Union internationale des télécommunications

UIT-T

SECTEUR DE LA NORMALISATION
DES TÉLÉCOMMUNICATIONS
DE L'UIT

Z.140

(04/2003)

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

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 à divers points d'accès de communication. Les sortes d'application typiques sont les tests des protocoles (y compris les protocoles de communications mobiles et Internet), les tests des services (y compris les services complémentaires), les tests des modules, 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.

La notation TTCN-3 a évolué à partir de la notation combinée arborescente et tabulaire, version 2 (TTCN-2, *tree and tabular combined notation, edition 2*) définie dans la Rec. UIT-T X.292, mais elle en est syntaxiquement très différente. Contrairement à la notation TTCN-2, 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 conserve l'essentiel de la fonctionnalité de base – bien établie – de la notation TTCN-2, mais a été améliorée afin de comprendre éléments tels que:

– la capacité de spécifier des configurations de test dynamiquement concurrentes;
– les opérations permettant la 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 signatures avec de puissants mécanismes d'appariement;
– le paramétrage du type et de la valeur;
– l'affectation et la manipulation de verdicts de test;
– le paramétrage des suites de tests et des mécanismes des test élémentaire;
– l'utilisation combinée de la notation TTCN-3 avec la notation ASN.1 et éventuellement avec d'autres langages;
– des définitions correctes de la syntaxe, du format d'échange et de la sémantique statique.

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 la mise en œuvre de ces formats de présentation doivent être assurées sur la base du langage noyau.

Depuis la première publication des Recommandations UIT-T Z.140 et Z.141 en juillet 2001, plusieurs mises à jour importantes ont été apportées au langage noyau de la notation TTCN-3 et ont été incluses dans la présente Recommandation:

1) correction de certains exemples et autres rectifications rédactionnelles;
2) restructuration de la Recommandation afin d'en faciliter l'intelligibilité;
3) déplacement de l'Annexe B, Sémantique opérationnelle, dans une spécification distincte;
4) extension des fonctions définies par l'utilisateur;
5) correction de bogues dans le formalisme BNF et incorporation des modifications dues aux mises à jour du formalisme BNF;
6) adjonction de mécanismes d'appariement de formes;
7) amélioration du système de types (y compris une meilleure définition de l'équivalence de types) et adjonction de nouveaux types de caractères;
8) amélioration du mécanisme d'importation;
9) remplacement des variantes nommées par des variantes, avec amélioration de la sémantique.

Source

La Recommandation UIT-T Z.140 a été approuvée le 22 avril 2003 par la Commission d'études 17 (2001-2004) de l'UIT-T selon la procédure définie dans la Recommandation UIT-T A.8.

La présente Recommandation fait partie d'une série de Recommandations couvrant la version 3 de la notation de test et de commande de test, telles qu'indiquées ci-dessous:

– **Z.140 "TTCN-3: Langage noyau";**
– Z.141 "TTCN-3: Format de présentation tabulaire (TFT)";
– Z.142 "TTCN-3: Format de présentation graphique (GFT)".

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 de renseignements les plus récents, il est vivement recommandé aux responsables de la mise en œuvre de consulter la base de données des brevets du TSB.

TABLE DES MATIÈRES

		Page
1	Domaine d'application	1
2	Références normatives	1
3	Définitions et abréviations	2
	3.1 Définitions	2
	3.2 Abréviations	5
4	Introduction	5
	4.0 Généralités	5
	4.1 Langage noyau et formats de présentation	6
	4.2 Unicité de la spécification	7
	4.3 Conformité	7
5	Eléments linguistiques fondamentaux	7
	5.0 Généralités	7
	5.1 Séquencement des éléments linguistiques	9
	5.2 Paramétrage	10
	5.3 Règles de portée	12
	5.4 Identificateurs et mots clés	15
6	Types et valeurs	15
	6.0 Généralités	15
	6.1 Types et valeurs de base	16
	6.2 Sous-typage de types de base	19
	6.3 Types et valeurs structurés	21
	6.4 Le type "Any"	28
	6.5 Séquences tabulaires	29
	6.6 Types récursifs	30
	6.7 Compatibilité des types	30
7	Modules	34
	7.0 Généralités	34
	7.1 Nommage des modules	34
	7.2 Paramètres de module	35
	7.3 Partie d'un module relative aux définitions	35
	7.4 Partie d'un module relative à la commande	37
	7.5 Importation à partir de modules	37
8	Configurations de test	48
	8.0 Généralités	48
	8.1 Modèle de communication entre points d'accès	49
	8.2 Restrictions relatives aux connexions	49
	8.3 Interface avec un système de test abstrait	51
	8.4 Définition des types d'accès de communication	52

		Page
8.5	Définition des types de composant	53
8.6	Adressage d'entités à l'intérieur du système SUT	54
8.7	Références de composant	55
8.8	Définition de l'interface avec le système de test	56
9	Déclaration des constantes	57
10	Déclaration de variables	57
11	Déclaration des temporisations	58
11.0	Généralités	58
11.1	Temporisations utilisées comme paramètres	59
12	Déclaration des messages	59
13	Déclaration des signatures de procédure	59
13.0	Généralités	59
13.1	Signatures pour communications bloquantes et non bloquantes	59
13.2	Paramètres des signatures de procédure	60
13.3	Procédures distantes retournant une valeur	60
13.4	Spécification des exceptions	60
14	Déclaration des modèles	61
14.0	Généralités	61
14.1	Déclaration des modèles de message	61
14.2	Déclaration des modèles de signature	63
14.3	Mécanismes d'appariement de modèles	64
14.4	Paramétrage de modèles	66
14.5	Transmission de modèles comme paramètres	67
14.6	Modèles modifiés	67
14.7	Modification des champs de modèle	69
14.8	Opération d'appariement	69
14.9	Opération de valuation	69
15	Opérateurs	70
15.0	Généralités	70
15.1	Opérateurs arithmétiques	71
15.2	Opérateurs de concaténation	72
15.3	Opérateurs relationnels	72
15.4	Opérateurs logiques	74
15.5	Opérateurs binaires	75
15.6	Opérateurs de décalage	76
15.7	Opérateurs de rotation	76
16	Fonctions et variantes	77
16.1	Fonctions	77
16.2	Variantes	80

			Page
	16.3	Fonctions et variantes pour des types différents de composant	83
17	Tests élémentaires		83
	17.0	Généralités	83
	17.1	Paramétrage de tests élémentaires	84
18	Aperçu général des instructions de programmation et des opérations		84
19	Instructions de programmation de base		87
	19.0	Généralités	87
	19.1	Expressions	87
	19.2	Affectations	88
	19.3	L'instruction "Log"	88
	19.4	L'instruction "Label"	88
	19.5	L'instruction "Goto" (saut)	89
	19.6	L'instruction "If-else" (échappement conditionnel)	90
	19.7	L'instruction for (pour)	90
	19.8	L'instruction "While" (tant que)	91
	19.9	L'instruction "Do-while" (exécution tant que)	91
	19.10	L'instruction "Stop" (arrêt d'exécution)	91
20	Instructions de programmation comportementales		92
	20.0	Généralités	92
	20.1	Comportement à options	92
	20.2	L'instruction "Repeat" (répétition)	97
	20.3	Comportement entrelacé	98
	20.4	L'instruction "Return" (retour)	100
21	Manipulation des valeurs par défaut		100
	21.0	Généralités	100
	21.1	Le mécanisme de comportement par défaut	101
	21.2	Références de valeurs par défaut	101
	21.3	L'opération "Activate" (activation)	102
	21.4	L'opération "Deactivate" (désactivation)	102
22	Opérations de configuration		103
	22.0	Généralités	103
	22.1	L'opération "Create" (création)	104
	22.2	Les opérations "Connect" (connexion) et "Map" (affectation)	104
	22.3	Les opérations "Disconnect" (déconnexion) et "Unmap" (désaffectation)	106
	22.4	Les opérations "MTC", "System" et "Self"	106
	22.5	L'opération "Start" (lancement de composant de test)	107
	22.6	L'opération "Stop" (arrêt de composant de test)	107
	22.7	L'opération "Running" (exécution)	109
	22.8	L'opération "done" (fin d'exécution)	109

			Page
	22.9	Utilisation de séquences tabulaires de composants	110
	22.10	Résumé de l'utilisation de "any" et "all" avec des composants	110
23		Opérations de communication	111
	23.0	Généralités	111
	23.1	Format général des opérations de communication	112
	23.2	Communication en mode message	114
	23.3	Communication en mode procédure	118
	23.4	L'opération "Check" (vérification)	128
	23.5	Opérations de commande des points d'accès de communication	129
	23.6	Utilisation de "Any" et de "All" avec des points d'accès	130
24		Opérations de temporisation	131
	24.0	Généralités	131
	24.1	L'opération "Start timer" (armement de temporisateur)	132
	24.2	L'opération "Stop timer" (désarmement de temporisateur)	133
	24.3	L'opération "Read timer" (lecture de temporisateur)	133
	24.4	L'opération "Running timer" (temporisateur armé)	133
	24.5	L'opération "Timeout" (expiration de temporisateur)	133
	24.6	Résumé de l'utilisation de "Any" et de "All" avec des temporisations	134
25		Opérations de verdict de test	134
	25.0	Généralités	134
	25.1	Verdict de test élémentaire	135
	25.2	Valeurs de verdict et règles de surécriture	135
26		Actions externes	136
27		Partie d'un module relative à la commande	137
	27.0	Généralités	137
	27.1	Exécution de test élémentaire	137
	27.2	Terminaison de test élémentaire	137
	27.3	Contrôle de l'exécution de tests élémentaires	138
	27.4	Sélection de test élémentaire	138
	27.5	Utilisation de temporisateurs dans les commandes	139
28		Spécification des attributs	140
	28.0	Généralités	140
	28.1	Attributs d'affichage	140
	28.2	Codage de valeurs	140
	28.3	Attributs d'extension	143
	28.4	Portée des attributs	143
	28.5	Règles de surécriture pour attributs	144
	28.6	Modification des attributs d'éléments linguistiques importés	145

Page

Annexe A – Formalisme BNF et sémantique statique 146
A.1 Formalisme BNF de la notation TTCN-3 146
Annexe B – Appariement de valeurs entrantes 168
B.1 Mécanismes d'appariement de modèles 168
Annexe C – Fonctions de notation TTCN-3 prédéfinies 176
C.1 Conversion d'entier en caractère 176
C.2 Conversion de caractère en entier 176
C.3 Conversion d'entier en caractère universel 176
C.4 Conversion de caractère universel en entier 176
C.5 Conversion de chaîne binaire en entier 176
C.6 Conversion de chaîne hexadécimale en entier 177
C.7 Conversion de chaîne d'octets en entier 177
C.8 Conversion de chaîne de caractères en entier 177
C.9 Conversion d'entier en chaîne binaire 177
C.10 Conversion d'entier en chaîne hexadécimale 177
C.11 Conversion d'entier en chaîne d'octets 178
C.12 Conversion d'entier en chaîne de caractères 178
C.13 Longueur de type chaîne 178
C.14 Nombre d'éléments contenus dans un type structuré 179
C.15 La fonction "IsPresent" 179
C.16 La fonction "IsChosen" 180
C.17 La fonction "Regexp" 180
C.18 Conversion de chaîne binaire en chaîne de caractères 181
C.19 Conversion de chaîne hexadécimale en chaîne de caractères 181
C.20 Conversion de chaîne d'octets en chaîne de caractères 181
C.21 Conversion de chaîne de caractères en chaîne d'octets 181
C.22 Conversion de chaîne binaire en chaîne hexadécimale 182
C.23 Conversion de chaîne hexadécimale en chaîne d'octets 182
C.24 Conversion de chaîne binaire en chaîne d'octets 182
C.25 Conversion de chaîne hexadécimale en chaîne binaire 183
C.26 Conversion de chaîne d'octets en chaîne hexadécimale 183
C.27 Conversion de chaîne d'octets en chaîne binaire 183
C.28 Conversion d'entier en nombre à virgule flottante 183
C.29 Conversion de nombre à virgule flottante en entier 184
C.30 La fonction de générateur de nombre aléatoire 184
C.31 La fonction de sous-chaîne 184
Annexe D – Utilisation d'autres types de données avec la notation TTCN-3 185
D.1 Utilisation de la notation ASN.1 avec la notation TTCN-3 185

Page

Annexe E (informative) – Bibliothèque de types utiles 194
E.1 Limitations 194
E.2 Types utiles en notation TTCN-3 194
Annexe F (informative) – Bibliographie 198
Annexe G (informative) – Utilisation d'expressions régulières et de mécanismes d'appariement en notation TTCN-3 199
G.1 Est-ce que le caractère qui suit le symbole "" est un métacaractère? 199
G.2 Est-ce que l'astérisque "*" correspond à la plus courte ou à la plus longue séquence de caractères possible? 199
G.3 Est-ce que les métacaractères "?" et "*" correspondent à des fins de ligne? 200
G.4 Quel est le comportement des métacaractères non produits par échappement mais "illégaux"? Est-ce que "ab]" correspond à a-b-right-crochet-droit? 200
G.5 Si foo := "ab", est-ce que la structure séquentielle "{foo}#(2)" correspond à abb ou abab? 200
G.6 Comment le caractère "^" est-il manipulé quand il n'est pas le premier caractère d'un groupe? 200
G.7 Est-ce que les métacaractères sont autorisés dans un ensemble précédé par un ""? 200
G.8 Des ensembles peuvent-ils être imbriqués? 200
G.9 Une expression de référence peut-elle être utilisée à l'intérieur d'un ensemble? 201
G.10 Comment les caractères "?" et "*" sont-ils utilisés dans les expressions régulières de la notation TTCN-3? 201

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 version 3 de la notation TTCN (ou TTCN-3), laquelle peut servir à la spécification de tous les types de test de système réactif à divers points d'accès de communication. Les sortes d'application typiques sont les tests des protocoles (y compris les protocoles de communications mobiles et Internet), les tests des services (y compris les services complémentaires), les tests des modules, les tests des plates-formes en architecture CORBA, les 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 [1] et un format de présentation graphique [2]. La spécification de ces formats est hors du domaine d'application de la présente Recommandation.

La présente Recommandation définit une façon normative d'utiliser, avec la notation TTCN-3, la notation ASN.1 définie dans la série des Recommandations UIT-T X.680 [7], [8], [9] et [10]. L'harmonisation d'autres langages avec la notation TTCN-3 n'est pas couverte par 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) à partir de suites de tests abstraits (ATS) est hors du domaine d'application de la présente Recommandation.

2 Références normatives

La présente Recommandation renvoie à certaines dispositions des Recommandations UIT-T ou à d'autres textes de référence qui en font ainsi partie intégrante. Les versions indiquées étaient en vigueur au moment de la publication de la présente Recommandation. Les Recommandations ou textes de référence énumérés ci-après sont tous sujets à révision; les utilisateurs de la présente Recommandation sont invités à se reporter, si possible, à leurs versions les plus récentes. La liste des Recommandations UIT-T en vigueur est publiée régulièrement. La référence à un document figurant dans cette Recommandation ne donne pas à la présente Recommandation, en tant que telle, le statut d'une Recommandation.

[1] Recommandation UIT-T Z.141 (2003), *Notation de test et de commande de test version 3 (TTCN-3): format de présentation tabulaire.* Cette Recommandation est également disponible comme norme ETSI ES 201 873-2 V2.2.1 (2003-02).

[2] Recommandation UIT-T Z.142 (2003), *Notation de test et de commande de test version 3 (TTCN-3): format de présentation graphique.* Cette Recommandation est également disponible comme norme ETSI ES 201 873-3 V2.2.1 (2003-02).

[3] 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*.

[4] 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)*.

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

[6] ISO/CEI 10646:1993, *Technologies de l'information – Jeu universel de caractères codés sur plusieurs octets (JUC)*.

[7] Recommandation UIT-T X.680 (2002) | ISO/CEI 8824-1:2002, *Technologies de l'information – Notation de syntaxe abstraite numéro un: spécification de la notation de base*.

[8] Recommandation UIT-T X.681 (2002) | ISO/CEI 8824-2:2003, *Technologies de l'information – Notation de syntaxe abstraite numéro un: spécification des objets informationnels*.

[9] Recommandation UIT-T X.682 (2002) | ISO/CEI 8824-3:2002, *Technologies de l'information – Notation de syntaxe abstraite numéro un: spécification des contraintes*.

[10] Recommandation UIT-T X.683 (2002) | ISO/CEI 8824-4:2002, *Technologies de l'information – Notation de syntaxe abstraite numéro un: paramétrage des spécifications de la notation de syntaxe abstraite numéro un*.

[11] Recommandation UIT-T X.690 (2002) | ISO/CEI 8825-1:2002, *Technologies de l'information – Règles de codage ASN.1: spécification des règles de codage de base, des règles de codage canoniques et des règles de codage distinctives*.

[12] Recommandation UIT-T X.691 (2002) | ISO/CEI 8825-2:2002, *Technologies de l'information – Règles de codage ASN.1: spécification des règles de codage compact*.

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

[14] Recommandation UIT-T T.100 (1988), *Echange international d'informations pour le Vidéotex interactif*.

[15] Recommandation UIT-T T.101 (1994), *Interfonctionnement international pour les services Vidéotex*.

[16] Recommandation UIT-T X.660 (1992) | ISO/CEI 9834-1:1993, *Technologies de l'information – Interconnexion de systèmes ouverts – Procédures pour le fonctionnement des autorités d'enregistrement OSI: procédures générales*.

3 Définitions et abréviations

3.1 Définitions

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

3.1.1 paramètre effectif: valeur, 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 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 accès de communication: mécanisme abstrait facilitant la communication entre composants de test.

NOTE – Un point d'accès de communication est modélisé comme une file d'attente directe dans le sens de réception. Les accès peuvent être en mode message, en mode procédure, ou à base d'un mélange des deux modes.

3.1.5 types de données: nom commun désignant les simples types de base, les types concaténés de base, les types structurés, le type de données spécial 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, point d'accès et composant et le type par défaut) ainsi que tous les types définis par l'utilisateur, déclarés soit dans le module ou importés à partir d'un autre module TTCN-3.

3.1.7 paramétrage dynamique: sorte de paramétrage dans laquelle les paramètres effectifs dépendent d'événements d'exécution; p. ex. 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 définie) est propagée par une entité répondante si celle-ci ne peut pas répondre à un appel de procédure distante au moyen de la réponse normalement attendue.

3.1.9 paramètre formel: valeur ou référence nominative de modèle (identificateur) 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): voir la Rec. UIT-T X.290.

3.1.12 informations complémentaires sur l'instance destinées au test (IXIT): voir la Rec. UIT-T X.290.

3.1.13 instance sous test (IUT): voir la Rec. UIT-T X.290.

3.1.14 types connus: ensemble de types définis, de types ASN.1 importés et d'autres types externes importés.

3.1.15 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.

3.1.16 composante de test principal (MTC): voir la Rec. UIT-T X.292.

3.1.17 transmission de paramètre par valeur: mode de transmission de paramètre 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.

3.1.18 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.

3.1.19 composante de test parallèle (PTC): voir la Rec. UIT-T X.292.

3.1.20 type radical: 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.

NOTE – En cas de types fondés sur des types ASN.1 importés, le type radical est déterminé à partir du type TTCN-3 associé (voir § D.1.2).

3.1.21 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 (p.ex. 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).

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

3.1.23 système sous test (SUT): voir la Rec. UIT-T X.290.

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

3.1.24 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.

3.1.25 test élémentaire: voir la Rec. UIT-T X.290.

3.1.26 erreur de test élémentaire: voir la Rec. UIT-T X.290.

3.1.27 suite de tests: module TTCN-3 qui spécifie complètement, explicitement ou implicitement au moyen d'instructions d'importation, toutes les définitions et toutes les descriptions de comportement nécessaires afin de définir complètement un ensemble de tests élémentaires.

3.1.28 système de test: voir la Rec. UIT-T X.290.

3.1.29 interface avec le système de test: composant de test qui fournit une affectation des accès disponibles dans le système de test (abstrait) TTCN-3 vers les accès offerts par un système de test réel.

3.1.30 compatibilité des types: élément linguistique qui permet d'employer des valeurs ou des modèles d'un type donné comme valeurs effectives d'un autre type (p. ex. 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 actuelle doivent tous deux être compatibles avec l'autre type de modèle ou de valeur.

3.1.31 paramétrage en valeur: capacité de transmettre une valeur ou un modèle comme un paramètre effectif vers un objet paramétré.

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

3.1.32 type défini par l'utilisateur: type qui est défini par sous-typage d'un type de base, par déclaration d'un type structuré ou par forçage du type "anytype" en type simple au moyen d'une notation à points.

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

3.1.33 notation de valeur: notation par laquelle un identificateur est associé à une valeur donnée ou à une étendue 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*)
ASN.1	notation de syntaxe abstraite numéro un (*abstract syntax notation one*)
ASP	primitive de service abstraite (*abstract service primitive*)
ATS	suite de tests abstraits (*abstract test suite*)
BNF	formalisme de Backus-Naur (*Backus-Naur form*)
CORBA	architecture 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*)
IDL	langage de définition d'interface (*interface definition language*)
IUT	instance sous test (*implementation under test*)
MTC	composante de test principal (*master test component*)
PDU	unité de données protocolaire (*protocol data unit*)
(P)ICS	déclaration de conformité d'une instance (de protocole) (*(protocol) implementation conformance statement*)
(P)IXIT	informations complémentaires sur l'implémentation (de protocole) destinées au test (*(protocol) implementation extra information for testi*ng)
PTC	composante de test parallèle (*parallel test component*)
SUT	système sous test (*system under test*)
TTCN	notation de test et de commande de test (*testing et test control notation*)

4 Introduction

4.0 Généralités

La notation TTCN-3 est un langage flexible et puissant qui est applicable à la spécification de tous les types de tests sur système réactif, à diverses interfaces de communication. Les sortes d'application typiques sont les tests des protocoles (y compris les protocoles de communication mobile et Internet), les tests des services (y compris les services complémentaires), les tests des modules, les tests de plates-formes en architecture CORBA, les 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.

