniques.

La définition de ce type est:

```
type hexstring hex length (1);
```

E.2.4.5 Type d'octets uniques

Type dont les valeurs distinctives sont des paires de chiffres hexadécimaux.

La définition de ce type est:

```
type octetstring octet length (1);
```

Annexe F (informative)

Opérations sur des objets actifs TTCN-3

F.1 Généralités

La présente annexe décrit sous forme abrégée la sémantique des opérations sur les objets actifs de la notation TTCN-3 que constituent les composants de test, les temporisations et les ports. Ce comportement dynamique est écrit au moyen d'automates à états, dans lesquels:

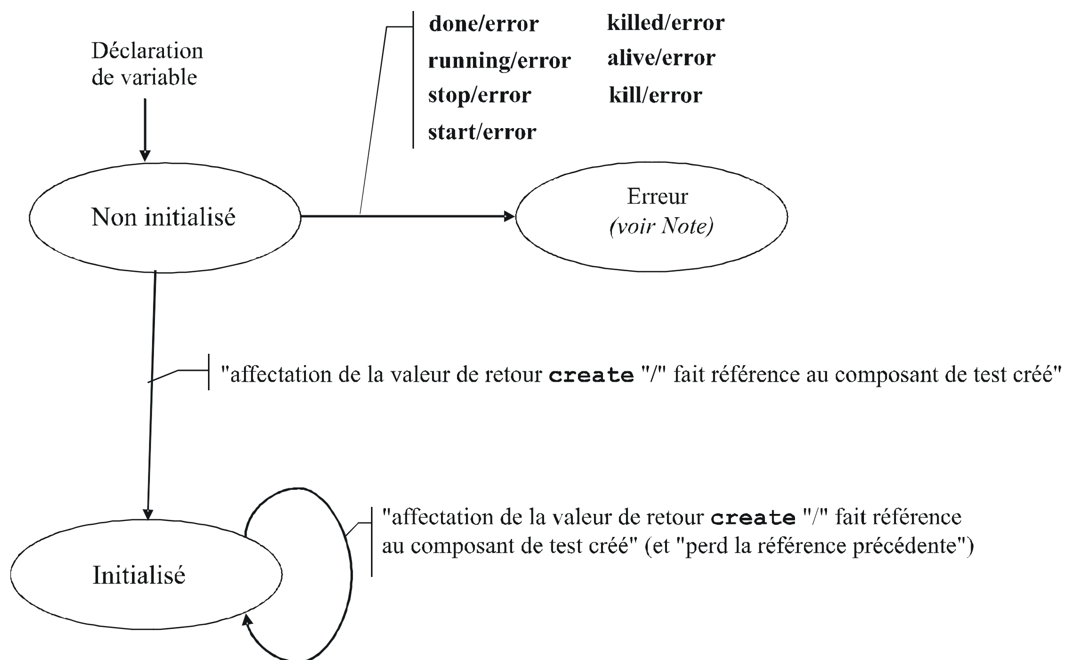
- les états sont nommés et identifiés comme des nœuds;
- l'état initial est identifié par une flèche entrante;
- les transitions entre états connectent deux états (pas nécessairement différents) et sont identifiées par des flèches;
- les transitions sont marquées par la condition d'activation de la transition considérée (c'est-à-dire l'appel d'une opération ou d'une instruction) et la condition résultante (par exemple, une erreur de test élémentaire), l'une et l'autre étant séparées par le symbole '/':
  - les appels d'opérations ou d'instructions sont les opérations et les instructions de la notation TTCN-3 applicables à l'objet (écrit en caractères gras);
  - une erreur comme condition résultante indique une erreur de test élémentaire (écrite en caractères gras);
  - la valeur null comme condition résultante signifie qu'à l'exception d'un éventuel changement d'état, aucun autre résultat ne s'applique (écrite en caractères gras);
  - la valeur match/no match (appariement/non-appariement) fait référence aux résultats d'appariement d'une transition (écrite en caractères gras);
  - les valeurs concrètes sont des résultats booléens ou en virgule flottante (écrites en caractères gras italiques);
  - toutes les autres conditions résultantes sont décrites textuellement (écrites dans une police normalisée);
- des notes explicatives donnent de plus amples précisions sur l'automate à états.