Du point de vue syntaxique, la notation TTCN-3 est très différente des versions antérieures du langage défini dans la Rec. UIT-T X.292 [4]. Cependant, l'essentiel de la fonctionnalité de base – bien établie – de la notation TTCN a été conservé et a été, dans certains cas, amélioré. La notation TTCN-3 comprend les caractéristiques essentielles ci-après:

- la capacité de spécifier des configurations de test dynamiques en concurrence;
- 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 du type et de la valeur;
- l'affectation et la manipulation 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 la notation ASN.1 (et la possibilité d'utiliser d'autres langages comme l'IDL);
- des définitions correctes de la syntaxe, du format d'échange et de la sémantique statique;
- différent formats de présentation (p. ex. tabulaire et graphique);
- un algorithme d'exécution précis (sémantique opérationnelle).

4.1 Langage noyau et formats de présentation

Historiquement, la notation TTCN a toujours été associée aux essais de conformité. Afin d'ouvrir ce langage à une plus large étendue d'applications d'essais aussi bien dans le domaine des normes que dans celui de l'industrie, la présente Recommandation subdivise la spécification de la notation TTCN-3 en plusieurs parties. La première, définie dans la présente Recommandation, est le langage noyau. La deuxième, définie dans la Rec. UIT-T Z.141 [1], est le format de présentation tabulaire, d'apparence et de fonctionnalité similaires aux versions antérieures de la notation TTCN. La troisième partie, définie dans la Rec. UIT-T Z.142 [2], est le format de présentation graphique. La sémantique opérationnelle du langage peut être consultée dans la norme ETSI ES 201 873-4 (voir Annexe F).

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 entre utilitaires TTCN;

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

Le langage noyau peut être utilisé indépendamment des formats de présentation. Ni le format de présentation tabulaire ni le format de présentation graphique ne peuvent cependant être utilisés sans le langage noyau. L'utilisation et la mise en œuvre 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 qui pourront être normalisés ou qui seront propres à un constructeur, selon les définitions données par les utilisateurs de la notation TTCN-3 eux-mêmes. Ces formats additionnels ne sont pas définis dans la présente Recommandation.

La notation TTCN-3 est pleinement harmonisée avec la notation ASN.1, qui peut facultativement être utilisée avec des modules TTCN-3 sous la forme d'une variante syntaxique de type et de valeur de données. L'utilisation de la notation ASN.1 dans des modules TTCN-3 est définie dans l'Annexe D. La méthode employée afin de combiner les notations ASN.1 et TTCN-3 pourra être

appliquée de façon à faciliter l'utilisation, avec la notation TTCN-3, d'autres systèmes à types et valeurs. Les détails n'en sont pas cependant pas définis dans la présente Recommandation. (Voir Figure 1.)

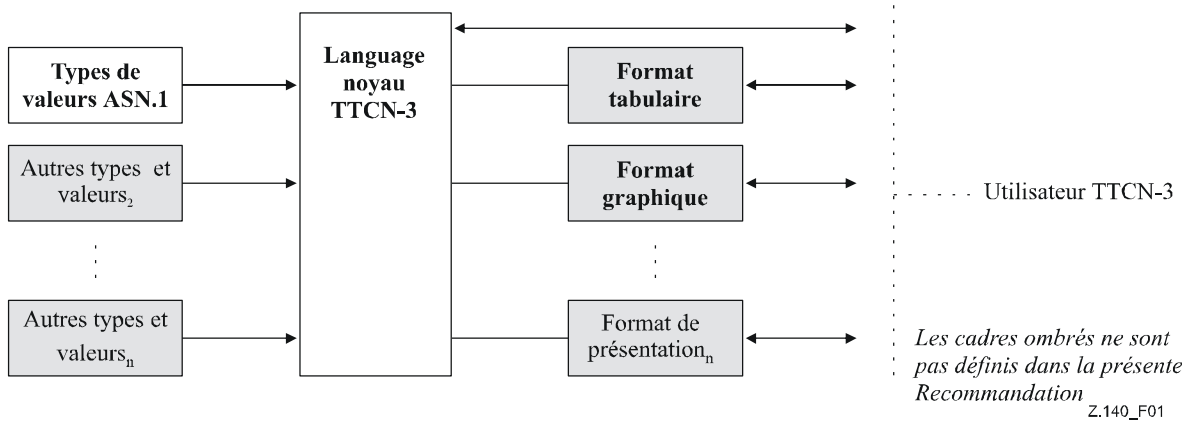

Figure 1/Z.140 – Vue d'utilisateur du langage noyau et des divers formats de présentation

Le langage noyau est défini par une syntaxe complète (voir Annexe A) et par une sémantique opérationnelle (ETSI ES 201 873-4). Il contient une sémantique statique minimale (reproduite dans le corps de la présente Recommandation et dans son Annexe A) qui ne limite pas l'emploi du langage en raison d'un domaine d'application sous-jacent ou d'une méthode d'application sous-jacente. La fonctionnalité des versions antérieures de la notation TTCN, comme les indices de suite de tests, que l'on peut réaliser au moyen d'utilitaires non normalisés, ne fait pas partie de la notation TTCN-3.

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

La présente Recommandation ne spécifie pas de niveaux d'implémentation pour le langage. Cependant, pour une implémentation revendiquant la conformité à la présente version du langage, tous les éléments de la présente Recommandation implémentés doivent être compatibles avec les exigences figurant dans la présente Recommandation.

NOTE – Cette règle n'interdit à aucune implémentation conforme de réaliser des éléments supplémentaires, non spécifiés dans la présente Recommandation.

5 Eléments linguistiques 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 listes de paramètres de façon à offrir une sorte de paramétrage des suites de tests qui est similaire aux mécanismes de paramétrage PICS et PIXIT de la notation TTCN-2.

Un module se compose d'une partie définitions et d'une partie commande. La partie définitions d'un module définit les composants de test, les points d'accès de communication, les types de données, les constantes, les modèles de données de test, les fonctions, les signatures des appels de procédure aux accès, 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 séquences tabulaires. Les types et valeurs ASN.1 importés peuvent être utilisés avec la notation TTCN-3.

Une sorte spéciale de structure séquentielle de données, appelée "modèle", offre des mécanismes de paramétrage et d'appariement permettant de spécifier des données de test à envoyer ou à recevoir par les accès de test. Les opérations effectuées à ces accès fournissent des capacités de communication aussi bien 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 comportementale comme la réception en variante d'événements de communication et de temporisation, l'entrelacement et le comportement par défaut. Les mécanismes d'affectation de verdict de test affectation et de journalisation sont également pris en charge.

Finalement, les éléments linguistiques de la notation TTCN-3 (voir Tableau 1) peuvent être affectés d'attributs comme ceux d'information de codage et ceux d'affichage. Il est également possible de spécifier des attributs définis par l'utilisateur (non normalisés).

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

Elément linguistique	Mot clé associé	Spécifié dans la partie définitions du module	Spécifié dans la partie commande du module	Spécifiés dans fonctions/ variantes/ tests élémentaires	Spécifiés dans 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 d'accès 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

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

Elément linguistique	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 composant de test
Définitions de modèle de données/signature	**template**	Oui			
Définitions de fonction	**function**	Oui			
Définitions de variante	**altstep**	Oui			
Définitions de test élémentaire	**testcase**	Oui			
Déclarations de variable	**var**		Oui	Oui	Oui
Déclarations de temporisateur	**timer**		Oui	Oui	Oui

5.1 Séquencement des éléments linguistiques

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;
     :
}

 :
```

5.1.1 Références anticipées

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

Celles-ci ne sont autorisées que pour des déclarations contenues dans la partie d'un module relative aux définitions. Des références anticipées ne doivent jamais être insérées à l'intérieur de partie d'un module relative à la commande, dans les définitions de tests élémentaires, dans les fonctions et dans les variantes. Autrement dit, il ne doit jamais y avoir de références anticipées à des variables locales, à des temporisateurs locaux et à des constantes locales. La seule exception à cette règle concerne les étiquettes. Des références anticipées à des étiquettes peuvent être utilisées dans des instructions **`goto`** afin d'effectuer un saut en avant (voir § 19.5).

5.2 Paramétrage

5.2.0 Paramétrage statique et paramétrage dynamique

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

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

b) l'élément linguistique **`module`** permet un paramétrage *statique* en valeur afin de prendre en charge des paramètres de suite de tests; c'est-à-dire que ce paramétrage peut éventuellement être résolue 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 paramétriques 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* en valeur, c'est-à-dire que ce paramétrage doit être résolue au moment de la compilation;

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

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

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

Mot clé	Paramétrage en valeur	Types de valeurs autorisés à apparaître dans les listes de paramètres formels/effectifs
module	Statiqeu 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 à l'exécution	*Valeurs de:* tous types de base, tous types définis par l'utilisateur, type **`address`** et **`template`**.
function	Dynamique à 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 à 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 à l'exécution	*Valeurs de:* tous types de base et tous types définis par l'utilisateur, types **`address`** et **`template`**.
signature	Dynamique à l'exécution	*Valeurs de:* tous types de base, tous types définis par l'utilisateur et types **`address`** et **`component`**.
NOTE 1 – Les définitions des types **`record of`**, **`set of`**, **`enumerated`**, **`port`**, composant et **`subtype`** ne permettent pas le paramétrage. NOTE 2 – Des exemples de syntaxe et d'utilisation spécifique du paramétrage avec les différents éléments linguistiques 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 concaténés 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, les mots clé **out** ou **inout** doit être utilisé.

Les temporisateurs et les accès sont toujours transmis par référence. Ils sont identifiés par les mots clés **timer** et **port**. Le mot clé **inout** peut facultativement 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 **altsteps** appelées explicitement, à des fonctions, à des signatures et à des tests élémentaires (**functions**, **signatures, testcase**) 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).

b) les paramètres effectifs ne doivent être que des variables (et non des constantes ou des modèles, par exemple).

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 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 linguistiques 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

Généralement, toutes les entités paramétrées qui sont spécifiées sous forme de paramètre effectif doivent avoir leur propres paramètres résolus dans la liste des paramètres effectifs.

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
{    :
     TC001(MyTemplate(7));
     :
}
```

5.3 Règles de portée

5.3.0 Généralités

La notation TTCN-3 offre six 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 détection de portée est donnée dans le § 7.3.1 pour les groupes.

NOTE 2 – Une règle supplémentaire de détection 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, de fonctions, de tests élémentaires, de variantes et de "blocs d'instructions et de déclarations" contenus dans des instructions composites peuvent en outre spécifier une certaine forme de comportement au moyen de les 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 d'un module relative à la commande 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 eux, c'est-à-dire que les déclarations effectuées au début de leur corps ont une visibilité locale et ne peuvent être utilisées que dans le test élémentaire, la variante **`altstep`** ou la fonction indiquée (p. ex. 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, p. ex. **`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, p. ex. 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.

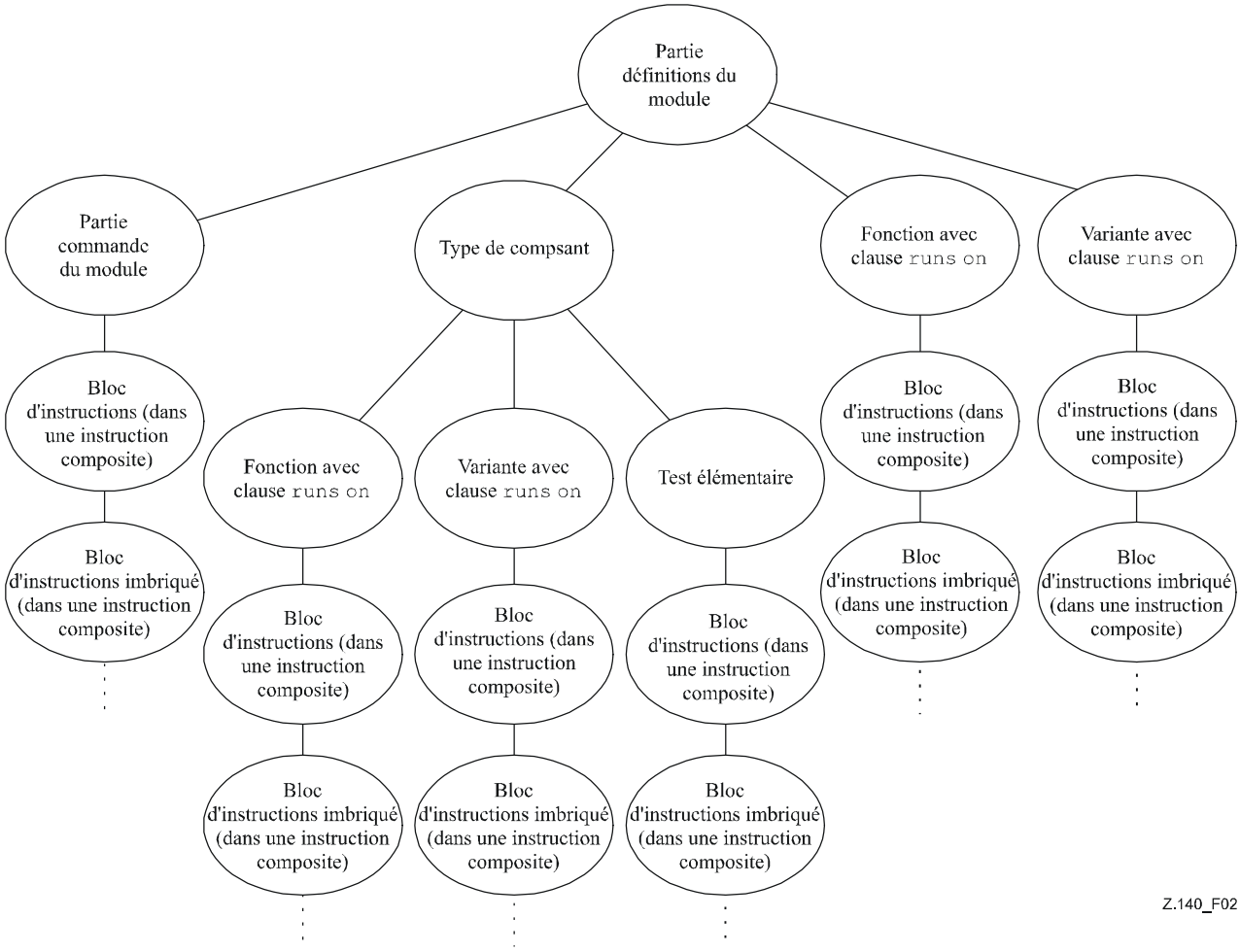

Figure 2/Z.140 – Hiérarchie des unités de portée

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 linguistique paramétré (p. ex. dans un appel de fonction) doit être limitée à la définition dans laquelle ces paramètres apparaissent et aux

niveaux inférieurs de la même portée hiérarchique. Autrement dit, ils suivent les règles normales de portée (voir § 5.4).

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 portée hiérarchique 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 portée hiérarchique. 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
                    :
          }
     }
}

// Les constantes suivantes SONT autorisées car elles ne sont pas déclarées
// dans la même portée hiérarchique (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

En notation TTCN-3, les identificateurs sont sensibles à l'inversion majuscules/minuscules et les mots clés doivent y 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 en 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 concaténés, ainsi que certains types TTCN-3 spécifiques comme **objid** et **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 connus 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 à la manipulation des valeurs par défaut (voir § 21).

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

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

Classe de type	Mot clé	Sous-type
Simples types de base	**integer**	étendue, liste
	char	étendue, liste
	universal char	étendue, liste
	float	étendue, liste
	boolean	liste
	objid	liste
	verdicttype	liste
Types concaténés de base	**bitstring**	liste, longueur
	hexstring	liste, longueur
	octetstring	liste, longueur
	charstring	étendue, liste, longueur
	universal charstring	étendue, liste, longueur
Types structurés	**record**	liste
	record of	liste, longueur
	set	liste
	set of	liste, longueur
	enumerated	liste
	union	liste
Types spéciaux de données	**anytype**	liste
Types de configuration spéciaux	**address**	
	port	
	component	
Types par défaut spéciaux	**default**	

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) **`char`**: type dont les valeurs distinctives sont les caractères de la Rec. UIT-T T.50 [5] conformes à la Version internationale de référence (IRV, *international reference version*) spécifiée dans le 8.2/T.50 [5].

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

Les opérateurs relationnels d'égalité (`==`) et d'inégalité (`!=`) peuvent servir à comparer des valeurs de type **`char`**.

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

Les valeurs du type **`universal char`** peuvent être englobées dans des guillemets droits ("), ou être calculées au moyen d'une fonction de conversion prédéfinie avec la valeur d'entier positif de leur codage comme argument ou par un "quadruplet". Celui-ci ne peut désigner qu'un seul caractère, au moyen des valeurs décimales de son groupe, de son plan, de sa rangée et de sa cellulle conformément à l'ISO/CEI 10646 [6]. Ces valeurs sont précédées par le mot clé **`char`**, puis sont incluses entre des parenthèses et sont séparées par des virgules (p. ex. **`char`** (0, 0, 1, 113) indique le caractère hongrois "ű").

NOTE 1 – Les caractères de commande ne peuvent être indiqués qu'au moyen de la forme quadruple.

Par défaut, le type **`universal char`** doit être conforme à la représentation codée du jeu UCS-4 spécifié dans le § 14.2 de l'ISO/CEI 10646 [6]. Ce codage par défaut peut être neutralisé au moyen des attributs de codage définis (voir § 28.2.1).

NOTE 2 – Le jeu UCS-4 est un format de codage qui représente tout caractère UCS sur un champ de longueur fixe de 32 éléments binaires.

Les opérateurs relationnels d'égalité (`==`) et d'inégalité (`!=`) peuvent servir à comparer des valeurs de type **`universal char`**.

d) **`float`**: type permettant de décrire des nombres en virgule flottante.

Les nombres en virgule flottante sont représentés par: <*mantisse*> × <*base*>$^{<exposant>}$,

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

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

– la notation normale avec un point 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 $-123456789 \times 10^{-6}$); 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 $12{,}3 \times 10^{4}$) ou –12.3E-4 (qui représente $-12{,}3 \times 10^{-4}$).

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

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

f) **`objid`**: type dont les valeurs distinctives sont l'ensemble de tous les identificateurs d'objet conformes au § 6.2/X.660 [16]. Dans les identificateurs d'objet, les tirets sont remplacés par des soulignements.

Exemple:

{itu_t(0) identified_organization(4) etsi(0)}

ou, en variante {itu_t identified_organization etsi}

ou, en variante { 0 4 0}

g) **verdicttype**: type à utiliser avec des verdicts de test composés de 5 valeurs distinctives.

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

6.1.1 Types et valeurs concaténés de base

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

NOTE 1 – Le terme général "string" ou "string type" 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** doivent être indiquées par un nombre arbitraire (éventuellement zéro) constitué des 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** doivent être indiquées par un nombre arbitraire (éventuellement zéro) constitué des chiffres hexadécimaux suivants:

0 1 2 3 4 5 6 7 8 9 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 semi-octet en représentation hexadécimale.

Exemple 2:

'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** doivent être indiquées par un nombre arbitraire, mais pair (éventuellement zéro) constitué des chiffres hexadécimaux suivants:

0 1 2 3 4 5 6 7 8 9 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 semi-octet en représentation hexadécimale.

Exemple 3:

'FF96'O

d) **charstring**: type dont les valeurs distinctives sont zéro, un, ou plusieurs caractères de la Rec. UIT-T T.50 [5], conformes à la Version internationale de référence (IRV) comme spécifié dans le § 8.2/T.50 [5].

Le type chaîne de caractères, précédé par le mot clé **universal**, indique des types dont les valeurs distinctives sont zéro, un, ou plusieurs caractères extraits de l'ISO/CEI 10646 [6].

Les valeurs du type **charstring** doivent être indiquées par un nombre arbitraire (éventuellement zéro) de caractères extraits du jeu de caractères approprié, précédé et suivi par un guillemet droit(").

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 d'espacement intermédiaires.

Exemple 4: ""abcd"" représente la chaîne de littéraux"abcd".

Le type **universal charstring** peut également être indiqué par un nombre arbitraire (éventuellement zéro) de caractères extraits du jeu de caractères approprié, précédé et suivi par un guillemet droit(") ou par un "quadruplet". Celui-ci ne peut indiquer qu'un seul caractère, par les valeurs décimales de son groupe, de son plan, de sa rangée et de sa cellule conformément à l'ISO/CEI 10646 [6], précédées par le mot clé **char,** incluses dans une paire de parenthèses et séparées par des virgules (p. ex. **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 d'espacement intermédiaires. Ces deux méthodes peuvent être mixtes en une seule notation pour une même 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 de littéraux: "le caractère Braille ressemble à cela: ⣿".

NOTE 2 – Les caractères de commande ne peuvent être indiqués qu'au moyen de la forme quadruple.

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 [6].

NOTE 3 – Le jeu UCS-4 est un format de codage qui représente chaque caractère sur un champ de longueur fixe de 32 bits.

Ce codage par défaut peut être neutralisé au moyen des attributs de codage définis (voir § 28.2.1). Les types concaténés 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 éléments individuels d'un type chaîne peuvent être manipulés au moyen d'une syntaxe de type séquence tabulaire. Seuls les éléments isolés de la chaîne peuvent être manipulés.

Les unités de longueur de différent éléments de type 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 indiqué 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 simples de base et des types concaténés de base, conformément au Tableau 3.

6.2.1 Listes de valeurs

La notation TTCN-3 permet la spécification d'une liste de valeurs distinctives de type quelconque comme énuméré dans le Tableau 3. Les valeurs contenues dans la liste doivent être du type radical et doivent être un sous-ensemble vrai des valeurs définies par le type radical. 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 universal char SpecialLetter (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 d'une étendue de valeurs de type **integer, char, universal char** et **float** (ou des dérivés de ces types). Le sous-type défini par cette étendue limite les valeurs autorisées du sous-type à celles qui se trouvent dans l'étendue située entre la limite inférieure et la limite supérieure, incluses. Dans le cas des types **char** et **universal char**, les limites doivent prendre des valeurs valides de position de caractère, conformément à la ou aux tables du jeu codé de caractères de ces types (p. ex. la position donnée ne doit pas être vide). Des positions vides entre les limites inférieure et supérieure ne sont pas considérées d'être des valeurs valides de l'étendue spécifiée.

Exemple 1:

```
type integer MyIntegerRange (0 .. 255);
type char MyCharRange ("a" .. "z");
type float piRange (3.14 .. 3142E-3);
```

La spécification d'étendue du type **char** peut également être utilisée dans des définitions de sous-type du type **charstring** et l'étendue du type **universal char** peut être utilisée dans des définitions de sous-type du type **universal charstring**. Dans ces cas, l'étendue limite les valeurs autorisées pour chaque caractère distinct contenu dans les chaînes.

Exemple 2:

```
type charstring MyCharString ("a" .. "z");
// Définit un type chaîne de longueur quelconque pour chaque caractère
// contenu dans l'étendue spécifiée
type universal charstring MyUCharString1 ("a" .. "z");
// Définit un type chaîne de longueur quelconque pour chaque caractère
// contenu dans l'étendue spécifiée, au moyen de la notation par guillemet
// droit
type universal charstring MyUCharString2 (char(0, 0, 1, 111) .. char(0, 0,
1, 113));
// Définit un type chaîne de longueur quelconque pour chaque caractère
// contenu dans l'étendue spécifiée, au moyen de la notation quadruple.
```

6.2.2.1 Étendues 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'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

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

Exemple:

```
type integer MyIntegerRange (1, 2, 3, 10 .. 20, 99, 100);
type char MyCharRange ("a", "b", "c", "0" .. "9");
```

Dans le cadre de la définition des sous-types de **charstring** et **universal charstring**, listes et étendues ne doivent pas être mélangées dans la même définition de sous-type.

6.2.3 Restrictions de longueur de chaîne

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

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 types différents concaténés.

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

Type	Unités 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.

6.3 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 parr 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 := "Une chaîne"
}

// Ou:
const MyRecordType MyRecordValue:= {'11001'B, true, "Une chaîne"}
//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. Si l'on utilise la notation à liste de valeurs, tous les champs contenus dans la structure séquentielle doivent être spécifiés, soit avec une valeur ou avec le symbole de non-utilisation "-" ou avec le mot clé **omit**.

Exemple 2:

```
var MyRecordType MyVariable:=                //notation par affectation
{
     field1 := '11001'B,
     field3 := "Une chaîne"
}

// Ou:
var MyRecordType MyVariable:= {'11001'B, -, "Une chaîne"}
//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,
"Une chaîne"}
```

Dans les deux notations -par affectation et par liste de valeurs- les champs facultatifs doivent être omis au moyen de la valeur explicite omettre dans le champ approprié. Le fait d'omettre un champ fait que la valeur du champ approprié devient indéfinie quelle que soit la valeur qu'il avait auparavant. Le mot clé omettre ne doit pas être utilisé pour des champs obligatoires.

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 séquences tabulaires). 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 ou indirectement.

```
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 1:

```
type record MyEmptyRecord {  }
```

Une valeur de type **record** est affectée sur la base de chaque élément individuel.

Exemple 2:

```
var integer MyIntegerValue:= 1;

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

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

Ou au moyen d'une liste de valeurs.

Exemple 3:

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

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

Exemple 4:

```
MyRecordValue:= {MyIntegerValue, omettre , "Une chaîne"};

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

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 doivent être spécifiés au moyen du mot clé **optional**.

Exemple:

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

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** champs 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;
// Noter que le type Ensemble, dont le champ est désigné par
// l'identificateur "myElement2", est imbriqué dans un type Enregistrement.
```

6.3.2.2 Eléments facultatifs dans un ensemble

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

6.3.3 Enregistrements et ensembles de types particuliers

La notation TTCN-3 prend en charge la spécification d'enregistrements et d'ensembles dont les éléments sont tous du même type. 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 séquence tabulaire ordonnée et à une séquence tabulaire 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..infini) of integer MyRecordOfType;
// enregistrement d'au moins 10 entiers

type set of boolean MySetOfType;
// 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 séquences tabulaires; voir § 6.5).

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 (p. ex. 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 à droite qu'à 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 à gauche d'une affectation se rapporte à un élément inexistant, la valeur de 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** valeur 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 };

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

// Des valeurs indexées sont également autorisées à gauche des
affectations:
MyRecordVar[1] := MyVar; // La value MyVar est affectée au second élément

// Les deux affectations suivantes:

MyRecordVar := { 0, 1, -, 2, omettre };
MyRecordVar[6] := 6;
```

```
//se traduiront par { 0, 1, <unchanged>, 2, <undefined>, <undefined>, 6 };
// Noter aussi que le 3e élément restera indéfini s'il n'a encore jamais eu
// de valeur affectée. Et le 6e élément(d'indice 5) n'a eu aucune valeur
// affectée avant cette affectation.
```

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 séquentielle de données similaire à des séquences tabulaires 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 séquence tabulaire à deux dimensions:
var MyRecordOfType myRecordOfArray;
// et la référence à un élément particulier aura l'allure suivante:
// (valeur du second élément de la troisième structure
//'MyBasicRecordOfType')
myRecordOfArray [2][1] := 1;
```

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 doivent par conséquent n'être 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 portée hiérarchique (voir la portée hiérarchique 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.nombre+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, c'est-à-dire que le mot clé **optional** ne doit pas être utilisé dans les types **union**.

6.4 Le type "Any"

Le type spécial **anytype** est défini comme un abrégé pour la réunion de tous les *types connus* qui sont contenus dans un module TTCN-3. La définition du terme "types connus" est donnée dans le § 3.1.

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

Exemple:

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

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

MyVarThree := MyVarOne * 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 – 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 faut donc prêter une attention appropriée à de tels cas.

6.5 Séquences tabulaires

En commun avec de nombreux langages de programmation, les séquences tabulaires ne sont pas considérées d'être des types en notation TTCN-3. Elles peuvent en revanche être spécifiées au niveau d'une déclaration de variable. Les séquences tabulaires peuvent être déclarées d'être unidimensionnelles ou multidimensionnelles.

Exemple 1:

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

Les dimensions d'une séquence tabulaire doivent être spécifiées au moyen d'expressions de constantes qui doivent prendre une valeur **integer** positif. Les dimensions de séquence tabulaire 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 2:

```
var integer MyArray3[1 .. 5];    // Instancie une séquence 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 séquence
                                         // bidimensionnelle d'entiers à
                                         // 5 * 2 éléments d'indices
                                         // de (1,2) à (5,3)
```

Les valeurs des éléments de séquence tabulaire 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 séquence tabulaire (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 séquences tabulaires multidimensionnelles, chaque dimension affectée doit correspondre à un ensemble de valeurs englobées entre des accolades.

Exemple 3:

```
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}, {11, 12, 13, 14, 15}}
```

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

Exemple 4:

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

6.6 Types récursifs

Si applicable, 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 "b"est la "valeur effective" à affecter, à transmettre comme paramètre etc. Le type "B" est le "type de la valeur "b"" et le type "A" est la "définition de type de la valeur", afin d'obtenir la valeur "b" réelle.

6.7.1 Compatibilité des types non structurés

Dans les variables non structurées, les constantes, les modèles, etc., la valeur "b" est compatible avec le type "A" si le type "B" correspond au même type radical que le type "A" (c'est-à-dire au type **integer**) et ne viole pas le sous-typage (p. ex. 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 radical 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
```

6.7.2 Compatibilité des types de 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 des champs, le type des champs et l'offre d'options des champs sont identiques dans l'ordre textuel défini et si les valeurs "b" de chaque champ existant sont compatibles avec le type de leur champ correspondant dans le type "A". Les valeurs de chaque champ contenu dans la valeur "b" sont affectées au champ correspondant qui est contenu dans la valeur de type "A".

Exemple 1:

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

type record BType {
     integer            a    optional,
     integer (0..10)    b    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 parce que les
                   // options offertes par les champs ne correspondent pas
MyVarA := MyVarE;  // n'est PAS une affectation valide parce que 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 parce que champ "d"
                   // de MyVarC viole le sous-typage du champ "a" de AType
```

Dans les types **record of** et dans les séquences tabulaires, 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 séquence tabulaire 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 séquences tabulaires 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 séquence "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 (c'est-à-dire que les champs facultatifs doivent toujours être inclus dans le décompte). L'affectation des valeurs d'élément du type **record of** ou de la séquence tabulaire 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** 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 séquence tabulaire unidimensionnelle si aucune restriction de longueur du type **record of** type n'est violée ou si la dimension de la séquence tabulaire 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 les types de MyArrayVar et de 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 – 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**.

Exemple:

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

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

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

// 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 tranches

Les règles définies dans le présent paragraphe pour la compatibilité des types structurés sont également applicables à la sous-structure de tels types, c'est-à-dire à l'équivalence entre tranches.

Exemple:

```
// Compte tenu des déclarations ci-dessus,
MyVarJ.H := MyVarH;
// est une affectation valide car les types des champs H des types J et 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

Une référence de composant "b" du type de composant "B" est compatible avec un type de composant "A" si la définition du type "B" comprend la définition du type "A". Autrement dit, le type "B" comprend au moins les mêmes instances d'accès, de variable et de temporisation et les mêmes déclarations de constante que le type "A". Pour les instances d'accès, de variable et de temporisation, aussi bien le type que l'identificateur doivent être identiques.

Une référence de composant "b" de type "B" peut être affectée à une variable "a" du type de composant "A" si la référence "b" est compatible avec le type "A".

6.7.4 Compatibilité des types d'opérations de communication

Les opérations de communication (voir § 23) **`send`**, **`receive`**, **`trigger`**, **`call`**, **`getcall`**, **`reply`**, **`getreply`** et **`raise`** font exception à la règle plus 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 d'accès 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 de type **`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 radical, 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 suivi par un identificateur d'objet facultatif.

NOTE 1 – L'identificateur de module est le nom alphanumérique informel de ce module.

NOTE 2 – Les noms de module ne peuvent différer que par la partie relative à l'identificateur d'objet. Des précautions appropriées doivent cependant être prises dans ce cas, lors de l'importation, afin d'éviter tout conflit de noms car l'adjonction de préfixes aux identificateurs (voir § 7.5.8) n'est pas en mesure de résoudre une telle sorte de conflits.

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 énumération de leurs identificateurs et de leurs types entre une paire d'accolades après le mot clé **modulepar**. 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; boolean TS_Par2 };
     :
     template MyType  Mytemplate
          {
          field     TS_Par3
          };
     modulepar { hexstring TS_Par3 };
}
```

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

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 de littéral 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; sinon, la valeur effective fournie par le système de test doit être utilisée.

Exemple:

```
module MyModuleDefaultParameter
{
     modulepar { integer TS_Par0 := 0, TS_Par1; 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 linguistiques 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 linguistiques 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 à la suite de tests si nécessaire. Les groupes (imbriqués ou non) n'ont aucune détection de 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 groupe situés au même niveau hiérarchique doivent être uniques et les sous-groupes situés à un niveau hiérarchique inférieur ne doivent pas avoir le même nom qu'un groupe situé àsx un niveau hiérarchique supérieur. Si nécessaire, la notation à points doit servir à identifier sans équivoque des sous-groupes dans la hiérarchie du groupe, p. ex. pour l'importation d'un sous-groupe spécifique;

– les règles de neutralisation applicables aux attributs sont indiquées dans le § 28.4.

Exemple:

```
// Un recueil de définitions:
group MyGroup {
     const integer MyConst:= 1;
      :
     type record MyMessageType { … };
     group MyGroup1 {   // Sous-groupe avec définitions
          type record AutreMessageType { … };
          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 l'identificateur d'objet est fourni comme partie du nom de module (à partir duquel les définitions sont importées) dans l'instruction d'importation, cet identificateur d'objet doit servir à identifier le module correct.

Toutes les définitions qui sont importées à partir d'un même module ne doivent être désignées que dans une même instruction **`import`**.

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 selon 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'import 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, p. ex. un nom de fonction), une *spécification* (p. ex. 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:

```
                Name                    Specification                    Behaviour description
function    MyFunction  (inout MyType1 MyPar) return MyType2    {
                        runs on MyCompType                        const MyType3 MyConst := …;
                                                                  : // suite du comportement
                                                                }

              Specification          Name                  Specification
type    record                     MyRecordType  {
                                                   field1 MyType4,
                                                   field2 integer
                                                 }
```

```
                Specification       Name                Specification
template        MyType5             MyTemplate  {
                                                  field1 := 1,
                                                  field2 := MyConst,       // MyConst est une
                                                                           // constante de mod.
                                                  field3 := ModulePar      // ModulePar est un
                                                                           // paramètre de mod.
                                              }
```

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* (p. ex. des noms de champ de définitions de types structurés ou des valeurs de types énumérés) et des *définitions référencées* (p. ex. 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	MyType3, integer
template	MyTemplate		MyType5, field1, field2, field3, MyConst, ModulePar

NOTE 1 – La colonne des définitions locales ne se rapporte qu'aux identificateurse récemment définis dans la définition importable. Les valeurs affectées à des champs individuels de définitions importables, p. ex. 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 accès		Types de message, signatures
• type de composant	Noms de constante, noms de variable, noms de temporisateur et noms de point d'accès	Types de constante, types de variable, types de point d'accès
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 (§ **`runs on`**)
Fonction externe	Noms de paramètre	Types de paramètre, type de retour
Variante	Noms de paramètre	Types de paramètre, type de composant (§ **`runs on`**)
Test élémentaire	Noms de paramètre	Types de paramètre, types de composant (§ **`runs on`** et **`system`**)

Le mécanisme d'importation TTCN-3 établit une distinction entre *identificateur de définitions référencées* 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 (p. ex. 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) est autorisée;

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

NOTE 1 – Une définition locale, p. ex. 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; p. ex. un nom de champ d'un type Enregistrement ne peut servir qu'à accéder à un champ du type 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; p. ex., 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) par défaut, les identificateurs de définitions référencées ne sont pas automatiquement importés. Si l'on souhaite que les identificateurs des définitions référencées soient importés implicitement, l'instruction de type **recursive** (voir § 7.5.3) doit être utilisée;

NOTE 3 – Si l'on souhaite utiliser les définitions référencées dans le module d'importation quand le mécanisme d'importation par défaut est utilisé (p. ex. pour une instanciation de variable), 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.

Exemple:

```
module ModuleONE {

     modulepar {
          integer ModPar1, ModPar2 := 7
     }

     type record RecordType_T1 {
          integer   Field1_T1,
          boolean   Field2_T1
     }

     type record RecordType_T2 {
          MyRecordType_T1    Field1_T2,    // Utilisation de RecordType_T1
          MyRecordType_T1    Field2_T2,
          integer                Field3_T2
     }

     const integer MyConst := 13;

   template RecordType_T2 Template_T2 (RecordType_T1 TempPar_T2):= { //
   paramétrés modèle
          Field1_T2 := TempPar_T2,       // Référence à paramètre de modèle
          Field2_T2 := {MyConst, true}, // Référence à constante de module
          Field3_T2 := ModPar1          // Référence à 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
   // dans le contexte de Template_T2, p. ex.  lors de la fourniture d'une
   // valeur paramétrique effective. Toutes les informations nécessaires
   // pour l'usage de Template_T2, p. ex. 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, p. ex.  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 d'entier

} // 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 un 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
                                                            // inout
                                                            // 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 Importation récursive

Le mécanisme de valeurs par défaut d'importation TTCN-3 importe les définitions référencées sans leur identificateur. Autrement dit, une définition référencée ne peut pas être utilisée dans le module d'importation pour, par exemple, la déclaration d'une variable ou pour son envoi à un accès. Bien que ce mécanisme de valeurs par défaut d'importation évite le brouillage de l'espace nominatif du module d'importation, l'on souhaite parfois importer toutes les définitions référencées en même temps que leur identificateurs. En notation TTCN-3, le mot clé **`recursive`** mot clé fournit cet élément.

En utilisant le type **`import`** en même temps qu'une directive de type **`recursive`**, les règles suivantes doivent être appliquées:

a) les règles a), b), c) et f) du § 7.5.2 restent valides;

b) une définition importée par récurrence est importée en même temps que toutes les définitions référencées, c'est-à-dire que l'identificateur de toutes les définitions référencées devient visible et utilisable dans le module d'importation.

NOTE 1 – Les instructions d'importation récursive sont transitives dans les modules sources: si par exemple un module A importe par récurrence une définition à partir d'un module B qui utilise un type T également défini dans ce module B, alors le type T est automatiquement importé dans le module A.

NOTE 2 – Les instructions d'importation récursive ne sont pas transitives au-delà des limites de module: si par exemple un module A importe une définition par récurrence à partir d'un module B qui utilise un type T défini dans le module C, ce type T n'est pas automatiquement importé dans le module A. Le type T doit être importé explicitement à partir du module C, c'est-à-dire à partir de son module source.

Exemple:

```
// Les modules ModuleONE et ModuleTHREE sont définis comme dans les
// exemples pour 7.5.2.

module ModuleFIVE {

     import from ModuleONE recursive {
          template Template_T2
     }

     // L'importation récursive de Template_T2 va également importer les
     // définitions de RecordType_T2, RecordType_T1, MyConst et ModPar1 à
     // partir de ModuleONE. En raison de l'importation des types
     // RecordType_T2 et RecordType_T1, les noms de champ de ces types,
     // Field1_T1, Field2_T1, Field1_T2, Field2_T2 et Field3_T3, vont
     // devenir visibles dans ModuleFIVE

} // fin du module ModuleFIVE

module ModuleSIX {

     import from ModuleTHREE recursive {
          testcase MyTestCase
     }
```

```
        // Va provoquer une ERREUR si le module ne comporte pas une autre
        // instruction d'importation qui importe RecordType_T2 par récurrence
        // à partir de ModuleONE! L'importation récursive de MyTestCase à
        // partir de ModuleTHREE exige l'importation récursive de
        // RecordType_T2 et de MyCompType à partir de leur module source.
        // Le module source de RecordType_T2 est le module ModuleONE. Bien que
        // le module source de MyCompType soit ModuleTHREE, son importation
        // récursive va également provoquer une erreur, parce que cette
        // définition exige également des définitions à partir de ModuleONE.

} // fin du moduleSIX

module ModuleSEVEN {

     import from ModuleONE recursive {
          modulepar ModPar2;
          type RecordType_T2
     }

     // importe ModPar2, RecordType_T2 et RecordType_T1 (RecordType_T1 est
     // utilisé par RecordType_T2)à partir de ModuleONE. Au moyen de
     // l'importation récursive, les noms de champ Field1_T1,
     // Field2_T1,Field1_T2, Field2_T2 et Field3_T3 de RecordType_T2 et
     // de RecordType_T1 vont également devenir visibles.

     import from ModuleTHREE recursive {
          testcase MyTestCase
     }

     // importe MyTestCase, MyCompType (utilisé par MyTestCase)et
     // MyPortType (utilisé par MyCompType)à partir de ModuleTHREE.
     // Au moyen de l'importation récursive de MyTestCase et de MyComptype,
     // les identificateurs MyPar (définis dans MyTestCase), MyComponentVar
     // et MyPort(tous deux définis dans MyCompType)vont devenir visibles.
     // Les définitions à partir de ModuleOne qui sont requises pour
     // l'importation récursive de MyTestCase sont importées récursivement
     // par la précédente instruction d'importation.

} // fin du moduleSEVEN

module ModuleEIGHT {

     import from ModuleONE {
          modulepar ModPar2;
          type RecordType_T2
     }

     import from ModuleTHREE recursive {
          testcase MyTestCase
     }

     // Va provoquer une erreur parce que, pour l0importation récursive
     // complète de MyTestCase, le type RecordType_T1 a également besoin
     // d'être importé complètement à partir de ModuleONE ou, en d'autres
     // termes, RecordType_T2 a besoin d'être importé récursivement.