Pour plus de détails, voir la sémantique opérationnelle de la notation TTCN-3 [3]. En cas de contradiction entre la présente annexe et la sémantique opérationnelle de la notation TTCN-3 [3], cette dernière a priorité.

F.2 Composants de test

F.2.1 Références de composant de test

Constituant des variables des types de composant de test, les opérations **`self`** et **`mtc`** sont utilisées pour faire référence à des composants de test. Les opérations **`start`**, **`stop`**, **`done`** et **`running`** ne sont pas directement appliquées aux composants de test, mais aux références de ces composants. Le système de test doit décider si l'opération demandée doit produire l'objet composant lui-même ou si une autre action est souhaitable (par exemple, une erreur se produit quand la référence d'un composant PTC arrêté est utilisée dans une opération de lancement de composant). L'opération **`create`** utilisée pour créer des composants PTC retourne une référence unique au composant PTC créé, qui est généralement associé à une variable de composant de test. Le comportement associé aux variables de composant de test elles-mêmes est représenté dans la Figure F.1.

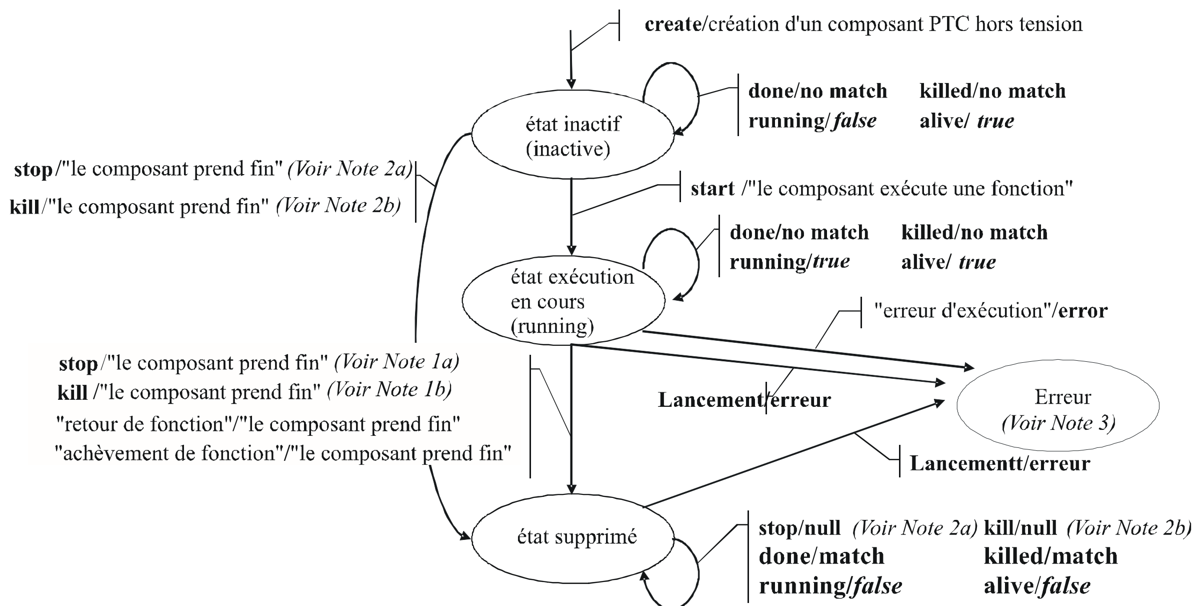


NOTE – Chaque fois qu'un composant de test passe à son état d'erreur, le verdict d'erreur est affecté à son verdict local, le test élémentaire prend fin et le résultat global de celui-ci sera une erreur.

Figure F.1/Z.140 – Traitement des références de composant de test

F.2.2 Comportement dynamique des composants PTC

Les composants PTC peuvent être du type hors tension (non-alive) ou du type sous-tension (alive). Les composants PTC du type hors tension peuvent être dans les états inactif (Inactive), exécution (Running) et supprimé (Killed). Leur comportement dynamique est représenté dans la Figure F.2.



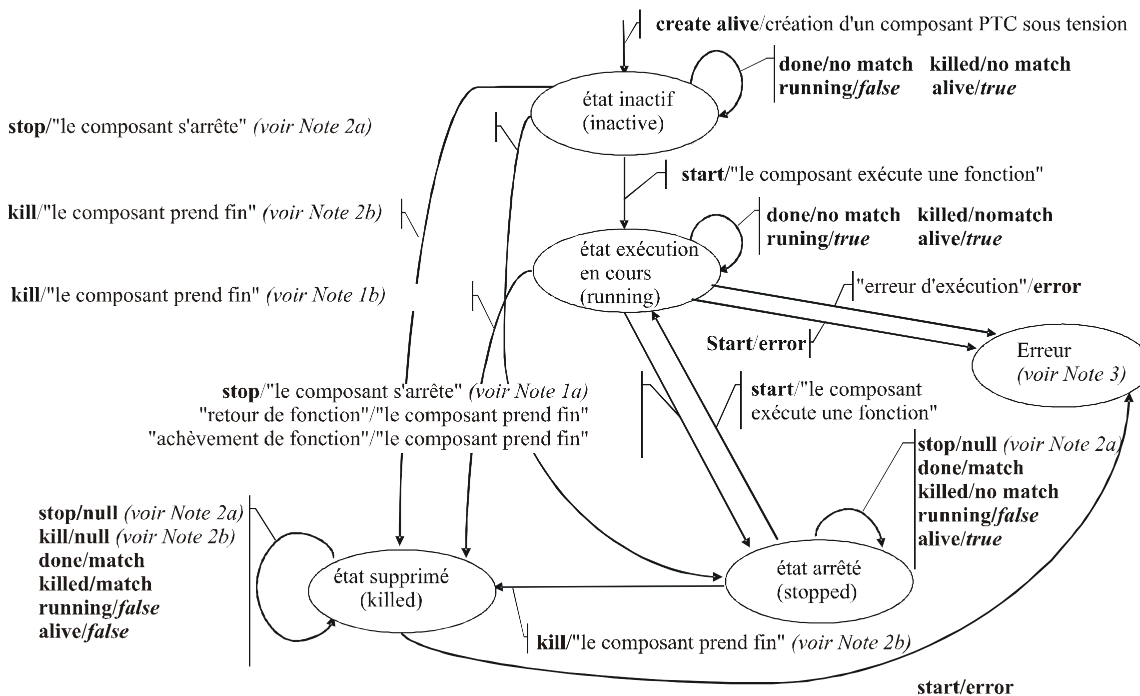
NOTE 1 – a) L'arrêt peut être un arrêt stop, un arrêt automatique (self.stop) ou un arrêt (stop) imputable à un autre composant de test;
b) La suppression peut être une suppression (kill), une suppression automatique (self.kill), une suppression (kill) imputable à un autre composant de test ou une suppression (kill) imputable au système de test (en cas d'erreur).

NOTE 2 – a) L'arrêt (stop) ne peut être imputable qu'à un autre composant de test;
b) La suppression (kill) ne peut être imputable qu'à un autre composant de test ou au système de test (en cas d'erreur).

NOTE 3 – Chaque fois qu'un composant de test passe à son état d'erreur, le verdict d'erreur est affecté à son verdict local, le test élémentaire prend fin et le résultat global de celui-ci sera une erreur.

Figure F.2/Z.140 – Comportement dynamique de composants PTC de type hors tension

Les composants PTC de type sous-tension (alive) peuvent être dans les états inactif (Inactive), exécution (Running), arrêté (Stop) et supprimé (Killed). Leur comportement dynamique est représenté dans la Figure F.3.



NOTE 1 – a) L'arrêt peut être un arrêt (stop), un arrêt automatique (self.stop) ou un arrêt (stop) imputable à un autre composant de test;
b) La suppression peut être une suppression (kill), une suppression automatique (self.kill), une suppression (kill) imputable à un autre composant de test ou une suppression (kill) imputable au système de test (en cas d'erreur).