} // fin du ModuleEIGHT
```

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
          // exclut les déclarations de type MyType3 et MyType5 à partir de
         // l'instruction d'importation mais importe toutes les autres
         // déclarations de MyModule
}
```

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

Exemple 3:

```
import from MyModule all except {
     type MyType3, MyType5;  // exclut les deux types à partir de
                             // l'instruction d'importation
     template all            // exclut tous modèles déclarés dans MyModule
                             // à partir 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
                    // exclut la définition des types MyType3 et MyType5
                    // à partir de l'instruction d'importation mais importe
                    // toutes les autres définitions de MyGroup
     }
}
```

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

Exemple 3:

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

Exemple 1:

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

Si certaines déclarations d'une sorte sont choisies pour exclusion à partir 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 Manipulation des conflits de noms lors d'une importation

Tous les modules TTCN-3 doivent avoir leur propres espace nominatif dans lequel toutes les définitions doivent être identifiées de façon univoque. Des conflits de noms peuvent apparaître en raison d'importations, p. ex. Une importation à partir de différents modules, une importation de groupes ou une importation de définitions récursives. 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 chaîne"; // un préfixe
                                                            // doit
                                                            // impérativement
                                                            // être utilisé
          var MyTypeA MyVar2 := '10110011'B; // c'est le type original
                                             // MyTypeA
           :
          var MyTypeB MyVar3 := "Test chaîne"; // un préfixe n'a pas besoin
                                               // d'être utilisé …
         var SomeModuleC.MyTypeB MyVar3 := "Test chaîne";  // … mais peut
                                                           // l'être
                                                           // au besoin
           :
     }
}
```

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 Manipulation 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, p. ex. La surécriture 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 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. L'instruction **`except`** exclut les définitions à partir de cette seule 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

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 (p. ex. module, ensemble, 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. Les identificateurs linguistiques spécifiés pour les modules ASN.1 sont indiqués au § D.1.

Exemple:

```
import from MyASN1Module language "ASN.1:2002" {
     type MyASN1Type
}
```

NOTE – Le mécanisme d'importation est conçu de façon à permettre de réutiliser des définitions en notation TTCN-3 ou ASN.1 à partir d'autres modules TTCN-3 ou ASN.1. Les règles applicables à l'importation de définitions à partir de spécifications écrites dans d'autres langages, p. ex. les ensembles en langage SDL, peuvent suivre les règles TTCN-3 ou être définies séparément. L'importation de règles applicables à des langages autres que TTCN-3 et ASN.1 n'est pas couverte par la présente Recommandation.

8 Configurations de test

8.0 Généralités

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

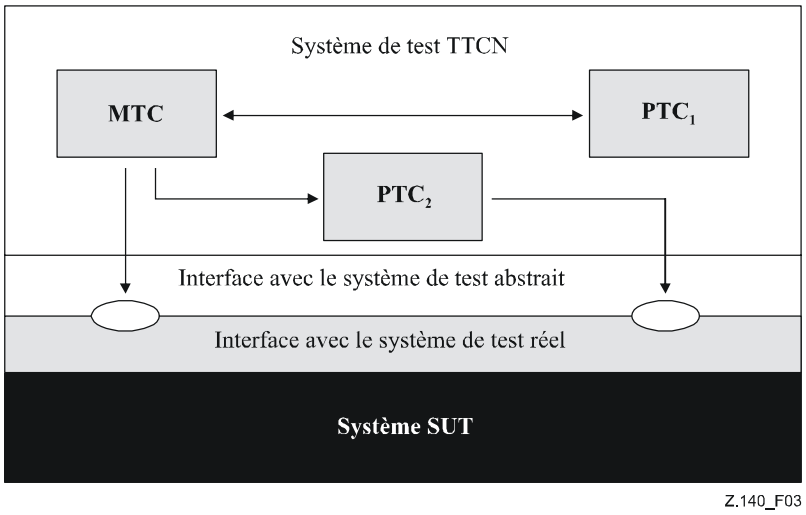

Figure 3/Z.140 – Vue théorique d'une configuration de test TTCN-3 typique

A l'intérieur de chaque configuration, il doit y avoir un (et un seul) composant de test principal (MTC). Les composants de test qui ne sont pas de type MTC sont appelés *composants de test parallèle* 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 avec le système de test est réalisée au moyen de points d'accès de communication (voir § 8.1).

Les types de composant de test et les types de point d'accès, 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 opération **`create`** et **`connect`** opérations dans le cadre du comportement de test élémentaire. Les accès de composant sont connectés aux accès de l'interface avec le système de test au moyen de l'opération **`map`** (voir § 22.2).

8.1 Modèle de communication entre points d'accès

Les composants de test sont connectés au moyen de leurs points d'accès; c'est-à-dire que les connexions entre les composants, et entre ceux-ci et l'interface avec le système de test, sont orientées vers des points d'accès. Chaque point d'accès 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 cet accès.

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

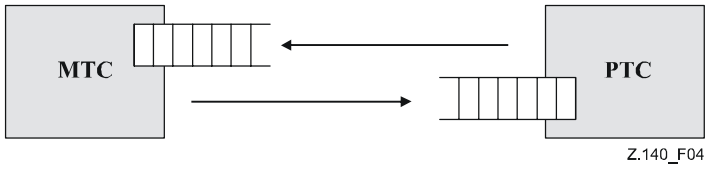

Figure 4/Z.140 – Le modèle de point d'accès de communication TTCN-3

8.2 Restrictions relatives aux connexions

En notation TTCN-3, les connexions s'effectuent d'accès à accès et d'accès à interface avec le système de test (voir Figure 1). Il n'n'y a aucune restriction relatives au nombre de connexions qu'un composant peut entretenir. Des connexions point à multipoint sont également autorisées (p. ex. Figure 5 g) ou Figure 5 h)).

Les connexions suivantes ne sont pas autorisées:

- un point d'accès détenu par un composant A ne doit pas être connecté avec deux ou plus de deux accès détenus par le même composant (Figure 6 a) et Figure 6 e));
- un point d'accès détenu par un composant A ne doit pas être connecté avec deux ou plus de deux accès détenus par un composant B (voir Figure 6 c));

- un point d'accès 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 avec le système de test ne sont pas autorisées (voir Figure 6 f)).

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.

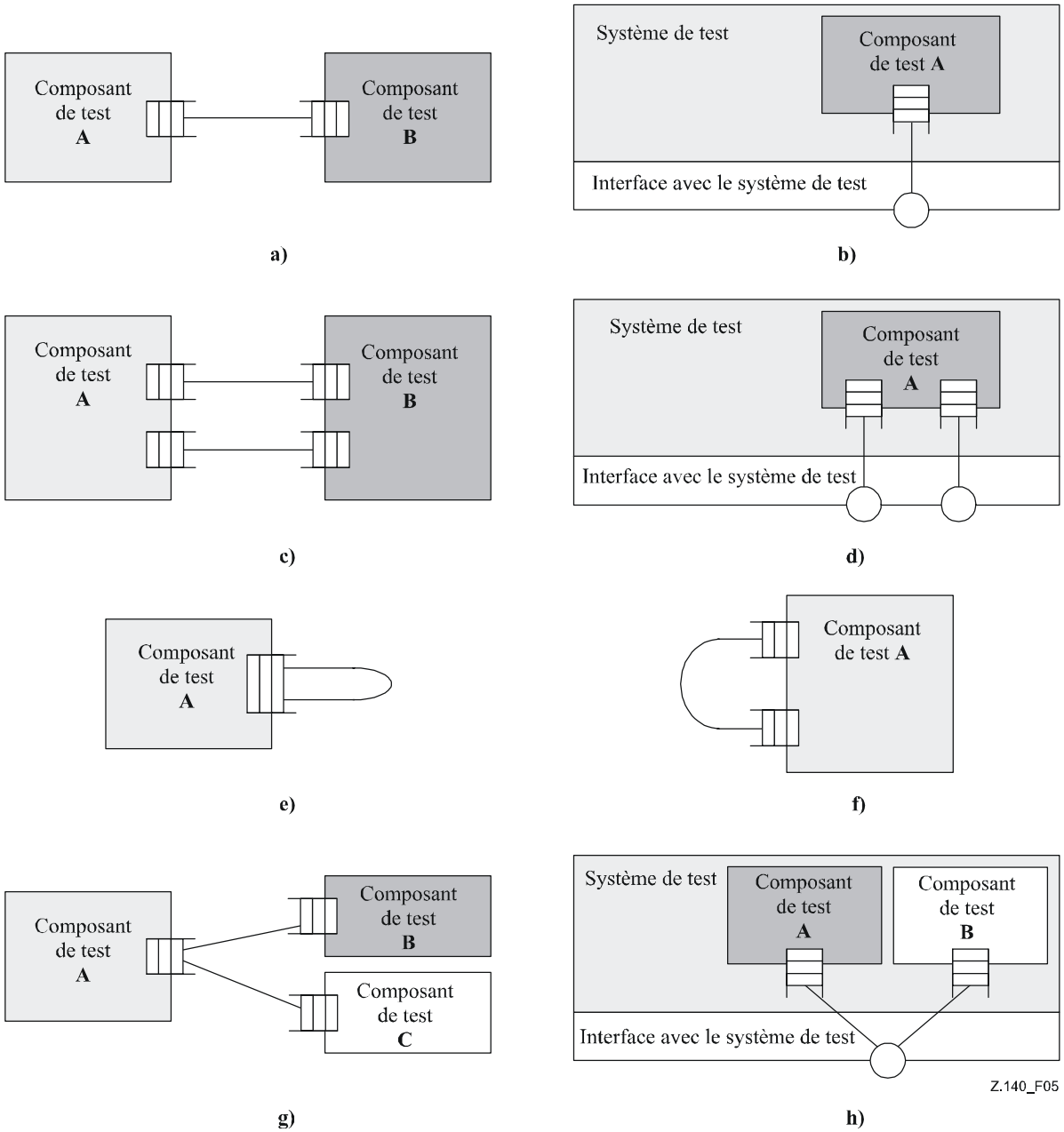

Figure 5/Z.140 – Connexions autorisées

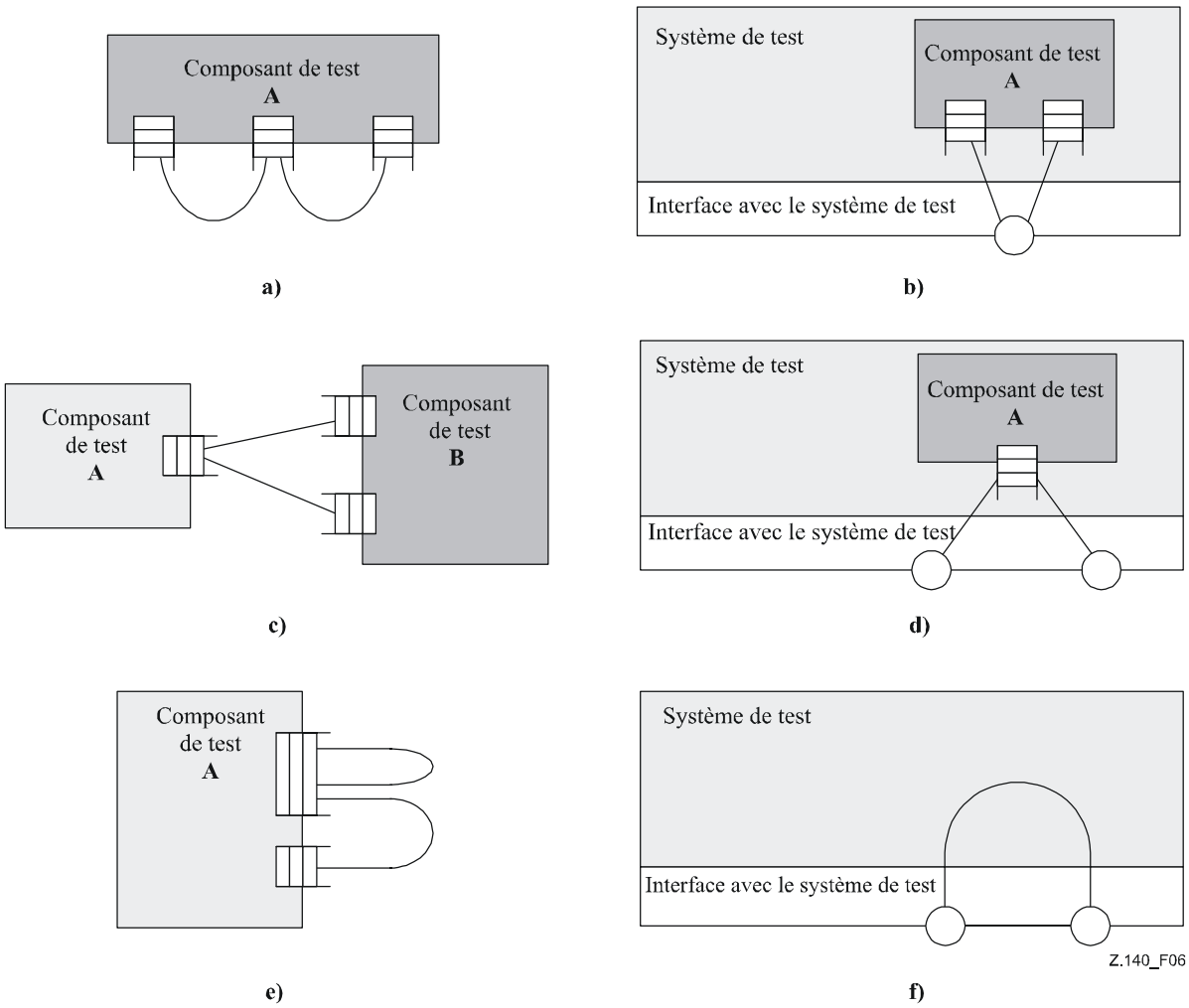

Figure 6/Z.140 – Connexions NON autorisées

8.3 Interface avec un système de test abstrait

La notation TTCN-3 sert à tester des implémentations. L'objet soumis au test est appelé *instance sous test* ou *instance IUT*. L'instance 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) avec le système de test doit être par contre associée à chaque test élémentaire. Une définition de l'interface avec le système de test est identique à une définition de composant; c'est-à-dire qu'il s'agit d'une liste de tous les points d'accès de communication possibles au moyen desquels le test élémentaire est connecté au système SUT.

L'interface avec le système de test définit statiquement le nombre et le type des connexions par point d'accès au système SUT pendant un processus d'exécution de test. Cependant, les connexions entre l'interface avec le 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 d'accès de communication

8.4.0 Généralités

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

La notation TTCN-3 prend en charge les points d'accès en mode message ou en mode procédure. Chaque point d'accès doit être défini comme étant en mode message ou en mode procédure (ou les deux au même instant comme décrit dans le § 8.4.1). Les accès en mode message doivent être identifiés par le mot clé **message** et les accès en mode procédure doivent être identifiés par le mot clé **procédure** dans la définition associée du type d'accès.

Les points d'accès sont directionnels. 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 d'accès 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.

Exemple 1:

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

// Point d'accès en mode procédure qui permet l'appel distant des
// procédures Proc1, Proc2 et Proc3. Noter que les procédures Proc1, Proc2
// et Proc3 sont définies en tant que signatures
type point d'accès 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 nom de type.

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

Exemple 2:

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

8.4.1 Points d'accès mixtes

Il est possible de définir un point d'accès comme permettant deux sortes de communication, ce qui est indiqué par le mot clé **mixed**. Autrement dit, les listes relatives aux accès mixtes seront également être mixtes et comprendront à la fois des signatures et des types. Le mot clé **all** indique dans ce cas tous les types et toutes les signatures de procédure définis dans le module. Aucune séparation n'est opérée dans la définition.

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

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

type port MyAllMixedPortType mixed
{
     inout     all
}
```

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

Les opérations servant à commander les accès (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 d'accès en mode mixte.

8.5 Définition des types de composant

8.5.0 Généralités

Le type **`component`** définit les accès qui sont associés à un composant. Ces définitions doivent être effectuées dans la partie du module relative aux définitions. Les noms de point d'accès contenus dans une définition de composant sont locaux par rapport à ce composant c'est-à-dire qu'un autre composant peut avoir des accès portant les mêmes noms. Les accès du même composant doivent 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 points d'accès.

NOTE – La notation TTCN-3 diffère de la notation TTCN-2 à cet égard, car la configuration de test est statique et la déclaration des composants de test, des accès PCO et des primitives ASP implique leur connexion automatique quand l'exécution d'un test élémentaire est lancée.

Exemple:

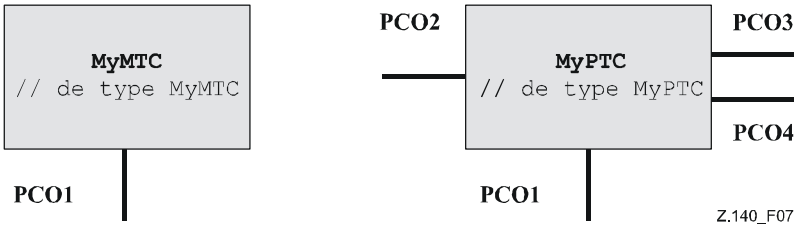

Figure 7/Z.140 – Composants typiques

```
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 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 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 avec le système de tests (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 séquences tabulaires de points d'accès

Il est possible de définir des séquences tabulaires de points d'accès dans les définitions de type de composant (voir également § 22.9).

Exemple:

```
type component My3pcoCompType
{
     port MyMessageInterfaceType PCO[3]
     // Définit un type de composant qui a une séquence de 3 points
     // d'accès.
}
```

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 d'accès 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 accès appliqués 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 transmise 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, p. ex. pour l'initialisation de variables du type adresse.

Exemple:

```
// Associe le type integer à l'adresse du type ouvert
type integer adress;
 :
// nouvelle variable d'adresse initialisée par la valeur null
var adress MySUTentity := null;
 :
// réception d'une valeur d'adresse et affectation de celle-ci à la
// variable MySUTentity
PCO.receive(adress:*) -> 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 accès connectés à des composants de test autres que l'interface avec le 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, p. ex. pour l'initialisation de variables manipulant 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 la manipulation 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 de manipulation 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 de deux variables pour la manipulation de références à des
// composants de type MyCompType et pour la création d'un composant de ce
// type
var MyCompType MyCompInst := MyCompType.create;

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

// Utilisation 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
// accès 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 avec le système de test

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

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

Généralement, une référence de type de composant définissant l'interface avec le système de test doit être associée à chaque test élémentaire utilisant plus d'un seul composant de test. Les accès de l'interface avec le système de test doivent automatiquement être instanciés par le système en même temps que le composant MTC quand l'exécution du test élémentaire commence, c'est-à-dire quand le test élémentaire est appelé à partir de la partie commande du module.

L'opération retournant la référence de composant de l'interface avec le système de test est de type **system**. Elle doit servir à s'adresser aux accès 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 avec le 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 avec le 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**. La valeur de la constante doit être affectée au point de déclaration.

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;    // déclaration de constante
                                           // externe
```

Une constante externe peut être d'un type quelconque mais celui-ci doit être connu dans le module, c'est-à-dire qu'il doit être un type radical ou un type défini par l'utilisateur dans le module, ou un type importé à partir d'un autre module. L'affectation du type à la représentation externe d'une constante externe est également hors du domaine d'application de la présente Recommandation. Le mécanisme de transmission de la valeur d'une constante externe vers un module est hors du domaine d'application de la présente Recommandation.

10 Déclaration de variables

Les variables sont indiquées par le mot clé **var**. Elles 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). Une déclaration de variable peut avoir une valeur initiale facultative qui lui a été affectée.

Exemple:

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

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

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 du temporisateur peut avoir une facultatif valeur facultative de durée par défaut qui lui est affectée. Le temporisateur doit être armé 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:

```
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 armé
```

En plus des instances de temporisation isolées, des séquences tabulaires de temporisations peuvent également être déclarées. La ou les durées par défaut des éléments d'une séquence de temporisations doivent être affectées au moyen d'une séquence de valeurs. Le premier élément de la séquence de valeurs est affecté au premier élément de la séquence de temporisations, la seconde valeur au second élément, etc. Si l'on souhaite omettre l'affectation de durée par défaut pour certains éléments de la séquence de temporisations, cela doit être déclaré explicitement au moyen du symbole de non-utilisation ("-").

NOTE – Autrement dit, le nombre des éléments de la séquence de valeurs doit impérativement être le même que le nombre d'éléments contenus dans la séquence de temporisations.

Par exemple:

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

timer t_Mytimer2[5] := { 1.0, 2.0, 3.0, 4.0, - }
     // le dernier élément (t_Mytimer2[4]) est laissé 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 comportementale 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 armé 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 manipulés 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
fonction MyBehaviour (temporisation 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 points d'accès de communication définis dans la configuration de test. Ces messages peuvent être ceux qui sont explicitement orientés vers les tests du système SUT ou ceux qui sont explicitement orientés vers 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), les unités de données de protocole (PDU) 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 mot clé **`noblock`** sont censées servir aux communications bloquantes en mode procédure.

Exemple:

```
signature MyRemoteProcOne ();     // MyRemoteProcOne sera utilisé pour les
                                  // communications bloquantes sur la base
                                  // de procédures. Il n'a ni paramètres ni
                                  // valeur de retour

signature MyRemoteProcTwo () noblock; // La procédure MyRemoteProcTwo sera
                                      // utilisée pour une communication
                                      // non bloquante sur la base de
                                      // procédures. 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
// 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 propagées par des procédures distantes sont représentés 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 que par leur représentation 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 peut retourner 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
// bloquante en mode procédure. Dans le cas d'un échec de terminaison, la
// procédure MyRemoteProcSix peut déclencher 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 reçues correspond au modèle spécifié.

Les modèles offrent les possibilités suivantes:

a) ils permettent d'organiser et de réutiliser des données de test, y compris un 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. A 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. La déclaration de modèles pour des messages ne doit pas utiliser le symbole de non-utilisation "-". Celui-ci peut être utilisé dans les modèles de signature pour des paramètres qui ne sont pas applicables et dans toutes les déclarations de modèle modifié afin d'indiquer l'absence de modification dans le champ ou élément spécifié.

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 spéciaux de configuration et de défaut (**`port, component, address`** 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 accès de test. Au moment de l'opération de type **send**, le modèle doit être pleinement 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 := "Ma chaîne",
     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** 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.

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 **opération de type "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 négligée; il est donc permis de spécifier des mécanismes d'appariement pour ces champs, ou de les omettre (voir Annexe B).

Exemple:

```
// Dans les exemples de l'introduction du § 14.2 …

// 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 bilatéral Par3 a un
// attribut d'appariement et non une valeur
MyPCO.call(RemoteProc:Template3);

// Les modèles ne renvoient 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 négligés dans le processus d'appariement.

Exemple:

```
// Dans les exemples de l'introduction du § 14.2 …

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

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

// Opération traitement d'appel 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

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 utilisé 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**, **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, objid, bitstring, hexstring, octetstring.

Le type du modèle en ligne doit cependant être dans la liste des points d'accès par lesquels le modèle est reçu. En cas d'ambiguïté entre le type énuméré et le type de la valeur fournie (p. ex. 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 qui s'évalue comme 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);

– **(**de bas en **(to)** haut**)**: une étendue de valeurs d'entier entre et y compris les limites inférieure et supérieure.

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

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

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

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

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

– **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 équivalents en notation TTCN-3 et ASN.1 définis dans la série de Recommandations UIT-T X.680 [7], [8], [9] et [10] auxquels ces mécanismes d'appariement s'appliquent. Une description complète de chaque mécanisme d'appariement peut être trouvée dans l'Annexe B.

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

Mécanisme utilisé avec valeurs de	Valeur		Au lieu des valeurs							l'intérieur valeurs		Attributs	
	SpecificValue	OmitValue	ComplementedList	ValueList	AnyValue(?)	AnyValueOrNone(*)	Range	Superset	Subset	AnyElement(?)	AnyElementsOrNone(*)	LengthRestriction	IfPresent
boolean	Oui	Oui	Oui	Oui	Oui	Oui							Oui
integer	Oui	Oui	Oui	Oui	Oui	Oui	Oui						Oui
char	Oui	Oui	Oui	Oui	Oui	Oui	Oui						Oui
universal char	Oui	Oui	Oui	Oui	Oui	Oui	Oui						Oui
float	Oui	Oui	Oui	Oui	Oui	Oui	Oui						Oui
bitstring	Oui	Oui	Oui	Oui	Oui	Oui				Oui	Oui	Oui	Oui
octetstring	Oui	Oui	Oui	Oui	Oui	Oui				Oui	Oui	Oui	Oui
hexstring	Oui	Oui	Oui	Oui	Oui	Oui				Oui	Oui	Oui	Oui
character strings	Oui	Oui	Oui	Oui	Oui	Oui	Oui			Oui	Oui	Oui	Oui
record	Oui	Oui	Oui	Oui	Oui	Oui							Oui
record of	Oui	Oui	Oui	Oui	Oui	Oui				Oui	Oui	Oui	Oui
array	Oui	Oui	Oui	Oui	Oui	Oui				Oui	Oui	Oui	Oui
set	Oui	Oui	Oui	Oui	Oui	Oui							Oui
set of	Oui	Oui	Oui	Oui	Oui	Oui		Oui	Oui	Oui	Oui	Oui	Oui
enumerated	Oui	Oui	Oui	Oui	Oui	Oui							Oui
union	Oui	Oui	Oui	Oui	Oui	Oui							Oui

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));
```

14.4.1 Paramétrage avec attributs d'appariement

Afin que l'on puisse transmettre des modèles ou les symboles correspondants sous forme de paramètres, le mot clé supplémentaire **template** doit être ajouté avant le champ de type. Cela transforme le paramètre en type de modèle et élargit en fait les paramètres autorisés pour le type associé afin qu'ils 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,
     field2 := pattern "abc*xyz",
     field3 := true
}

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

14.5 Transmission de modèles comme paramètres

Seules les définitions des types **function**, **testcase**, **altstep** et **template** peuvent avoir des modèles comme paramètres formels.

Exemple:

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

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 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 supérieur dont le modèle nouveau ou modifié doit être issu. Ce modèle supérieur peut être soit le 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 ou le symbole d'appariement spécifié remplace celui qui a été spécifié dans le modèle supérieur. 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 supérieur 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:

```
// si l'on a
template MyRecordType MyTemplate1 :=
{
     field1 := 123,
     field2 := "Une chaîne",
     field3 := true
}
// alors le fait d'écrire
template MyRecordType MyTemplate2 modifies MyTemplate1 :=
{
     field1 := omit,         // le champ field1 est facultatif mais présent
                             // dans MyTemplate1
     field2 := "Une chaîne modifiée"
                             // le champ field3 est inchangé
}
// revient à écrire
template MyRecordType MyTemplate2 :=
{
     field1 := omit,
     field2 := "Une chaîne modifiée",
     field3 := true
}
```

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;

d) les champs de modèle de base contenant des modèles paramétrés ne doivent pas être modifiés ni être explicitement omis dans un modèle modifié.

Exemple:

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

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

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 (p. ex. **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'opé**ration match** permet de comparer la valeur d'une variable 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 aux champs d'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	**<=**
Opérateurs logiques	non logique	**not**
	et logique	**and**
	ou logique	**or**
	ou exclusif logique	**xor**

Tableau 7/Z.140 – Liste des opérateurs TTCN-3

Catégorie	Opérateur	Symbole ou mot clé
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, l'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`
	Binaire	`or`
La moins élevée		

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** qui ne doit être utilisé 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|          si   x >= 0
                  = 0                        si     x < 0 et    x rem |y| = 0
                  = |y| + x rem |y|          si     x < 0 et    x rem |y| < 0
```

Le Tableau 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 gives '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 des opérandes que 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 (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 de ces champs choisis sont compatibles et si les valeurs effectives de ces 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 SetA    conSetA1  :=   { a1 := 0, a3 := 2 };
              // Noter que l'ordre de définition des valeurs du champ n'a
              // pas d'importance
const SetB    conSetB1  :=   { b1 := 0, b3 := 2 };
const SetB    conSetB2  :=   { b2 := 0, b3 := 2 };
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 (ce n'est pas l'élément
         // correspondant qui est 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**.

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, 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 0. Autrement dit:

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

Exemple 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.

NOTE – Un verdict d'erreur doit être rendu si un débordement dû au système se produit lors de l'application – à l'opérande de gauche – de l'opération de décalage à 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, ce qui est indiqué par le mot clé **return** suivi par un identificateur de type. Le mot clé **return**, quand il est utilisé dans le corps de la fonction avec un type de retour défini dans son en-tête, doit toujours être suivi par une valeur, par une constante ou par une référence de variable ou par une expression présentant la valeur de retour. Le type de la valeur de retour doit être compatible avec le type 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
}
```

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 d'accès.

```
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
```

NOTE 1 – Les fonctions TTCN-3 remplacent les définitions procédurales des "modules de test" et des "suites de test"; les fonctions externes remplacent les opérations de suite de test en notation TTCN-2. Des fonctions informelles peuvent être déclarées en tant que fonctions externes avec des commentaires explicatifs ou au moyen d'une fonction formelle vide, assortie de commentaires.

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 accès 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 utilisées dans la fonction au sujet du composant sont transmises 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;           // d'accès d'envoi. Cette fonction
     PCO1.send(MyVar);            // exige donc une clause "runs on"
                                  // afin de résoudre les identificateurs
                                  // de point d'accès 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 2 – 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** paragraphe. 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 identiques

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 **`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) fonctions (voir Tableau 10) 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 **char**	**`int2char`**
	Convertit une valeur **integer** en valeur **universal char**	**`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 **char** en valeur **integer**	**`char2int`**
	Convertit une valeur **universal char** 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** à **bitstring**	**`hex2bit`**
	Convertit une valeur **hexstring** à **octetstring**	**`hex2oct`**
	Convertit une valeur **hexstring** à **charstring**	**`hex2str`**
	Convertit une valeur **octetstring** en valeur **integer**	**`oct2int`**

Tableau 10/Z.140 – Liste des fonctions prédéfinies en notation TTCN-3

Catégorie	Fonction	Mot clé
	Convertit une valeur **octetstring** en valeur **bitstring**	`oct2bit`
	Convertit **octetstring** en valeur **hexstring**	`oct2hex`
	Convertit une valeur **octetstring** en valeur **charstring**	`oct2str`
	Convertit une valeur **charstring** en valeur **integer**	`str2int`
	Convertit une valeur **charstring** en valeur **octetstring**	`str2oct`
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`
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 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`
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.

16.2 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 paramètre de valeurs, c'est-à-dire des paramètres de type **in**. 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 effectif et l'état du composant, les opérations suivantes ne doivent pas être appelées pendant l'initialisation d'une définition locale:

– l'opération de fin d'exécution **done**;

– toutes les opérations d'accès, c'est-à-dire: début (d'accès), arrêt (d'accès), libération, émission, réception, déclenchement, appel, traitement d'appel, réponse, traitement de réponse, propagation, acquisition, vérification;

NOTE 1 – L'exécution des opérations **done**, **start** (début d'accès), **stop** (arrêt d'accès), **clear**, **receive**, **trigger**, **getcall**, **getreply**, **catch** et **check** peut provoquer des conflits avec l'instantané effectif. Elle peut supprimer des informations de files d'attente à un accès, limiter l'accession à des files d'attente d'accès et/ou provoquer un nouvel instantané à l'intérieur de l'évaluation de l'instantané effectif.

NOTE 2 – Les opérations **send**, **call**, réponse et **raise** doivent être évitées aux fins de la lisibilité, c'est-à-dire que toutes les communications doivent être rendues explicites et non présentées comme un effet secondaire pendant la communication;

– les opérations **start** (début de temporisation), **timeout** et **stop** (arrêt de temporisation);

NOTE 3 – Il est permis d'utiliser les opérations de temporisation **readtimer** et **running**.

NOTE 4 – Ces restrictions à l'initialisation de définitions locales dans des variantes sont les mêmes que celles qui visent à éviter des effets secondaires pour les expressions booléennes utilisées afin de sélectionner et désélectionner des options dans l'instruction **alt** ou dans des variantes.

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 {
          …
        }
     [] AutreAltStep(); // appel explicite de la variante AutreAltStep
                        // présenté comme 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 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 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**.

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

Une fonction ou variante qui se rapporte au type de composant "B" dans sa clause **runs on** peut être lancée lors d'une instance du type de composant "A" si le type "B" est compatible avec le type "A" conformément au § 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 accès du composant MTC et l'interface avec le système de test sont instanciés et le comportement spécifié dans les définition 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 points d'accès 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 accès requis par l'interface avec le 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 accès de l'interface avec le système de test.

Exemple:

```
testcase MyTestCaseOne()
runs on MyMtcType1          // Définit le type du composant MTC
system MyTestSystemType     // Rend les noms d'accès 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 instructions de programmation de base telles que des expressions, des affectations, des structures itératives et analogues, des instructions comportementales telles que le comportement séquentiel, le comportement à options, l'entrelacement, les valeurs par défauts 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'une bloc vide d'instructions et de déclarations, c'est-à-dire {}, peut être trouvée dans des instructions composites, p. ex. Une branche dans une instruction **`alt`**. Elle implique qu'aucune action n'est entreprise.

Tableau 11/Z.140 – Aperçu général des 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
Instructions de programmation de base			
Expressions	`(…)`	Oui	Oui
Affectations	`:=`	Oui	Oui
Journalisation	`log`	Oui	Oui
Étiqueter et Saut	`label / goto`	Oui	Oui
Condition "si-sinon"	`if (…) {…} else {…}`	Oui	Oui
Boucle "pour"	`for (…) {…}`	Oui	Oui
Boucle "tant que"	`while (…) {…}`	Oui	Oui
Boucle "exécuter tant que"	`do {…} while (…)`	Oui	Oui
Arrêt d'exécution	`stop`	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
Comportement entrelacé	`interleave {…}`	Oui (voir Note 1)	Oui
Commande de retour	`return`		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 composant à un autre composant	`connect`		Oui
Déconnexion de deux composants	`disconnect`		Oui
Affectation d'un accès à une interface avec le système de test	`map`		Oui
Désaffectation d'un accès de l'interface avec le système de test	`unmap`		Oui
Obtention d'adresse de composant MTC	`mtc`		Oui
Obtention d'adresse d'interface avec le système de test	`system`		Oui

Tableau 11/Z.140 – Aperçu général des 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
Obtention d'adresses propres	`self`		Oui
Début d'exécution de composant de test	`start`		Oui
Arrêt d'exécution de composant de test	`stop`		Oui
Vérification de terminaison d'un PTC	`running`		Oui
Attente de terminaison d'un PTC	`done`		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 issu d'une entité distante	`getcall`		Oui
Traitement de la réponse issu d'un appel précédent	`getreply`		Oui
Acquisition d'une exception (issue d'une entité appelée)	`catch`		Oui
Vérification du message/appel reçu (actuellement)	`check`		Oui
Libération d'accès	`clear`		Oui
Libération et ouverture d'un accès	`start`		Oui
Fermeture d'un accès (réception et émission)	`stop`		Oui
Opérations de temporisation			
Armement de temporisateur	`start`	Oui	Oui
Arrêt de temporisateur	`stop`	Oui	Oui
Lecture de la durée écoulée	`read`	Oui	Oui
Vérification d'armement de temporisateur	`running`	Oui	Oui
Evénement d'expiration de temporisateur	`timeout`	Oui	Oui
Opérations de verdict			
Mise à jour de verdict local	`setverdict`		Oui
Requête de verdict local	`getverdict`		Oui

Tableau 11/Z.140 – Aperçu général des 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
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.			

19 Instructions de programmation de base

19.0 Généralités

Les instructions de programmation de base (voir Tableau 12) sont des expressions, des affectations, des opérations, des structures itératives etc. Toutes 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 12/Z.140 – Aperçu général des instructions de programmation de base TTCN-3

Instructions de programmation de base	
Instruction	**Mot clé ou symbole associé**
Expressions	`(…)`
Affectations	`:=`
Journalisation	`log`
Etiqueter et "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`

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) ou (not C) or (j<10));
```

19.2 Affectations

Des valeurs peut 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 la valeur d'un élément du même type que 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 une chaîne de caractères dans un dispositif de journalisation associé à une commande de test ou au composant de test dans lequel l'instruction est utilisée.

Exemple:

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

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

19.4 L'instruction "Label"

L'instruction **label** permet de spécifier des étiquettes dans des tests élémentaires, dans des fonctions, dans des variantes 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 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);
            goto L1;                        // Sauter à l'étiquette L1
        }
}
 :
```

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. 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 (p. ex. à partir d'une boucle de type **while**) et de sauter plusieurs niveaux à partir d'instructions composites imbriquées (p. ex. 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.

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 $\texttt{statementBlock}_x$ se rapporte à un bloc d'instructions.

Exemple:

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

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

Une proposition plus complexe pourrait être:

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

Dans de tels cas, la lisibilité dépend fortement du formatage mais celui-ci ne doit 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 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 du test. 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 {
      :   // Partie définitions du module
     testcase MyTestCase() runs on MyMTCType system MySystemType{
           :
          stop      // arrête un composant de test
     }
     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).

20 Instructions de programmation comportementales

20.0 Généralités

Les instructions de programmation comportementales peuvent être utilisées dans les tests élémentaires, dans les fonctions et dans 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 (voir Tableau 13) spécifient le comportement dynamique des composants de test aux points d'accès 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:

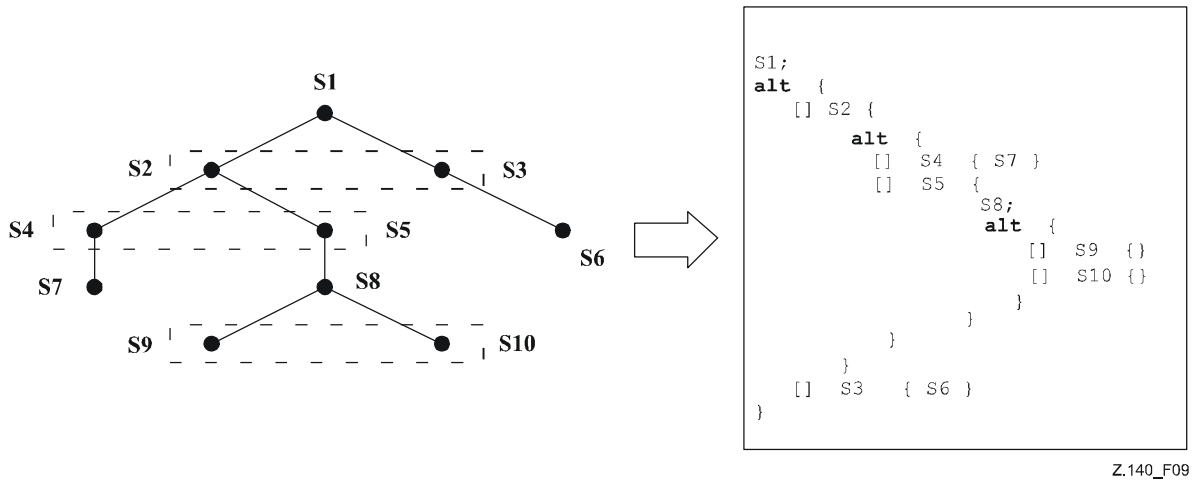

Figure 9/Z.140 – Illustration du comportement à options

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** et **done**. L'instruction **alt** indique un ensemble d'événements possibles qui doivent être appariés en fonction d'un instantané particulier (voir § 20.1.1).

NOTE – L'instruction **alt** correspond aux options situées au même niveau de décalage en notation TTCN-2. Cependant, il y a des différences notables, p. ex.:

a) il n'est pas possible d'examiner la file d'attente à un accès au moyen de l'expression **boolean** puis de désactiver une option;

b) il n'est pas possible d'appeler une fonction en tant qu'option dans l'instruction **alt**, sauf si une sauvegarde conditionnelle (c'est-à-dire un opérateur [**else**]) est le dernier choix dans l'instruction **alt** (voir § 20.1.3).

Exemple:

```
// Utilisation d'instructions "alt" imbriquées
 :
alt {
  [] L1.receive(DL_REL_CO:*) {
               setverdict(pass);
               TAC.stop;
               TNOAC.start;
               alt {
                      [] L1.receive(DL_EST_IN) {
                                 TNOAC.stop;
                                 setverdict(pass);
                                 }
                      [] TNOAC.timeout {
                                 L1.send(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 régissant les branches facultatives, tous les composants de test arrêtés qui sont applicables, tous les événements applicables d'expiration de temporisateur ainsi que les messages, appels, réponses et exceptions de niveau supérieur contenus dans les files applicables d'attente d'entrée dans un accès. Tout composant de test, toute temporisation et tout point d'accès 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 (ETSI ES 201 873-4 (voir Annexe F)).

NOTE 1 – Les instantanés d'analyse ne sont que moyens théoriques permettant de décrire le comportement de l'instruction **alt**. Les algorithmes concrets permettant la manipulation de l'instantané peuvent être trouvés dans la norme ETSI ES 201 873-4 (voir Annexe F).

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 d'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 de fin de temporiation de type **timeout** (*branche d'expiration de temporisateur*) ou par une opération de réception (*branche de réception*), c'est-à-dire de type

receive, **trigger**, **getcall**, **getreply** ou **catch**. L'évaluation des sentinelles booléennes doit être fondée sur l'instantané. La sentinelle booléenne est considérée comme étant *satisfaite* si aucune sentinelle booléenne est défini, 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 n'a par elle-même aucun autre effet.

Une *branche d'expiration de temporisateur* est choisie si la sentinelle booléenne est satisfaite et si l'événement spécifié d'expiration de temporisateur est contenu dans la liste des temporisateurs expirés 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 temporisateur à partir de la liste des temporisateurs expirés 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 accès, 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 soit 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 accès 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 accès applicables contiennent au moins un message, un appel, une réponse ou une exception.

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 booléennes (ou
// avec une sentinelle)
 :
alt {
  [x>1] L2.receive {                    // Expression/sentinelle
                                        // booléenne
        setverdict(pass);
    }
  [x<=1] L2.receive {                   // Expression/sentinelle
                                        // booléenne
        setverdict(inconc);
    }
}
 :
```

20.1.3 Branche d'échappement dans une instruction "alt"

La dernière 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, d'expiration de temporisateur 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) booléennes
 :
alt {
  [x>1] L2.receive {
        setverdict(pass);
    }
  [x<=1] L2.receive {
        setverdict(inconc);
    }
  [else] {                          // branche d'échappement
        MyErrorManipulation();
        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 répéter comme dernière instruction d'une branche de type **else**.

20.1.4 Vide

20.1.5 Réévaluation d'instruction "alt"

La réévaluation d'une instruction **alt** instruction peut être spécifiée au moyen d'une instruction de type répéter (voir § 20.2).

Exemple:

```
alt {
  [] PCO3.receive {
                  count := count + 1;
                  repeat                  // utilisation de la répétition
     }
  [] T1.timeout { }
  [] tout 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 { }
  [] AutreAltStep();    // appel explicite de la variante AnotherAltStep
                        // comme option d'une instruction "alt"
  [] MyTimer.timeout { }
}
 :
```

20.2 L'instruction "Repeat" (répétition)

L'instruction **repeat** (répétition) provoque la réévaluation d'une 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. L'instruction **repeat** (répétition) ne doit être utilisée que comme dernière instruction d'une option contenue dans une instruction **alt** ou que comme dernière instruction d'une option sommitale contenue dans une définition de variante.

Si une instruction de type répéter instruction est utilisée comme dernière instruction d'option dans une instruction **alt** instruction, elle provoque un nouvel instantané et la réévaluation de l'instruction **alt**.

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 répéter est utilisée comme dernière instruction d'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:

```
// Usage of repeat in an altstep
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**, **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 norme ETSI ES 201 873-4 (voir Annexe F).

La règle d'évaluation d'une instruction d'entrelacement est ce qui suite:

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** 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 pleinement définie dans la norme ETSI ES 201 873-4 (voir Annexe F).

Exemple:

```
// Le fragment de code TTCN-3 suivant
 :
interleave {
[]   PCO1.receive(MySig1)
          { PCO1.send(MySig2);
```

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

// peut être interprété comme 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 et fait revenir la commande au point à partir duquel la fonction a été appelée. Une instruction **return** peut (à titre d'option) être associée à une valeur de retour. Une instruction **return** ne doit être utilisée que dans une fonction.

Exemple:

```
function MyFunction() return boolean {
     :
     if (date == "1.1.2000") {
          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. La valeurs par défaut sont énumérées dans l'ordre de leur activation. Les opérations TTCN-3 opérations **activate** (voir § 21.3) et **deactivate** (voir § 21.4) fonctionnent d'après la liste de valeurs par défaut. Une opération **activate** appose une nouvelle valeur par défaut à la fin 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. (Voir Tableau 14.)

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 ne pourrait être exécutée. Un mécanisme évoqué par défaut invoque la première variante contenue dans la liste des valeurs par défaut 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 seul des options sommitales 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 répéter (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 exigencela est que les valeurs par défaut soient appelées dans l'ordre 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 la manipulation de 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 apposer la variante désignée à 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:

```
 :
// Déclaration d'une variable pour la manipulation 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
 :
```

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, p. ex. au moment de son invocation par le mécanisme de valeurs par défaut).

Exemple:

```
 :
var default MyDefaultVar := null;
 :
MyDefaultVar := activate(MyAltStep2(5, MyVar);
     // MyAltStep2 est activée par défaut avec le paramètre effectif 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 désignée 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** opération 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, p. ex. 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 and MyThirdAltStep,
           // en l'occurrence
 :
```

22 Opérations 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 être utilisées que dans les tests élémentaires, fonctions et variantes TTCN-3 (c'est-à-dire qu'elles ne doivent 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

Opérations de configuration	
Instruction	**Nom de l'opération**
Création d'un composant de test parallèle	**create**
Connexion d'un composant à un autre composant	**connect**
Déconnexion de deux composants	**disconnect**
Affectation d'un accès de composant à un accès de l'interface avec le système de test	**map**
Désaffectation d'un accès de l'interface avec le système de test	**unmap**
Obtention d'adresse de composant MTC	**mtc**
Obtention d'adresse d'interface avec le système de test	**system**
Obtention d'adresses propres	**self**
Début d'exécution de composant de test	**start**

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

Opérations de configuration	
Instruction	**Nom de l'opération**
Arrêt d'exécution de composant de test	**stop**
Vérification de la terminaison d'un composant PTC	**running**
Attente de terminaison d'un composant PTC	**done**

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 d'accès, dont les files d'entrée sont vides. Par ailleurs, si un point d'accès est défini comme étant du type **in** ou **inout**, ce point 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.

Etant donné que tous les composants et accès sont implicitement détruits à la terminaison de chaque test élémentaire, celui-ci doit créer entièrement sa configuration requise de composants et de connexions quand il est invoqué.

```
// Cet exemple déclare une variable de type adresse, qui sert à mémoriser
// la référence d'un composant récemment créé de type MyComponentType qui
// est le résultat de l'opération "create".
 :
var MyComponenttype MyNewComponent;
 :
MyNewComponent := MyComponentType.create;
 :
```

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.