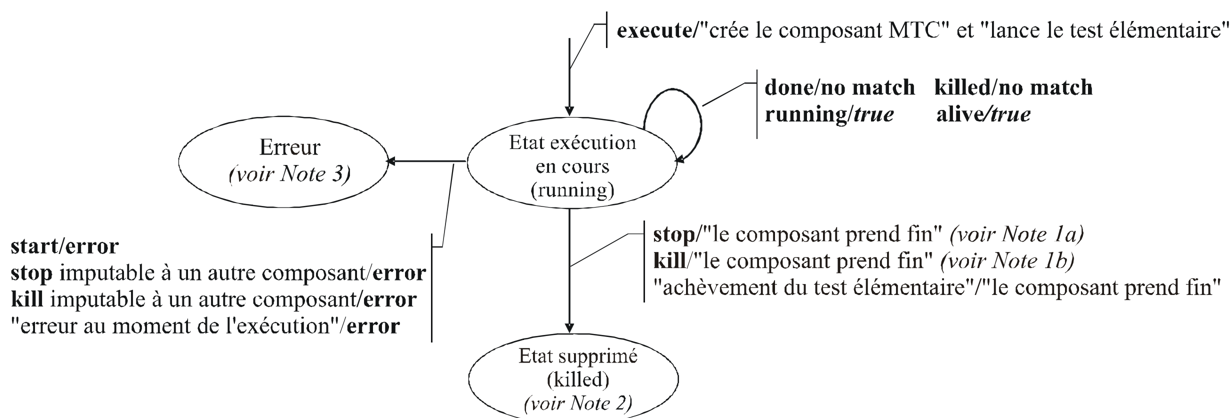
NOTE 2 – a) L'arrêt (stop) ne peut être imputable qu'à un autre composant de test;
b) La suppression (kill) ne peut être imputable qu'à un autre composant de test ou au système de test (en cas d'erreur) .

NOTE 3 – Chaque fois qu'un composant de test passe à son état d'erreur, le verdict d'erreur est affecté à son verdict local, le test élémentaire prend fin et le résultat global de celui-ci sera une erreur.

Figure F.3/Z.140 – Comportement dynamique des composants PTC de type sous-tension

F.2.3 Comportement dynamique du composant MTC

Le composant MTC peut être dans l'état exécution en cours (Running) ou supprimé (Killed). Le comportement dynamique du composant MTC est représenté dans la Figure F.4.



NOTE 1 – a) L'arrêt peut être un arrêt (stop), un arrêt automatique (self.stop) ou un arrêt (stop) imputable à un autre composant de test;

b) La suppression peut être une suppression (kill), une suppression automatique (self.kill), une suppression (kill) imputable à un autre composant de test ou une suppression (kill) imputable au système de test (en cas d'erreur).

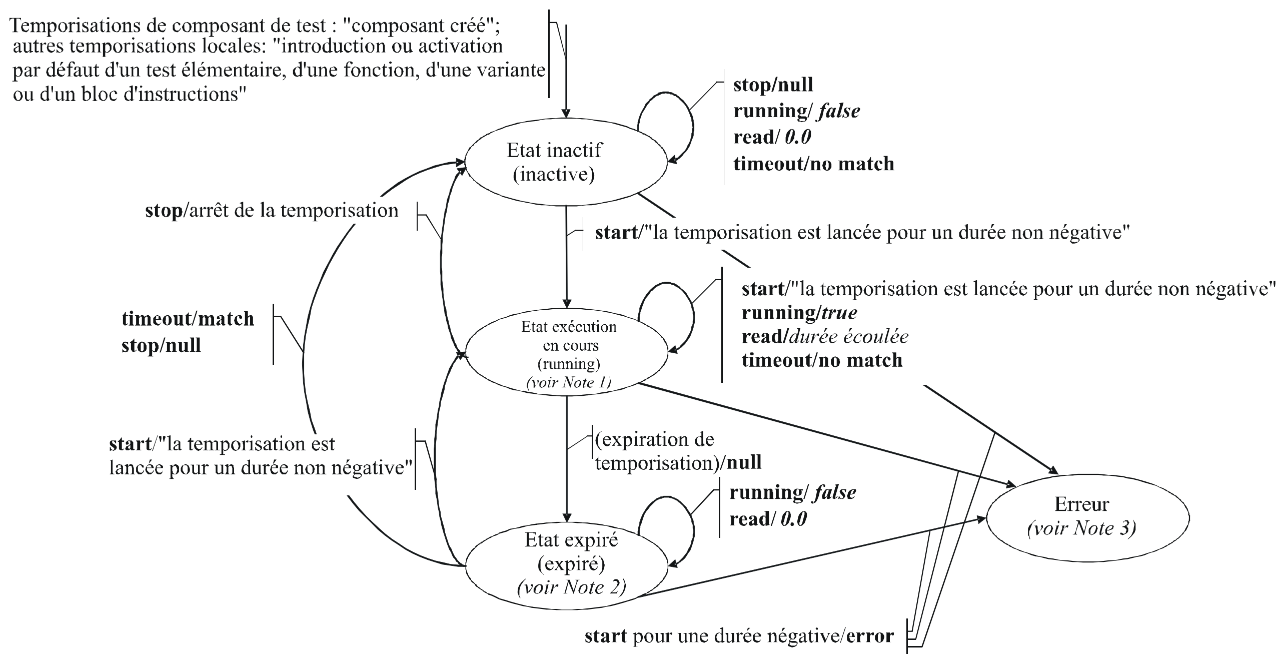
NOTE 2 – Tous les composants PTC restants doivent aussi être supprimés et le test élémentaire prend fin.