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" (affectation)

22.2.0 Généralités

Les accès d'un composant de test peuvent être connectés à d'autres composants ou aux accès de l'interface avec le système de test. Dans le cas de connexions entre deux composants de test, l'opération **connect** doit être utilisée. Lors de la connexion d'un composant de test à une interface avec le système de test, l'opération **map** doit être utilisée. L'opération **connect** relie directement un point d'accès 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. (Voir Figure 10.)

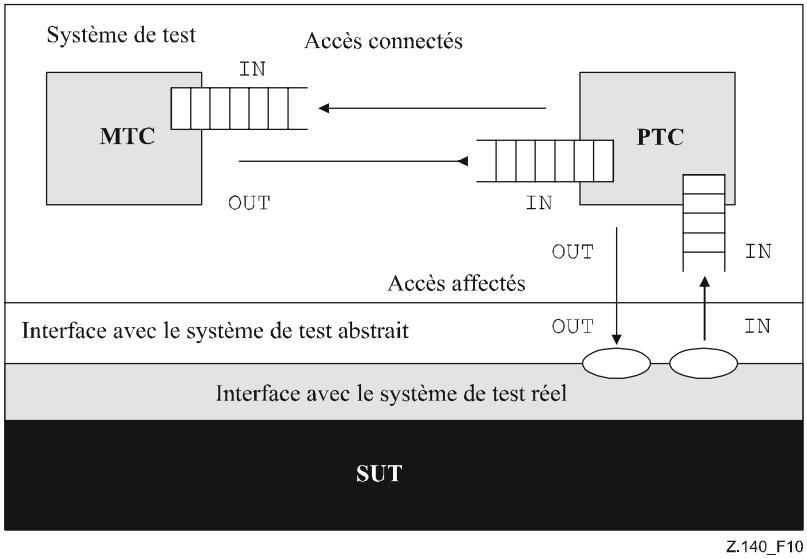

Figure 10/Z.140 – Illustration des opérations "connect" et "map"

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

L'opération **`mtc`** identifie le composant MTC et l'opération **`system`** identifie l'interface avec le système de test (voir § 22.4). Ces deux opérations peuvent servir à identifier et à connecter des points d'accès.

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 point d'accès applicable.

Les deux opérations **`map`** et **`connect`** permettent la connexion d'un point d'accès à plusieurs autres accès. Il n'est pas permis de connecter un composant à un point d'accès déjà affecté ni d'affecter un composant à un accès déjà connecté.

Exemple:

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

```
// "map", l'accès Port2 du nouveau composant est alors connecté au point
// d'accès PCO1 de l'interface avec le système de test
```

22.2.1 Connexions et affectations cohérentes

Dans les deux opérations **connect** et **map**, seules des connexions cohérentes sont autorisées.

Si l'on suppose ce qui suit:

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

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

c) outlist-PORT1définit 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 ⊆ inlist-PORT2 et si outlist-PORT2 ⊆ inlist-PORT1.

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

- outlist-PORT1 ⊆ outlist-PORT2 *et* si inlist-PORT2 ⊆ 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" (désaffectation)

Les opérations **disconnect** et **unmap** sont les opérations inverses des opérations **connect** et **map**. Elles effectuent la déconnexion d'accès (déjà connectés) de composants de test et la désaffectation d'accès de composants de test et d'accès de l'interface avec le système de test (déjà affectés).

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 accès 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:

```
 :
 :
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
```

22.4 Les opérations "MTC", "System" et "Self"

La référence de composant (voir § 8.7) comporte deux opérations, **mtc** et **system**, qui retournent respectivement la référence du composant de test principal et celle de l'interface avec le système de test. En outre, 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'attribution et l'équivalence.

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** (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.

Exemple:

```
// L'on part du principe que les accès Port1, Port2, Port3 et PCO1 sont
// correctement définis et déclarés dans les définitions correspondantes du
// type d'accès et du type de composant
 :
var MyComponentType MyNewPTC;
 :
MyNewPTC := MyComponentType.create;   // Création d'un nouveau composant de
                                      // test.
 :
connect(MyNewPTC:Port1, mtc:Port3);   // Connexion du nouveau composant
                                      // à son environnement.
map(MyNewPTCt:Port2, system:PCO1);
 :
 :
MyNewPTC.start(MyPTCBehaviour());     // début du nouveau 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 au même type de composant que le composant récemment créé;
- les accès et les temporisations ne doivent pas être transmis vers cette fonction.

NOTE – Etant donné que les accès 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 accès, déjà en attente de traitement.

22.6 L'opération "Stop" (arrêt de composant de test)

L'instruction **stop** permet de mettre fin à l'exécution d'un composant de test ou à l'exécution d'un autre composant de test. Si un composant ne s'arrête pas lui-même, mais arrête 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 s'arrêter lui-même au moyen d'une simple instruction **stop** (voir § 19.10) ou en s'adressant à lui-même dans l'opération **stop**, p. ex. au moyen de l'opération **self**. L'opération d'arrêt de composant de test n'a aucun argument.

Exemple 1:

NOTE 1 – Alors que les opérations **create**, **start**, **running** et **done** ne peuvent servir qu'à des composants PTC, l'opération **stop** peut servir aux deux composants, MTC et PTC.

```
var MyComponentType MyComp := MyComponentType.create; // Un nouveau composant de
                                                      // test est créé
MyComp.start(CompBehaviour());                  // Le nouveau composant est
                                                // lancé
 :
if (date == "1.1.2003") {
     MyComp.stop                          // Le nouveau composant est arrêté
                                          // au 1.1.2003
}

 :
if ( cond1 ) {
      :        // Un certain comportement
     self.stop      // Le composant de test s'arrête lui-même au moyen de
                    // l'opération "self"
}
else {
     stop           // Le composant de test s'arrête lui-même au moyen d'une
                    // simple opération "stop"
}
 :
```

Si le composant de test qui est arrêté est le composant MTC, tous les composants PTC restant en cours d'exécution doivent également être arrêtés et le test élémentaire s'achève (voir § 27.2).

NOTE 2 – Un composant PTC peut arrêter l'exécution d'un test élémentaire en arrêtant le composant MTC.

Toutes les ressources doivent être libérées quand un composant de test s'achève, soit explicitement au moyen de l'opération **stop** ou en atteignant une instruction **return** dans la fonction qui a lancé initialement le composant de test, ou implicitement quand le composant atteint la fin de sa définition comportementale. Toute variable mémorisant une référence de composant arrêté doit être indéfinie.

NOTE 3 – Une référence indéfinie signifie que la valeur ne peut servir à aucun calcul, c'est-à-dire qu'elle ne se rapporte à rien mais qu'elle n'est pas considérée comme étant de type **null**.

Les règles applicables à la terminaison de 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. Dans ce cas, le composant MTC ne sera pas arrêté. Il continue son exécution après l'instruction d'arrêt.

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 d'exécution ne peut servir qu'aux composants PTC. L'opération d'exécution 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, 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.

Quand le mot clé **all** est utilisé avec l'opération de fin d'exécution, celle-ci retournera la valeur **true** si aucun composant PTC n'est en train d'exécuter son comportement. Elle retourne également la valeur "true" si aucun composant PTC n'a été créé ou lancé. Sinon elle retourne la valeur **false**.

Quand le mot clé **any** est utilisé avec l'opération de fin d'exécution, celle-ci retournera la valeur **true** si au moins un composant PTC lancé mais non arrêté explicitement par un autre composant a fini d'exécuter son comportement. Sinon, elle retourne la valeur **false**.

NOTE – L'opération de fin d'exécution TTCN-3 (**done**) et l'opération de fin d'exécution TTCN-2 (DONE) ont une sémantique identique.

Exemple:

```
// Utilisation de l'opération "done" dans des instructions "alt"
:
```

```
alt {
     []   MyPTC.done {
                    setverdict(pass)
          }

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


// 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 Utilisation de séquences tabulaires de composants

Les opérations **create**, **connect**, **start** et **stop** ne s'appliquent pas directement à des séquences tabulaires de composants. En revanche, un élément spécifique de la séquence tabulaire doit être fourni comme paramètre. Dans les composants, l'effet d'une séquencela est obtenu au moyen d'une séquence de références de composant et par l'attribution de l'élément de séquence 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 séquences tabulaires de composants au
// moyen d'une boucle et au moyen de la mémorisation de la référence de
// composant créée dans une séquence de références de composant.

testcase MyTestCase() runs on MyMtcType system MyTestSystemInterface
{
      :
     var integer i;
     var MyPTCType1MyPtcType[11];
      :
     for (i:= 0; i<=10; i:=i+1)
     {
          MyPtc [i] := MyPtcType1.create;
          connect(self:PtcCoordination, MyPtcAddresses[i]:MtcCoordination);
          MyPtc [i].start(MyPTCBehaviour());
     }
      :
}
```

22.10 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
	`any`	`all`	
`create`			
`start`			
`running`	Oui mais à partir de MTC seulement	Oui mais à partir de MTC seulement	`any component.running` `all component.running`
`done`	Oui mais à partir de MTC seulement	Oui mais à partir de MTC seulement	`any component.done` `all component.done`
`stop`		Oui mais à partir de MTC seulement	`all component.stop`

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 accès et de commander l'arrivée à des accès 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 accès en mode message**	**Peut servir à un accès 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
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 accès			
Vérification de message/appel/exception/réponse reçu	`check`	Oui	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 accès en mode message**	**Peut servir à un accès en mode procédure**
Opérations de commande			
Libération d'accès	`clear`	Oui	Oui
Libération et ouverture d'accès	`start`	Oui	Oui
Fermeture d'accès (en réception et en émission)	`stop`	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 comme *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 point d'accès auquel l'opération spécifiée doit avoir lieu;
- définit la valeur des informations à transmettre;
- donne une expression d'adressage (facultative) qui identifie de façon univoque le correspondant de la communication dans le cas d'une connexion point-multipoint.

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

Exemple 1:

Partie envoi			Partie (facultative) réponse et manipulation d'exceptions
Accès et opération	**Partie valeur**	**Expression d'adressage (facultative)**	
`MyP1.`**`send`**	`(MyVariable + YourVariable - 2)`	**`to`** `MyPartner;`	

La partie réponse et manipulation d'exceptions n'est requises en cas de communication en mode procédure. La partie réponse et manipulation d'exception de l'opération d'appel est facultatif et est requis dans les cas où la procédure appelée retourne une valeur ou a paramètres **`out`** ou **`inout`** dont les valeurs sont requises dans l'appeling composant et dans les cas où la procédure appelée peut déclencher des exceptions qui nécessitent d'être manipulés par l'appeling composant.

La partie réponse et manipulation 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 manipulation d'exceptions
Accès et opération	**Partie valeur**	**Expression d'adressage (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) *attribution*.

La partie réception:

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

La partie attribution (facultative) dans une opération de réception est facultative. Aux accès en mode message, elle est utilisée quand il est nécessaire de mémoriser les messages reçus. Dans le cas des accès en mode procédure, elle est utilisée pour la mémorisation des paramètres **`in`** et **`inout`** d'un appel accepté ou pour la mémorisation d'exceptions. Pour la partie attribution, un typage fort est requis: p. ex., la variable utilisée pour mémoriser un message doit avoir le même type que le message entrant.

En outre, la partie attribution peut également servir à attribuer à une variable l'adresse de l'expéditeur (**`sender`**) d'un message, une exception, une réponse de type réponse 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 attribution (facultative)**		
Accès et opération	**Partie correspondante**	**Expression d'adressage (facultative)**		**Attribution de valeur (facultative)**	**Attribution de valeur paramétrique (facultative)**	**Attribution de valeur d'expéditeur (facultative)**
MyP1.**getreply**	(AProc:{?} **value** 5)		->		**param** (V1)	**sender** APeer

Partie réception				**Partie attribution (facultative)**		
Accès et opération	**Partie correspondante**	**Expression d'adressage (facultative)**		**Attribution de valeur (facultative)**	**Attribution de valeur paramétrique (facultative)**	**Attribution 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 possédant un certain critère de correspondance extrait d'un train de messages reçu à un certain accès entrant. Les messages situés au sommet de la file qui ne satisfont pas au critères de correspondance sont supprimés du point d'accès 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)

L'opération **send** sert à placer une valeur sur une file d'attente de messages sortant d'un accès. Cette valeur peut être spécifiée au moyen d'une référence à un modèle, à une variable, ou à une constante ou peut être définie en ligne à partir d'une expression (qui évidemment peut être une valeur

explicite). Lors de la définition de la valeur en ligne, le champ facultatif de type doit être utilisé s'il y a ambiguïté quant au type de la valeur en cours d'expédition.

L'opération **send** ne doit être utilisée qu'à des accès en mode message (ou mixte) et le type de la valeur à envoyer doit figurer dans la liste des types sortants de la définition du type d'accès.

Exemple 1:

```
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
```

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.send(charstring:"Ma chaîne") to MyPartner;     // envoie la chaîne
                                                      // "Ma chaîne" 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.
```

23.2.2 L'opération "Receive" (réception)

23.2.2.0 Généralités

L'opération **receive** sert à recevoir une valeur à partir d'une file d'attente de messages entrant dans un accès. La valeur peut être spécifiée au moyen d'une référence à un modèle, à une variable, ou à un constante ou peut être définie en ligne à partir d'une expression (qui évidemment peut être une valeur explicite). Lors de la définition de la valeur en ligne, le champ facultatif de type doit servir à éviter toute ambiguïté quant au type de la valeur reçue. L'opération **receive** ne doit être utilisée qu'à des accès en mode message (ou mixte) et le type de la valeur à recevoir doit être inclus dans la liste des types entrants de la définition du type d'accès.

L'opération **receive** supprime le message sommital de la file d'attente de messages entrants associée à un accès 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 accès, 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 peuvent être issus d'un modèle ou de la valeur résultant d'une expression (qui évidemment peut être une valeur explicite). 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 – 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 point d'accès MyPort.

MyPort.receive(A<B);    // correspond à une valeur booléenne qui dépend du
                        // résultat de A<B

MyPort.receive(integer:MyVar);   // correspond à une valeur d'entier avec
                                 // la valeur
                                 // de MyVar au point d'accès MyPort
MyPort.receive(MyVar);           // est une variante de l'exemple précédent

MyPort.receive(charstring:"Hello")from MyPeer;    // correspond à la chaîne
                                                  // "Hello" issue de MyPeer
```

Si la mise en correspondance est efficace, la valeur supprimée de la file d'attente à un accès peut être mémorisée dans une variable et l'adresse du composant qui a envoyé le message peut être extraite et mémorisée dans une variable. 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**.

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
                                      // attribuée à MyPeer

MyPort.receive(MyTemplate:{5, MyVarOne}) -> value MyVarTwo sender MyPeer;
// La valeur du message reçu est mémorisée dans MyVar et l'adresse de
// 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 accès si tous les autres critères de correspondance sont satisfaits.

NOTE – Cela est équivalent à l'instruction OTHERWISE de la notation TTCN-2.

Un message reçu par l'opération *ReceiveAnyMessage* ne doit pas être affecté à une variable.

Exemple:

```
MyPort.receive;              // 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 attribue l'adresse de
                                      // l'expéditeur à
                                      // MySenderVar
```

23.2.2.2 Réception à un accès quelconque

Afin de recevoir (**receive**) un message à tout point d'accès, il convient d'utiliser le mot clé **any**.

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 accès. Si ce message sommital satisfait les critères correspondants, l'opération **trigger** se comporte de la même façon qu'une opération **receive**. Si ce message sommital ne satisfait pas les critères correspondants, il doit être supprimé de la file sans aucune action complémentaire. L'opération **trigger** ne doit être utilisée qu'aux accès en mode message (ou mixte) et le type de la valeur à recevoir doit être inclus dans la liste des types entrants de la définition du type d'accès.

NOTE – La note du § 22.2.2 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 point
// d'accès MyPort.
```

L'opération **trigger** opération exige le nom du point d'accès, les critères de correspondance de type et de valeur, une restriction facultative **from** (c'est-à-dire une sélection de correspondant de communication) et une attribution facultative à des variables du composant correspondant de message et d'expéditeur.

Exemple 2:

```
MyPort.trigger(MyType:?) from MyPartner;
// Déclenche dès réception du premier message de type MyType au point
// d'accès MyPort, reçu de MyPartner.

MyPort.trigger(MyType:?) from MyPartner -> value MyRecMessage;
// Cet exemple est presque identique au précédent exemple. 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 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 attribué à une variable.

Exemple:

```
MyPort.trigger;

MyPort.trigger  from MyPartner;

MyPort.trigger -> sender MySenderVar;
```

23.2.3.2 Déclenchement à un accès quelconque

Le mot clé **any** est utilisé afin d'effectuer une opération **trigger** sur un message à un point d'accès 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 § 23.

Le schéma de principe des communications bloquantes en mode procédure est représenté dans la Figure 12. L'APPELANT appelle une procédure distante dans l'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 l'appel ou par une propagation (opération **raise**) d'exception. L'APPELANT manipule 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.

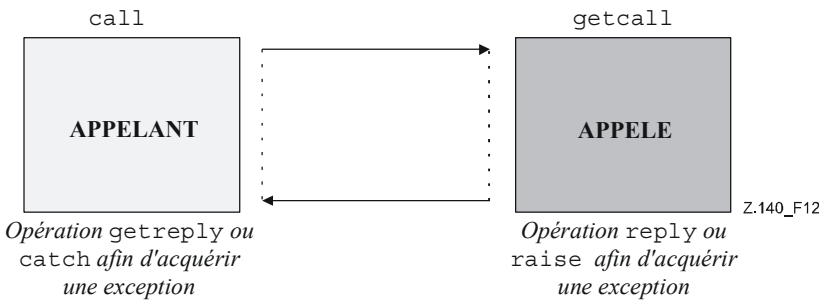

Figure 12/Z.140 – Illustration des communications bloquantes en mode procédure

Le principe des communications non bloquantes en mode procédure est représenté dans la Figure 13. L'APPELANT appelle une procédure distante dans l'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 manipuler l'exception au moyen d'une opération **catch** opération insérée dans une instruction **alt**. dans la Figure 13, le blocage de l'APPELE jusqu'à ce que la fin de la manipulation d'appel et déclenchement possible d'une exception sont indiqués au moyen d'une ligne pointillée.

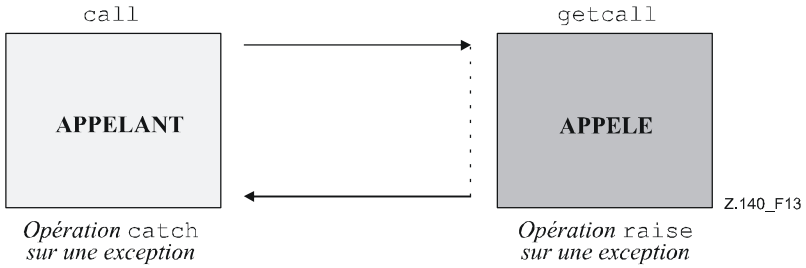

Figure 13/Z.140 – Illustration de communications bloquantes en mode procédure

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 points d'accès en mode procédure (ou mixte). La définition de type du point d'accès 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 cet accès.

Les informations à faire passer 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'attribution 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 manipulation d'exception de l'opération d'appel, au moyen des opérations **getreply** et **catch**. Cela permet d'utiliser des modèles de signature dans des opérations d'appel 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 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) à 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 ou si l'option **nowait** option est choisie (voir § 23.3.1.2), la manipulation des réponses ou exceptions à des opérations d'appel **call** est assurée au moyen des opérations **getreply** (voir § 23.3.2) et **catch** (voir § 23.3.6) sous forme d'options dans des instructions **alt**.

En cas de communications bloquantes en mode procédure, la manipulation de réponse ou exception à un appel est assurée dans la partie réponse et manipulation d'exception de l'opération d'appel, au moyen des opérations **getreply** (voir § 23.3.2) et **catch** (voir § 23.3.6).

La partie réponse et manipulation d'exception d'une opération d'appel 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. Autrement dit, 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 manipulation 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 (facultatives) sentinelles booléennes, peut inclure l'élément sommital (éventuel) du point d'accès auquel la procédure a été appelée et peut inclure une exception d'expiration de temporisateur 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 manipulation 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 Manipulation des exceptions d'expiration de temporisateur d'appel

L'opération d'appel 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 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 temporisateur **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 temporisateur située dans une opération d'appel 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 temporisateur.

Exemple 2:

```
MyPort.call(MyProc:{5, MyVar}, nowait); // Le composant de test d'appel
                                        // va continuer son exécution sans
                                        // attendre la terminaison de MyProc
```

Lorsque 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 accès au moyen d'une opération **getreply** ou **catch** opération 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 manipulation d'exception afin de manipuler 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 manipulation d'exception, ne doit déclencher aucune exception d'expiration de temporisateur et ne doit pas utiliser le mot clé **nowait**.

Les éventuelles exceptions propagée par procédure non bloquante doivent être supprimées de la file d'attente à un accès au moyen d'opérations de type **catch** opérations dans des instructions **alt** subséquentes.

23.3.2 L'opération "Getcall" (traitement d'appel)

23.3.2.0 Généralités

L'opération **getcall** opération 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 accès 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 d'accès.

L'opération **getcall** doit supprimer l'appel sommital de la file d'attente d'entrée à un accès, si et seulement si, les critères de correspondance associés à l'opération **getcall** sont satisfaits. Ces critères de correspondance sont associés à la signature de l'appel à traiter et au correspondant de la communication. Les critères de correspondance pour la signature peuvent soit être spécifiés en ligne ou être issus d'un modèle de signature.

Une opération **getcall** opération peut être limitée à un certain correspondant de communication dans le cas de connexions point-multipoint. Cette restriction doit être indiquée au moyen du mot clé **from**.

Exemple 1:

```
MyPort.getcall(MyProc(5, MyVar));
// accepte un appel de MyProc à l'accès MyPort

MyPort.getcall(MyProc:{5, MyVar}) from MyPeer;
// accepte un appel de MyProc à l'accès 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'attribution de valeurs paramétriques **in** et **inout** à des variables doit être effectuée dans la partie attribution de l'opération **getcall**. Cela permet d'utiliser des modèles de signature dans des opérations de type **getcall** opérations de la même façon que des modèles sont utilisés pour des types.

La partie attribution (facultative) de l'opération **getcall** inclut l'attribution à des variables de valeurs paramétriques **in** et **inout** et l'extraction de l'adresse du composant d'appel. Le mot clé **param** sert à extraire la valeur paramétrique d'un appel.