NOTE 3 – Chaque fois qu'un composant de test passe à son état d'erreur, le verdict d'erreur est affecté à son verdict local, le test élémentaire prend fin et le résultat global de celui-ci sera une erreur.

Figure F.4/Z.140 – Comportement dynamique du composant MTC

F.3 Temporisations

Les temporisations peuvent être dans l'état inactif (Inactive), exécution en cours (Running) ou expiré (Expired). Le comportement dynamique d'une temporisation est représenté dans la Figure F.5.



NOTE 1 – Pour toute unité de portée, toutes les temporisations de cette portée qui sont dans l'état exécution en cours constituent la liste des temporisations en cours.

NOTE 2 – Pour toute unité de portée, toutes les temporisations de cette portée qui sont dans l'état expiré constituent la liste des temporisations expirées.

NOTE 3 – Chaque fois qu'une temporisation passe à son état d'erreur, le composant de test dont il fait partie passe lui aussi à son état d'erreur, affecte un verdict d'erreur local, le test alimentaire prend fin et le résultat global de celui-ci sera une erreur.

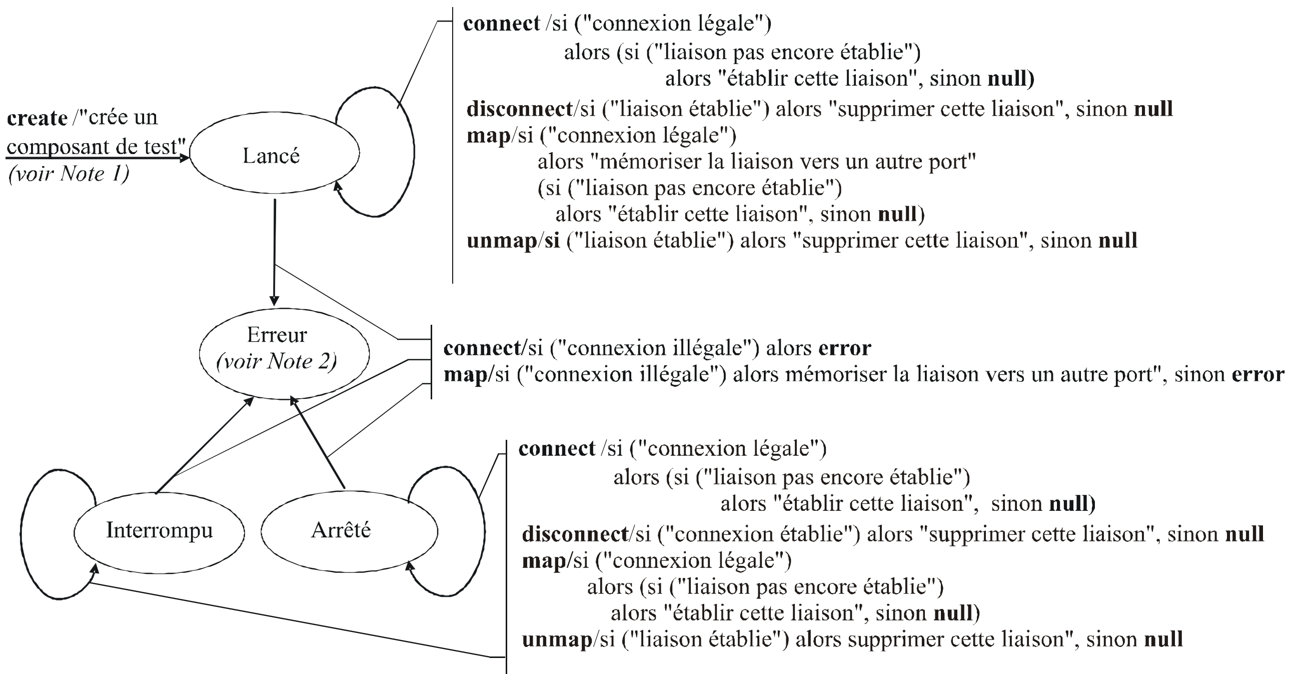
Figure F.5/Z.140 – Comportement dynamique des temporisations

F.4 Ports

Les ports peuvent être dans l'état lancé (Started) ou arrêté (Stopped). Comme leur comportement est relativement complexe, l'automate à états a été subdivisé de manière à indiquer le comportement dynamique des opérations de configuration (c'est-à-dire les opérations connect, disconnect, map et unmap), des opérations de commande de port (c'est-à-dire les opérations start, stop et clear) et des opérations de communication (c'est-à-dire les opérations send, receive, call, getcall, raise, catch, reply, getreply et check). L'opération trigger étant un abrégé de l'instruction alt conjuguée à l'instruction receive, elle n'est pas considérée ici.

F.4.1 Opérations de configuration

Les opérations de configuration de port (c'est-à-dire les opérations connect, disconnect, map et unmap) sont indifférentes à l'état du port. Leur comportement est représenté dans la Figure F.6.



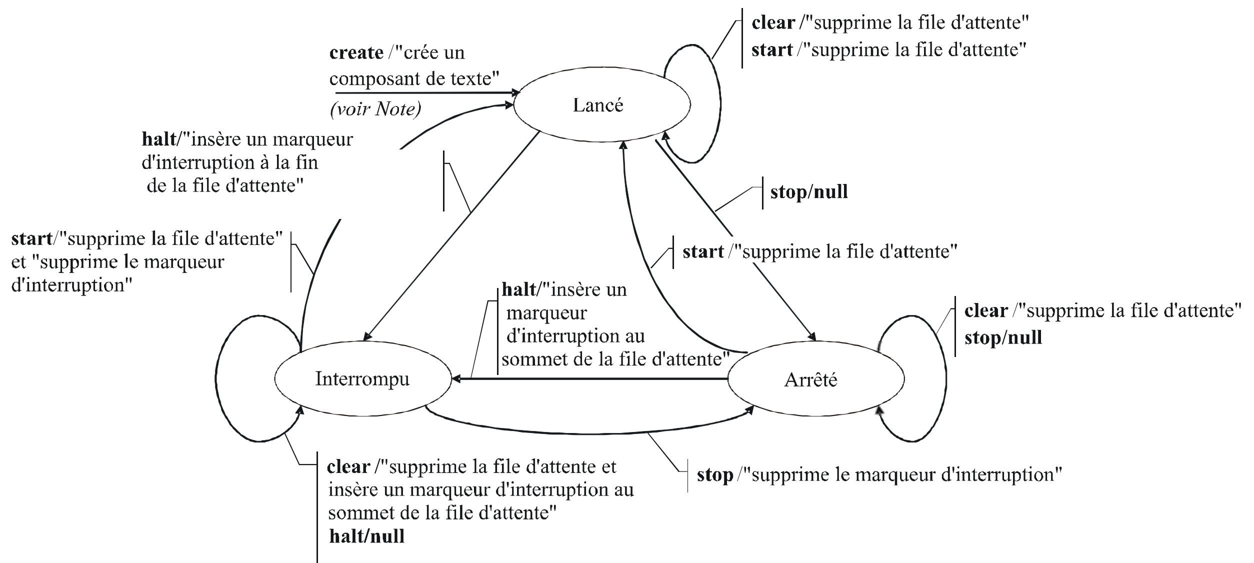
NOTE 1 – Lors de la création d'un composant PTC, les ports de ce composant PTC sont créés et lancés; lors de la création du composant MTC, les ports de ce composant MTC et les ports de l'interface TSI sont créés et lancés.
NOTE 2 – Chaque fois qu'un port passe à son état d'erreur, le composant de test dont il fait partie passe aussi à son état d'erreur, affecte un verdict d'erreur local, le test élémentaire prend fin et le résultat global de celui-ci sera une erreur.