Le mot clé **sender** est utilisé quand il est requis d'extraire l'adresse de l'expéditeur (p. ex. afin d'adresser une réponse ou une exception à au correspondant de l'appel dans une configuration point-multipoint).

Exemple 2:

```
MyPort.getcall(MyProc:{?, ?}) from MyPartner -> param (MyPar1Var,
MyPar2Var);
// La valeur paramétrique in ou inout de MyProc est attribuée 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 d'appel est extraite et mémorisée dans
// MySenderVar.

// Les exemples suivants d'opération getcall montrent les possibilités
// d'utiliser des attributs de correspondance 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'attribuer 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'attribution à des variables de valeur de paramètre
// in ou inout. Noter que les noms contenus dans la liste d'attribution 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 de correspondance de signature va supprimer l'appel sommital de la file d'attente (éventuelle) d'entrée à un accès si tous les autres critères de correspondance 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 point d'accès MyPort

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 point d'accès quelconque

L'opération **getcall** à un point d'accès 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 accès en mode procédure (ou mixte). La définition de type du point d'accès 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.

Dans le cas de connexions point-multipoint, le correspondant de la communication doit être spécifié explicitement et doit être unique. Cela doit être indiqué au moyen du mot clé **to**.

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(MyProc3:{5,MyVar} value 20);  // réponses à un appel accepté de
                                           // MyProc2.
```

23.3.4 L'opération "Getreply" (traitement de réponse)

23.3.4.0 Généralités

L'opération **getreply** sert à manipuler les réponses à une procédure déjà appelée. Une opération **getreply** ne doit être utilisée qu'à un accès en mode procédure (ou mixte).

L'opération **getreply** opération doit supprimer la réponse sommitale de la file d'attente d'entrée à un accès, si et seulement si les critères de correspondance associés à l'opération **getreply** sont satisfaits. Ces critères de correspondance sont associés à la signature de la procédure à traiter et au correspondant de la communication. Les critères de correspondance pour la signature peuvent soit être spécifiés en ligne ou être issus d'un modèle de signature.

La mise en correspondance en fonction d'une valeur de retour reçue peut être spécifiée au moyen du mot clé **value**.

Une opération **getreply** opération peut être limitée à un certain correspondant de 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
                                                  // MyProc à partir de
                                                  // MyPeer
```

L'argument de signature de l'opérations **getreply** opération ne doit pas servir à faire passer des noms de variable vers des paramètres **out** et **inout**. L'attribution de valeurs paramétriques **out** et **inout** à des variables doit être effectuée dans la partie attribution de l'opération **getreply**. Cela permet d'utiliser des modèles de signature dans des opérations **getcall** de la même façon que des modèles sont utilisés pour des types.

La partie attribution (facultative) de l'opération **getreply** inclut l'attribution 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 requis d'extraire l'adresse de l'expéditeur.

Exemple 2:

```
MyPort.getreply(MyProc1:{?, ?} valeur ?) -> value MyRetValue
param(MyPar1,MyPar2);
// La valeur retournée est attribuée à la variable MyRetValue et la valeur
// des deux paramètres out ou inout est attribuée 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 lors de
// l'exécution subséquente du test et l'adresse du composant expéditeur est
// extraite telle que mémorisée dans la variable MySender.

// L'exemple suivant décrit certaines possibilités d'attribution 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 de correspondance de signature doit supprimer une message de réponse sommital de la file d'attente (éventuelle) d'entrée à un accès si tous les autres critères de correspondance sont satisfaits. Les paramètres ou valeurs de retour des réponses acceptées par l'opération *GetAnyReply* ne doivent pas être affectés à une variable.

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 accès quelconque

Afin de traiter une réponse à un point d'accès quelconque, il convient d'utiliser le mot clé **any**.

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 accès en mode procédure (ou mixte). Une exception est une réaction à un appel de procédure accepté, dont le résultat conduit à un événement exceptionnel. Le type de l'exception doit être spécifié dans la signature de la procédure appelée. La définition de type du point d'accès 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.

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:

```
MyPort.raise(MySignature, MyVariable + YourVariable – 2);
// Propage une exception avec une valeur qui est le résultat de
// l'expression arithmétique à l'accès MyPort

MyPort.raise(MyProc, integer:5});
// Propage une exception avec la valeur d'entier 5 pour MyProc

MyPort.raise(MySignature, "Ma chaîne") à MyPartner;
// Propage une exception avec la valeur"Ma chaîne" à MyPort pour MyProc et
// l'envoie à MyPeer
```

23.3.6 L'opération "Catch" (acquisition)

23.3.6.0 Généralités

L'opération **catch** sert à acquérir des exceptions propagées par une entité homologue en réaction à un appel de procédure. L'opération **catch** ne doit être utilisée qu'à des points d'accès 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 p. ex. servir à distinguer 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 accès 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.

Une opération **catch** peut être limitée à un certain correspondant de 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 point
                    // d'accès MyPort.

MyPort.catch(MyProc, MyVar);      // Variante de l'exemple précédent
MyPort.catch(MyProc, A<B);        // Acquiert une exception de type booléen

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 attribution (facultative) de l'opération **catch** inclut l'attribution 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 requis 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 attribue sa valeur à
// MyVar.

MyPort.catch(MyProc, MyTemplate(5)) -> value MyVarTwo sender MyPeer;
// Acquiert une exception à partir de MyPartner, attribue sa valeur à MyVar
// et extrait l'adresse de l'expéditeur.
```

L'opération **catch** peut faire partie de la partie réponse et manipulation d'exception d'une opération d'appel ou servir à déterminer une option dans une instruction **alt**. Si l'opération **catch** opération est utilisée dans la partie acceptrice d'une opération d'appel, les informations relatives au nom du point d'accès 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. Cependant, pour des raisons de lisibilité (p. ex. dans le cas d'instructions d'appel complexes), ces informations doivent être répétées.

23.3.6.1 L'exception d'expiration de temporisateur

Il y a une seule exception **timeout** spéciale, qui est 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 ms
                    setverdict(fail);
```

```
                stop;
        }
    }
```

L'acquisition d'exceptions de type **timeout** doit être limitée à la manipulation de la partie exception d'un appel. Aucun autre critère de correspondance (y compris une partie **from**) ni aucune partie attribution n'est autorisé pour une opération **catch** qui manipule 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** et d'une partie attribution. Cette instruction 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 accès quelconque

Afin d'acquérir **catch** une exception à un point d'accès 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 point d'accès sans supprimer cet élément sommital de la file. L'opération de vérification doit manipuler des valeurs d'un certain type au niveau de points d'accès en mode message et doit distinguer entre appels à accepter, exceptions à acquérir et réponses issues d'appels précédents à des points d'accès en mode procédure.

Les opérations de réception **receive**, **getcall**, **getreply** et **catch**, avec leurs parties de correspondance et d'attribution, sont utilisées par l'opération de vérification afin de définir la condition qui doit être vérifiée et à extraire la ou les valeurs de ses paramètres si nécessaire.

C'est l'élément *sommital* d'une file d'attente d'entrée à un accès 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 effectuée et l'opération de réception spécifiée dans l'opération de vérification est effectuée sur la copie. L'opération de vérification **check** échoue si l'opération de réception échoue; c'est-à-dire que les critères de correspondance 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 point d'accès en mode message, doit provoquer une erreur de test élémentaire.

NOTE – Dans la plupart des cas, l'utilisation correcte de l'opération de vérification peut être vérifiée statiquement, c'est-à-dire avant 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 point d'accès MyPort2 à partir
// de MyPartner

MyPort2.check(getreply(MyProc:{5, MyVar} value 20));
// Vérifications pour une réponse à partir de la procédure MyProc à MyPort
// 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 accès. L'opération *CheckAny* 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 attribution d'abrégé avec une clause **sender**.

Exemple:

```
MyPort.check;

MyPort.check(from MyPartner);

MyPort.check(-> sender MySenderVar);
```

23.4.2 Vérification à un accès quelconque

Afin de vérifier à un point d'accès quelconque, utiliser le mot clé **any**.

Exemple:

```
any port.check;
```

23.5 Opérations de commande des points d'accès de communication

23.5.0 Généralités

En notation TTCN-3, les opérations de commande en mode message, en mode procédure et en accès mixtes sont les suivantes:

- **clear**: supprimer le contenu d'une file d'attente d'entrée à un accès;
- **start**: commencer la détection à un point d'accès et ouvrir celui-ci;
- **stop**: arrêter la détection et interdire les opérations d'envoi à un point d'accès.

23.5.1 L'opération "Clear port" (libération d'accès)

L'opération **clear** supprime le contenu de la file *entrante* du point d'accès spécifié. Si la file d'attente à un accès est déjà vide, alors cette opération ne doit avoir aucun effet.

Exemple:

```
MyPort.clear; // libère le point d'accès MyPort
```

23.5.2 L'opération "Start port" (ouverture de point d'accès)

Si un point d'accès est défini comme permettant des opérations de réception de type **receive**, **getcall**, etc., l'opération **start** libère la file entrante du point d'accès nommé et commence la détection du trafic à ce point d'accès. Si celui-ci est défini de façon à permettre des opérations d'envoi, alors les opérations de type **send**, **call**, **raise**, etc., sont également autorisées à être effectuées à ce point d'accès.

Exemple:

```
MyPort.start; // ouvre le point d'accès MyPort
```

Par défaut, tous les points d'accès d'un composant doivent être lancés implicitement quand un composant est créé. L'opération d'ouverture de point d'accès provoquera la réouverture des points d'accès non fermés, en supprimant tous les messages en attente dans la file entrante.

23.5.3 L'opération "Stop port" (fermeture de point d'accès)

Si un point d'accès est défini comme permettant des opérations de réception comme **receive** et **getcall,** l'opération **stop** provoque l'arrêt de la détection au point d'accès 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 point d'accès MyPort
```

NOTE – Arrêter la détection au point d'accès signifie que toutes les opérations de réception définies avant l'opération de fermeture doivent être complètement effectuées avant que le fonctionnement du point d'accès soit suspendu.

Exemple 2:

```
MyPort.receive (MyTemplate1) -> RecPDU;    // la valeur reçue est décodée,
                                           // appariée en fonction de
                                           // MyTemplate1 et la valeur de
                                           // correspondance 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 point d'accès ne soit rouvert par une opération
                   // "start" subséquente)
MyPort.receive (MyTemplate2);    // Cette opération n'est pas exécutée
```

23.6 Utilisation de "Any" et de "All" avec des points d'accès

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 points d'accès

Opération	Mot clé autorisé		Exemple
	`any`	`all`	
`receive, trigger, getcall, getreply, catch, check`)	Oui		`any port.receive`
`connect/map`			
`start, stop, clear`		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 temporisateurs armés* et sa propre *liste de temporisateurs expirés*, 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 temporisateurs armés et la liste des temporisateurs expirés ne sont que des listes théoriques qui ne limitent pas l'implémentation des temporisateurs. D'autres structures de données, comme un ensemble où l'accès aux événements d'expiration de temporisateur n'est pas limité, p. ex. par l'ordre dans lequel les événements d'expiration de temporisateur 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 temporisateurs armés et une liste des temporisateurs expirés, qui gèrent les événements d'armement/désarmement des temporisateurs déclarés 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 temporisateur est placé dans la liste des temporisateurs expirés 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 temporisateurs expirés de l'unité de portée dans laquelle cette temporisation a été déclarée à un moment quelconque.

Tous les temporisateurs armés doivent automatiquement être annulés 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**
Armement de temporisateur	**`Start`**
Arrêt de temporisateur	**`Stop`**
Lecture de la durée écoulée	**`Read`**
Vérification d'armement de temporisateur	**`running`**
Evénement d'expiration de temporisateur	**`timeout`**

24.1 L'opération "Start timer" (armement de temporisateur)

L'opération d'armement **`start`** de temporisateur sert à indiquer qu'un temporisateur devrait être armé. Les valeurs de temporisation doivent être des nombres non négatifs de type **`float`** (c'est-à-dire supérieurs ou égaux à 0,0). Quand un temporisateur est armé, son nom est ajouté à la liste des temporisateurs armés (pour l'unité de portée indiquée).

Exemple:

```
MyTimer1.start;         // MyTimer1 est armé avec la durée par défaut
MyTimer2.start(20E-3);  // MyTimer2 est armé avec une durée de 20 ms

// Des éléments de séquences tabulaires de temporisations peuvent également
// être armés en boucle, par exemple
timer t_Mytimer [5];
var float v_timerValues [5];

for (var integer i := 0; i==4; i:=1)
  { v_timerValues [i] := 1.0 }

for (var integer i := 0; i==4; 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 du temporisateur. Quand une durée de temporisation est neutralisée, la nouvelle valeur s'applique seulement à l'instance actuelle du temporisateur et toute opération ultérieure de type **`start`** pour ce temporisateur, qui ne spécifie pas de durée, doit utiliser la durée par défaut.

Armer un temporisateur avec la valeur de temporisation 0,0 signifie que la temporisation arrive à expiration immédiatement. Armer un temporisateur avec une valeur négative de temporisation (p. ex. 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 en virgule flottante zéro (0.0) jusqu'à la valeur maximale indiquée par le paramètre de durée.

L'opération **`start`** peut être appliquée à un temporisateur armé, auquel cas celui-ci est arrêté et ré-armé. Toute entrée dans une liste des temporisateurs expirés pour ce temporisateur doit être supprimée de cette liste.

24.2 L'opération "Stop timer" (désarmement de temporisateur)

L'opération **stop** sert à arrêter un temporisateur armé et à le supprimer de la liste des temporisateurs armés. Un temporisateur arrêté devient inactif et sa durée écoulée est réglée à la valeur en virgule flottante zéro (0.0).

L'arrêt d'un temporisateur inactif est une opération valide, bien qu'elle n'ait aucun effet. Toute entrée dans une liste des temporisateurs expirés pour ce temporisateur doit être supprimée de cette liste.

Le mot clé **all** peut servir à arrêter tous les temporisateurs 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 tous les temporisateurs armés
```

24.3 L'opération "Read timer" (lecture de temporisateur)

L'opération **read** opération sert à extraire la durée écoulée depuis que le temporisateur spécifié a été armé et à la mémoriser dans la variable spécifiée, qui doit être de type **float**.

Exemple:

```
var float Myvar;
MyVar := MyTimer1.read; // Attribuer à MyVar la durée écoulée depuis que
                        // MyTimer1 a été armé
```

L'application de l'opération **read** à un temporisateur inactif, c'est-à-dire non énuméré dans la liste des temporisateurs armés, retournera la valeur zéro.

24.4 L'opération "Running timer" (temporisateur armé)

L'opération de temporisateur armé **running** sert à vérifier si un temporisateur est énuméré dans la liste des temporisateurs armés de l'unité de portée indiquée (c'est-à-dire qu'il a été armé et n'a ni expiré ni été arrêté). L'opération retourne la valeur **true** si le temporisateur est énuméré dans cette liste, **false** sinon.

Exemple:

```
if (MyTimer1.running) { … }
```

24.5 L'opération "Timeout" (expiration de temporisateur)

L'opération d'expiration de temporisateur **timeout** permet de vérifier l'expiration d'un temporisateur, ou de tous les temporisateurs, dans une unité de portée d'un composant de test ou dans le module de commande dans lequel l'opération d'expiration de temporisateur a été appelée.

Quand une opération **timeout** est traitée, si un nom de temporisateur est indiqué, les listes de temporisateurs expirés du composant ou du module de commande sont explorées conformément aux règles de portées TTCN-3. S'il y a un événement d'expiration de temporisateur correspondant au nom de ce temporisateur, cet événement est supprimé de la liste des temporisateurs expirés 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** instruction 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 temporisateur(s).

NOTE – L'opération **timeout** en notation TTCN-3 et l'opération TIMEOUT en notation TTCN-2 ont la même sémantique.

Exemple 1:

```
MyTimer1.timeout;
// vérifications d'expiration du temporisateur MyTimer1 déjà armé
```

Le mot clé **any** utilisé avec l'opération **timeout** (plutôt qu'un temporisateur explicitement nommé) est efficace si la liste des temporisateurs expirés n'est pas vide.

Exemple 2:

```
any timer.timeout;
// Vérifications d'expiration de tout temporisateur déjà armé
```

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 verdicts au moyen, respectivement, des opérations **getverdict** et **setverdict**. 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 son instanciation. Il sert à suivre le verdict individuel dans chaque composant de test (c'est-à-dire dans le composant MTC et dans chacun des composants PTC).

NOTE – Contrairement à la notation TTCN-2, aucun verdict final ne peut être affecté à un composant de test. L'attribution d'un verdict n'arrête donc jamais l'exécution du composant de test dans lequel le comportement est en train d'exécuter. Au besoin, cela doit être explicitement indiqué au moyen de l'instruction **stop**.

25.1 Verdict de test élémentaire

Il y a par ailleurs un verdict global 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 (p. ex. attribué à une variable), alors il est perdu.

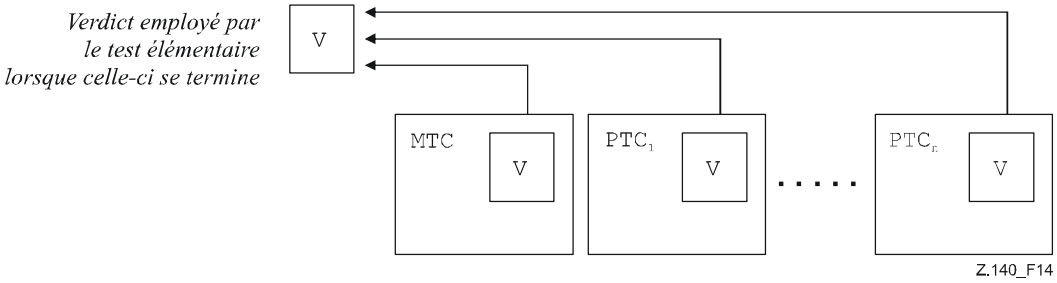

Figure 14/Z.140 – Illustration de la relation entre verdicts

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 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 (p. ex. une interface de gestion) mais il peut être nécessaire que le système SUT soit invité à effectuer certaines actions (p. ex. envoyer un message au système de test). Certaines actions peuvent également être requises de la part du personnel d'exécution des tests (p. ex. afin de modifier les conditions climatiques des tests comme la température, la tension de l'alimentation en énergie, etc.).

L'action requise peut être définie comme une chaîne.

Exemple 1:

```
action("Envoyer MyTemplate sur l'accès PCO inférieur");
// Description informelle de l'action du système SUT.
```

ou comme une référence à un modèle qui spécifie la structure du message à envoyer par le système SUT.

Exemple 2:

```
action(MyTemplate);
// Cela est équivalent à l'instruction IMPLICIT SEND de la notation TTCN-2.
```

Dans les deux cas, 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 spécification informelle de l'action requise proprement dite.

Les actions de système SUT peuvent être spécifiées 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, temporisations, etc. (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 ni les temporisateurs mondiaux 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 accès 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 test élémentaire

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`**, **`inconclusive`**, **`fail`** ou **`error`** doit être retourné et peut être attribué 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 test élémentaire

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 arrêtés 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 PTC, le composant MTC devrait veiller à ce que tous les composants PTC soient arrêtés (au moyen de l'instruction **`done`**) 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 peut être exécuté.

Exemple 1:

```
module MyTestSuite () {
      :
     control {
           :
          // Do this test 10 times
          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.

Les tests élémentaires retournent une valeur unique de type **verdicttype** de sorte qu'il est possible de commander l'ordre d'exécution en fonction du résultat d'un test élémentaire.

Exemple 2:

```
if (execute (MySimpleTestCase()) == pass) { execute (MyGoOnTestCase) }
else { execute (MyErrorRecoveryTestCase) };
```

27.4 Sélection de test élémentaire

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**.

NOTE – Cela est équivalent aux expressions de sélection de tests nommés en notation TTCN-2.

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 (MySelectionExpression1()) { MyTestCaseGroup2(); }
      :
}
```

27.5 Utilisation de temporisateurs dans les commandes

Les temporisateurs peuvent 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 le test élémentaire
// 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 ou x<10)  // Où T1 est un temporisateur déjà armé
{    execute(MyTestCase());
     x := x+1;
}

// Exemple de l'utilisation des opérations "start" et "timeout"

timer T1 := 1;
 :
execute(MyTestCase1());
T1.start;
T1.timeout;   // Pause avant l'exécution du prochain test élémentaire
execute(MyTestCase2());
```

28 Spécification des attributs

28.0 Généralités

Des attributs peuvent être associés aux éléments linguistiques 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 simplement 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 linguistiques TTCN-3 peuvent avoir des attributs de type **`display`** afin de spécifier la façon dont des éléments linguistiques 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 point d'accès 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.

Des chaînes d'attributs spécialement associées aux attributs de codage ASN.1 peuvent être trouvées dans l'Annexe D.

La façon dont les règles de codage effectives sont définies (p. ex. 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 définition**: 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 globale
     :
     group MyRecords
     {    :
          type record MyPDU1
          {
          integer   field1,   // field1 sera codé conformément à la règle 3
          boolean   field2,   // field2 sera codé conformément à la règle 3
          Mytype    field3    // field3 sera codé conformément à la règle 2
          }
          with { encode (field1, field2) "Rule 3" }
          :
     }
     with { encode "Rule 2" }

}
with { encode "Règle de codage globale" }
```

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.

Exemple:

```
module MyTTCNmodule1
{    :
     type charstring MyType;
     // Codage normal conformément à la règle 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) "forme de longueur 3" }
            :
        }
        with { encode "Rule 2" }

    }
    with { encode "Règle de codage globale" }
```

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 Annexe F).

Les chaînes suivantes sont les attributs prédéfinis (normalisés) de type **variant** pour **char**, **universal char**, **charstring** et **universal charstring** (voir § E.2.2):

a) l'attribut "UTF-8" signifie, quand il est appliqué aux types "universal char" et "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 [6];

b) l'attribut "UCS-2" signifie, quand il est appliqué aux types "universal char" et "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 [6]);

c) l'attribut "UTF-16" signifie, quand il est appliqué aux types "universal char" et "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 [6];

d) l'attribut "8 bit" signifie, quand il est appliqué aux types "char", "universal char", "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) (voir Annexe F).

Les chaînes suivantes sont les attributs prédéfinis (normalisés) 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 Annexe F).

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 § D.1.5.1), 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 linguistiques 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 linguistique particulier. Il est également possible d'associer des attributs à un certain nombre d'éléments linguistiques, p. ex. 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 linguistiques.

Exemple:

```
// MyPDU1 sera affichée comme PDU
type record MyPDU1 { … } avec { 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 de groupe MyPDUs sera affichée comme PDU est identique à
group MyPDUs {
     type record MyPDU3 { … } with { display "PDU"}
     type record MyPDU4 { … } with { display "PDU"}
}
```

28.5 Règles de surécriture pour attributs

Une définition d'attribut située dans unité de portée inférieure neutralisera un définition générale d'attribut dans une unité de portée supérieure.

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 surécrits conformément à leur ordre d'apparition.