Figure F.6/Z.140 – Comportement dynamique des ports: opérations de configuration de port

Les transitions ne modifient pas l'état principal du port, c'est-à-dire que le port reste dans l'état lancé ou arrêté.

F.4.2 Opérations de commande de port

Les résultats des opérations de commande de port sont indiqués dans la Figure F.7.

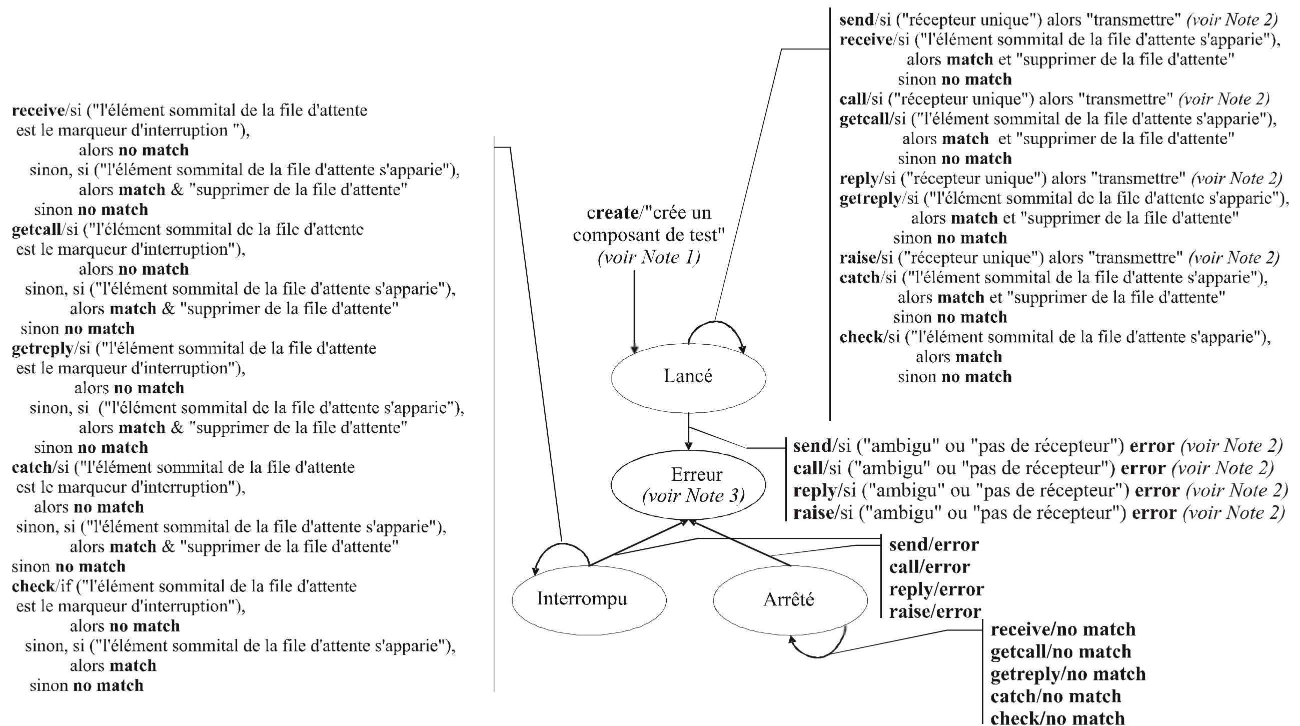


NOTE – Lors de la création d'un composant PTC, les ports de ce composant sont créés et lancés; lors de la création du composant MTC, les ports de ce composant MTC et les ports de l'interface TSI sont créés et lancés.

Figure F.7/Z.140 – Comportement dynamique des ports: opérations de commande de port

F.4.3 Opérations de communication

Les résultats des opérations de communication send, receive, call, getcall, raise, catch, reply, getreply, check sont indiqués dans la Figure F.8.



NOTE 1 – Lors de la création d'un composant PTC, les ports de ce composant PTC sont créés et lancés; lors de la création d'un composant MTC, les ports de ce composant MTC et les ports de l'interface TSI sont créés et lancés.
NOTE 2 – Un récepteur unique existe s'il n'y a qu'une seule liaison pour ce port ou si l'expression d'adressage "to" fait référence à un composant de test dont le port est relié à ce port (un composant de test terminé n'est pas un récepteur légal).
NOTE 3 – Chaque fois qu'un port passe à son état d'erreur, le composant de test dont il fait partie passe aussi à son état d'erreur, affecte un verdict d'erreur local, le test élémentaire prend fin et le résultat global de celui-ci sera une erreur.
NOTE 4 – L'opération trigger étant un abrégé de l'instruction halt conjuguée à l'instruction receive, elle n'est pas considérée ici.

Figure F.8/Z.140 – Comportement dynamique des ports: opérations de communication

Annexe G (informative)

Caractéristiques de langage déconseillées

G.1 Définition des paramètres de module fondée sur des groupes

La version antérieure de la présente Recommandation nécessitait l'utilisation d'une syntaxe fondée sur des groupes, telle qu'elle est représentée dans l'exemple ci-dessous, pour la déclaration des paramètres de module. Dans la présente version, la syntaxe des paramètres de module a été unifiée avec la syntaxe fondée sur la déclaration de constantes et de variables, mais la syntaxe fondée sur des groupes n'est pas intégralement supprimée, afin de laisser aux fournisseurs d'utilitaires et aux utilisateurs un délai pour passer de l'ancienne syntaxe à la nouvelle. La syntaxe fondée sur des groupes pour la déclaration des paramètres de module doit normalement être intégralement supprimée dans la prochaine édition à paraître de la présente Recommandation.

EXEMPLE (syntaxe superflue):

```
module MyModuleWithParameters
{
  modulepar { integer TS_Par0, TS_Par1 := 0;
              boolean TS_Par2 := true
            };
  modulepar { hexstring TS_Par3 };
}
```

G.2 Importation récursive

La version antérieure de la présente Recommandation autorisait l'importation implicite de définitions nommées, par l'importation d'autres définitions du même module utilisant celles-ci dans un mode récursif. Cette caractéristique est déconseillée dans la présente édition de la norme et doit en principe être intégralement supprimée dans la prochaine édition à paraître.

G.3 Utilisation du mot clé `all` dans les définitions de type de port

La version antérieure de la présente Recommandation autorisait l'utilisation du mot clé **`all`** dans les définitions de type de port au lieu d'une liste explicite des types et signatures autorisés via le port considéré. Cette caractéristique est déconseillée dans la présente édition de la norme et doit en principe être intégralement supprimée dans la prochaine édition à paraître.

BIBLIOGRAPHIE

- ETSI ES 201 873-1 V1.1.2 (2001-06), *Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3; Part 1: TTCN-3 Core Language*.
- ETSI ES 201 873-1 V2.2.1 (2003-02), *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*.
- ISO/CEI 8859-1:1998, *Technologies de l'information – Jeux de caractères graphiques codés sur un seul octet – Partie 1: Alphabet latin n° 1*.
- Object Management Group (OMG): *The Common Object Request Broker: Architecture and Specification*, Chapter 3 – IDL Syntax and Semantics. Version 2.6, FORMAL/01-12-01, décembre 2001.
- IEEE 754 (1985), *Binary Floating-Point Arithmetic*.

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