```
// Exemple de l'utilisation du procédé de surécriture 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 le
                                      // l'attribut d'extension propre à
                                      // l'application MySpecialRule
}
with
{
     display "PDU";        // Tous les types d'unité MPDU de groupe seront
     extension "MyRule";   // affichés comme des unités PDU et auront (s'ils
                           // ne sont pas surécrits)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" }
     groupe 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 surécrite 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
{
      :
     champA    MyRecordA
} with { encode override "RuleB" }
```

L'instruction **override** force à l'attribut spécifié tous les types contenus dans des unités de portée inférieure.

28.6 Modification des attributs d'éléments linguistiques importés

En général, un élément linguistique est importé en même temps que ses attributs. Dans certains cas, ces attributs peuvent nécessiter une modification lors de l'importation de l'élément linguistique, p. ex. 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.

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 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 linguistiques 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 linguistique se termine par une accolade de fermeture (}) oui s le symbole suivant est une accolade de fermeture (}), c'est-à-dire que la structure linguistique est la dernière instruction d'un bloc d'instructions, 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 */ autorisé */
```

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.

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.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'attribution	:=
Attribution 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

Tableau A.3/Z.140 – Liste des symboles terminaux TTCN-3 qui sont des mots réservés

action	fail	noblock	self
activate	false	none	send
address	float		sender
all	for	not	set
alt	from	not4b	setverdict
altstep	function	nowait	signature
and		null	start
and4b	getverdict		stop
any	getcall	objid	subset
anytype	getreply	octetstring	superset
	goto	of	system
bitstring	group	omit	
boolean		on	template
	hexstring	optional	testcase
call		or	timeout
catch	if	or4b	timer
char	ifpresent	out	to
charstring	import	override	trigger
check	in		true
clear	inconc	param	type
complement	infinity	Modulepar	
component	inout	pass	union
connect	integer	pattern	universal
const	interleave	port	unmap
control		procedure	
create	label		value
	language	raise	valueof
deactivate	length	read	var
default	log	receive	variant
disconnect		record	verdicttype
display	map		
do	match	recursive	while
done	message	rem	with
	mixed	repeat	
else	mod	reply	xor
encode	modifies	return	xor4b
enumerated	module	running	
error	mtc	runs	
except			
exception			
execute			
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 terminaux doivent être écrits entièrement en lettres minuscules.

A.1.6 Productions BNF de la syntaxe TTCN-3

A.1.6.0 Module TTCN

```
1. TTCN3Module ::= TTCN3ModuleKeyword TTCN3ModuleId
                BeginChar
                [ModuleDefinitionsPart]
                [ModuleControlPart]
                EndChar
                [WithStatement] [SemiColon]
2. TTCN3ModuleKeyword ::= "module"
3. TTCN3ModuleId ::= ModuleIdentifier [DefinitiveIdentifier]
4. ModuleIdentifier ::= Identifier
5. DefinitiveIdentifier ::= Dot ObjectIdentifierKeyword "{"
DefinitiveObjIdComponentList "}"
6. DefinitiveObjIdComponentList ::= {DefinitiveObjIdComponent}+
7. DefinitiveObjIdComponent ::=   NameForm |
                DefinitiveNumberForm |
                DefinitiveNameAndNumberForm
8. DefinitiveNumberForm ::= Number
9. DefinitiveNameAndNumberForm ::= Identifier  "(" DefinitiveNumberForm ")"
```

A.1.6.1 Partie d'un module relative aux définitions

A.1.6.1.0 Généralités

```
10. ModuleDefinitionsPart ::= ModuleDefinitionsList
11. ModuleDefinitionsList ::= {ModuleDefinition [SemiColon]}+
12. ModuleDefinition ::=    (TypeDef |
                ConstDef |
                TemplateDef |
                ModuleParDef |
                FunctionDef |
                SignatureDef |
                TestcaseDef |
                AltstepDef |
                ImportDef |
                GroupDef |
                ExtFunctionDef |
                ExtConstDef) [WithStatement]
```

A.1.6.1.1 TypeDef

```
13. TypeDef ::= TypeDefKeyword TypeDefBody
14. TypeDefBody ::= StructuredTypeDef | SubTypeDef
15. TypeDefKeyword ::= "type"
16. StructuredTypeDef ::= RecordDef |
                          UnionDef |
                          SetDef |
                          RecordOfDef |
                          SetOfDef |
                          EnumDef |
                          PortDef |
                          ComponentDef
17. RecordDef ::= RecordKeyword StructDefBody
18. RecordKeyword ::= "record"
19. StructDefBody ::= (StructTypeIdentifier  [StructDefFormalParList] |
AddressKeyword)
                BeginChar
                [StructFieldDef {"," StructFieldDef}]
EndChar
20 StructTypeIdentifier ::= Identifier
21. StructDefFormalParList ::= "(" StructDefFormalPar {"," StructDefFormalPar}
")"
```

```
22. StructDefFormalPar ::=  FormalValuePar
/* SÉMANTIQUE STATIQUE – FormalValuePar doit se réduire à un paramètre de type
"in" */
23. StructFieldDef ::= Type StructFieldIdentifier [ArrayDef] [SubTypeSpec]
[OptionalKeyword]
24. StructFieldIdentifier ::= Identifier
25. OptionalKeyword ::= "optional"
26. UnionDef ::= UnionKeyword UnionDefBody
27. UnionKeyword ::= "union"
28. UnionDefBody ::= (StructTypeIdentifier [StructDefFormalParList] |
AddressKeyword)
               BeginChar
               UnionFieldDef {"," UnionFieldDef}
               EndChar
29. UnionFieldDef ::= Type StructFieldIdentifier [ArrayDef] [SubTypeSpec]
30. SetDef ::= SetKeyword StructDefBody
31. SetKeyword ::= "set"
32. RecordOfDef ::= RecordKeyword [StringLength] OfKeyword StructOfDefBody
33. OfKeyword ::= "of"
34. StructOfDefBody ::= Type (StructTypeIdentifier | AddressKeyword)
[SubTypeSpec]
35. SetOfDef ::= SetKeyword [StringLength] OfKeyword StructOfDefBody
36. EnumDef ::= EnumKeyword (EnumTypeIdentifier | AddressKeyword)
               BeginChar
               EnumerationList
               EndChar
37. EnumKeyword ::= "enumerated"
38. EnumTypeIdentifier ::= Identifier
39. EnumerationList ::= Enumeration {"," Enumeration}
40. Enumeration ::= EnumerationIdentifier ["(" Number ")"]
41. EnumerationIdentifier ::= Identifier
42. SubTypeDef ::= Type (SubTypeIdentifier | AddressKeyword)  [ArrayDef]
[SubTypeSpec]
43. SubTypeIdentifier ::= Identifier
44. SubTypeSpec ::= AllowedValues | StringLength
/* SÉMANTIQUE STATIQUE – Les valeurs AllowedValues doivent être du même type que
le champ qui est en cours de sous-typage */
45. AllowedValues ::= "(" ValueOrRange {"," ValueOrRange} ")"
46. ValueOrRange ::= RangeDef | SingleConstExpression
/* SÉMANTIQUE STATIQUE – La production RangeDef ne doit être utilisée qu'avec
les types integer, char, universal char, charstring, universal charstring ou
float */
/* SÉMANTIQUE 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 */
47. RangeDef ::= LowerBound ".." UpperBound
48. StringLength ::= LengthKeyword "(" SingleConstExpression [".." UpperBound]
")"
/* SÉMANTIQUE 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) */
49.  LengthKeyword ::= "length"
50. PortType ::=   [GlobalModuleId Dot] PortTypeIdentifier
51. PortDef ::= PortKeyword PortDefBody
52. PortDefBody ::= PortTypeIdentifier PortDefAttribs
53. PortKeyword ::= "port"
54. PortTypeIdentifier ::= Identifier
55. PortDefAttribs ::= MessageAttribs | ProcedureAttribs | MixedAttribs
56. MessageAttribs ::= MessageKeyword
               BeginChar
               {MessageList  [SemiColon]}+
               EndChar
```

```
57. MessageList ::= Direction AllOrTypeList
58. Direction ::= InParKeyword | OutParKeyword | InOutParKeyword
59. MessageKeyword ::= "message"
60. AllOrTypeList ::= AllKeyword | TypeList
61. AllKeyword ::= "all"
62. TypeList ::= Type {"," Type}
63. ProcedureAttribs ::= ProcedureKeyword
                BeginChar
                {ProcedureList  [SemiColon]}+
                EndChar
64. ProcedureKeyword ::= "procedure"
65. ProcedureList ::= Direction AllOrSignatureList
66. AllOrSignatureList ::= AllKeyword | SignatureList
67. SignatureList ::= Signature {"," Signature}
68. MixedAttribs ::= MixedKeyword
                BeginChar
                {MixedList  [SemiColon]}+
                EndChar
69. MixedKeyword ::= "mixed"
70. MixedList ::= Direction ProcOrTypeList
71. ProcOrTypeList ::= AllKeyword | (ProcOrType {"," ProcOrType})
72. ProcOrType ::= Signature | Type
73. ComponentDef ::= ComponentKeyword ComponentTypeIdentifier
                BeginChar
                [ComponentDefList]
                EndChar
74. ComponentKeyword ::= "component"
75. ComponentType ::=   [GlobalModuleId Dot] ComponentTypeIdentifier
76. ComponentTypeIdentifier ::= Identifier
77. ComponentDefList ::= {ComponentElementDef [SemiColon]}
78. ComponentElementDef ::= PortInstance | VarInstance | TimerInstance |
ConstDef
79. PortInstance ::= PortKeyword PortType PortElement {"," PortElement}
80. PortElement ::= PortIdentifier [ArrayDef]
81. PortIdentifier ::= Identifier
```

A.1.6.1.2 Définitions de constante

```
82. ConstDef ::= ConstKeyword Type ConstList
83. ConstList ::= SingleConstDef  {"," SingleConstDef}
84. SingleConstDef ::= ConstIdentifier [ArrayDef] AssignmentChar
ConstantExpression
/* SÉMANTIQUE STATIQUE – La valeur de la production ConstantExpression doit être
du même type que celui qui est indiqué pour les constantes */
85. ConstKeyword ::= "const"
86. ConstIdentifier ::= Identifier
```

A.1.6.1.3 Définitions de modèle

```
87. TemplateDef ::=     TemplateKeyword BaseTemplate  [DerivedDef]
AssignmentChar TemplateBody
88. BaseTemplate ::= (Type | Signature) TemplateIdentifier  ["("
TemplateFormalParList ")"]
89. TemplateKeyword ::= "template"
90. TemplateIdentifier ::= Identifier
91. DerivedDef ::= ModifiesKeyword TemplateRef
92. ModifiesKeyword ::= "modifies"
93. TemplateFormalParList ::= TemplateFormalPar {"," TemplateFormalPar}
94. TemplateFormalPar ::=   FormalValuePar | FormalTemplatePar
/* SÉMANTIQUE STATIQUE – La production FormalValuePar doit correspondre à un
paramètre de type "in" */
```

```
95. TemplateBody ::= SimpleSpec | FieldSpecList | ArrayValueOrAttrib
/* SÉMANTIQUE STATIQUE – À l'intérieur de la production TemplateBody, la
production ArrayValueOrAttrib peut être utilisée pour les types array, record,
record of et set of. */
96. SimpleSpec ::= SingleValueOrAttrib
97. FieldSpecList ::= "{"[FieldSpec {"," FieldSpec}] "}"
98. FieldSpec ::= FieldReference AssignmentChar TemplateBody
99. FieldReference ::= StructFieldRef | ArrayOrBitRef | ParRef
100. StructFieldRef ::= StructFieldIdentifier
101. ParRef ::= SignatureParIdentifier
/* SÉMANTIQUE OPÉRATIONNELLE – La production SignatureParIdentifier doit être un
paramètre formel d'identification extrait de la définition de signature associée
*/
102. SignatureParIdentifier ::= ValueParIdentifier
103. ArrayOrBitRef ::= "[" FieldOrBitNumber "]"
/* SÉMANTIQUE 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 "record of" et "set of". La même notation peut servir à une
référence binaire à l'intérieur d'un type "bitstring" en notation ASN.1 ou TTCN
*/
104. FieldOrBitNumber ::= SingleExpression
/* SÉMANTIQUE STATIQUE – SingleExpression va correspondre à une valeur de type
entier (integer) */
105. SingleValueOrAttrib ::=     MatchingSymbol [ExtraMatchingAttributes] |
               SingleExpression [ExtraMatchingAttributes] |
               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
*/
106. ArrayValueOrAttrib ::= "{" ArrayElementSpecList "}"
107. ArrayElementSpecList ::= ArrayElementSpec {"," ArrayElementSpec}
108. ArrayElementSpec ::= NotUsedSymbol | TemplateBody
109. NotUsedSymbol ::= Dash
110. MatchingSymbol ::=     Complement |
                           AnyValue |
                           AnyOrOmit |
                           ValueOrAttribList |
                           Range |
                           BitStringMatch |
                           HexStringMatch |
                           OctetStringMatch |
                           CharStringMatch |
                           SubsetMatch |
                           SupersetMatch
111. ExtraMatchingAttributes ::= LengthMatch | IfPresentMatch | LengthMatch
IfPresentMatch
112. BitStringMatch ::= "'" {BinOrMatch} "'" "B"
113. BinOrMatch ::= Bin | AnyValue | AnyOrOmit
114. HexStringMatch ::= "'" {HexOrMatch} "'" "H"
115. HexOrMatch ::= Hex | AnyValue | AnyOrOmit
116. OctetStringMatch ::= "'" {OctOrMatch} "'" "O"
117. OctOrMatch ::= Oct | AnyValue | AnyOrOmit
118. CharStringMatch ::= PatternKeyword Cstring
119. PatternKeyword ::= "pattern"
120. Complement ::= ComplementKeyword ValueList
121. ComplementKeyword ::= "complement"
122. ValueList ::= "(" ConstantExpression {"," ConstantExpression} ")"
123. SubsetMatch ::= SubsetKeyword ValueList
/* SEMANTIQUE STATIQUE – Le sous-ensemble correspondant ne doit être utilisé
qu'avec le type "set of" */
124. SubsetKeyword ::= "subset"
```

```
125. SupersetMatch ::= SupersetKeyword ValueList
/* SEMANTIQUE STATIQUE – L'appariement d'un type "superset" ne doit être utilisé
qu'avec le type "set of" */
126. SupersetKeyword ::= "superset"
127. AnyValue ::= "?"
128. AnyOrOmit ::= "*"
129. ValueOrAttribList ::= "(" TemplateBody {"," TemplateBody}+ ")"
130. LengthMatch ::= StringLength
131. IfPresentMatch ::= IfPresentKeyword
132. IfPresentKeyword ::= "ifpresent"
133. Range ::= "(" LowerBound ".." UpperBound ")"
134. LowerBound ::= SingleConstExpression | Minus InfinityKeyword
135. UpperBound ::= SingleConstExpression | InfinityKeyword
/* SÉMANTIQUE STATIQUE – Les productions LowerBound et UpperBound doivent
s'évaluer comme des types integer, char, universal char ou float. Si LowerBound
ou UpperBound s'évalue comme un type char ou universal char, seule la production
SingleConstExpression peut être présente */
136. InfinityKeyword ::= "infinity"
137. TemplateInstance ::= InLineTemplate
138. TemplateRefWithParList ::=  [GlobalModuleId Dot] TemplateIdentifier
[TemplateActualParList] | TemplateParIdentifier
139. TemplateRef ::=    [GlobalModuleId Dot] TemplateIdentifier |
TemplateParIdentifier
140. InLineTemplate ::= [(Type | Signature) Colon] [DerivedDef AssignmentChar]
TemplateBody
/* SÉMANTIQUE STATIQUE – Le champ de type ne peut être omis que quand le type
est implicitement univoque */
141. TemplateActualParList ::= "(" TemplateActualPar {"," TemplateActualPar} ")"
142. 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 */
143. TemplateOps ::= MatchOp | ValueofOp
144. 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 */
145. MatchKeyword ::= "match"
146. ValueofOp ::= ValueofKeyword "(" TemplateInstance")"
147. ValueofKeyword ::= "valueof"
```

A.1.6.1.4 Définition de fonctions

```
148. FunctionDef ::= FunctionKeyword FunctionIdentifier
               "("[FunctionFormalParList] ")" [RunsOnSpec] [ReturnType]
StatementBlock
149. FunctionKeyword ::= "function"
150. FunctionIdentifier ::= Identifier
151. FunctionFormalParList ::= FunctionFormalPar {"," FunctionFormalPar}
152. FunctionFormalPar ::= FormalValuePar |
               FormalTimerPar |
               FormalTemplatePar |
               FormalPortPar
153. ReturnType ::= ReturnKeyword Type
154. ReturnKeyword ::= "return"
155. RunsOnSpec ::= RunsKeyword OnKeyword ComponentType
156. RunsKeyword ::= "runs"
157. OnKeyword ::= "on"
158. MTCKeyword ::= "mtc"
159. StatementBlock ::=     BeginChar [FunctionStatementOrDefList] EndChar
160. FunctionStatementOrDefList ::= {FunctionStatementOrDef [SemiColon]}+
161. FunctionStatementOrDef ::= FunctionLocalDef |
               FunctionLocalInst |
               FunctionStatement
```

```
162. FunctionLocalInst ::= VarInstance | TimerInstance
163. FunctionLocalDef ::= ConstDef
164. FunctionStatement ::= ConfigurationStatements |
               TimerStatements |
               CommunicationStatements |
               BasicStatements |
               BehaviourStatements |
               VerdictStatements |
               SUTStatements
165. FunctionInstance ::= FunctionRef "(" [FunctionActualParList] ")"
166. FunctionRef ::= [GlobalModuleId Dot] (FunctionIdentifier |
ExtFunctionIdentifier ) | PreDefFunctionIdentifier
167. PreDefFunctionIdentifier ::= Identifier
/* SÉMANTIQUE STATIQUE – L'identificateur sera un des identificateurs de
fonction prédéfinis en notation TTCN-3 d'après l'Annexe C de la présente
Recommandation*/
168. FunctionActualParList ::= FunctionActualPar {"," FunctionActualPar}
169. 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

```
170. SignatureDef ::= SignatureKeyword SignatureIdentifier
               "("[SignatureFormalParList] ")" [ReturnType | NoBlockKeyword]
               [ExceptionSpec]
171. SignatureKeyword ::= "signature"
172. SignatureIdentifier ::= Identifier
173. SignatureFormalParList ::= SignatureFormalPar {"," SignatureFormalPar}
174. SignatureFormalPar ::=  FormalValuePar
175. ExceptionSpec ::= ExceptionKeyword "(" ExceptionTypeList ")"
176. ExceptionKeyword ::= "exception"
177. ExceptionTypeList ::= Type {"," Type}
178. NoBlockKeyword ::= "noblock"
179. Signature ::= [GlobalModuleId Dot] SignatureIdentifier
```

A.1.6.1.6 Définitions de test élémentaire

```
180. TestcaseDef ::= TestcaseKeyword TestcaseIdentifier
               "("[TestcaseFormalParList] ")" ConfigSpec
StatementBlock
181. TestcaseKeyword ::= "testcase"
182. TestcaseIdentifier ::= Identifier
183. TestcaseFormalParList ::= TestcaseFormalPar {"," TestcaseFormalPar}
184. TestcaseFormalPar ::= FormalValuePar |
               FormalTemplatePar
185. ConfigSpec ::= RunsOnSpec [SystemSpec]
186. SystemSpec ::= SystemKeyword ComponentType
187. SystemKeyword ::= "system"
188. TestcaseInstance ::=   ExecuteKeyword "(" TestcaseRef "("
[TestcaseActualParList] ")" ["," TimerValue] ")"
189. ExecuteKeyword ::= "execute"
190. TestcaseRef ::= [GlobalModuleId Dot] TestcaseIdentifier
191. TestcaseActualParList ::= TestcaseActualPar {"," TestcaseActualPar}
192. TestcaseActualPar ::=
               TemplateInstance
/* SÉMANTIQUE STATIQUE – 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

```
193. AltstepDef ::= AltstepKeyword AltstepIdentifier
               "("[AltstepFormalParList] ")" [RunsOnSpec]
               BeginChar
               AltstepLocalDefList
               AltGuardList
               EndChar
194. AltstepKeyword ::= "altstep"
195. AltstepIdentifier ::= Identifier
196. AltstepFormalParList ::= FunctionFormalParList
/* SÉMANTIQUE STATIQUE – Tous les paramètres formels doivent impérativement être
des paramètres de valeur c'est-à-dire des paramètres de type "in" */
197. AltstepLocalDefList ::= {AltstepLocalDef [SemiColon]}
198. AltstepLocalDef ::= VarInstance | TimerInstance | ConstDef
199. AltstepInstance ::=    AltstepRef "(" [FunctionActualParList] ")"
200. AltstepRef ::= [GlobalModuleId Dot] AltstepIdentifier
```

A.1.6.1.8 Définitions d'importation

```
201. ImportDef ::= ImportKeyword ImportFromSpec (AllWithExcepts | (BeginChar
ImportSpec EndChar))
202. ImportKeyword ::= "import"
203. AllWithExcepts ::= AllKeyword [ExceptsDef]
204. ExceptsDef ::= ExceptKeyword BeginChar ExceptSpec EndChar
205. ExceptKeyword ::= "except"
206. 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 */
207. ExceptElement ::=  ExceptGroupSpec |
               ExceptTypeDefSpec |
               ExceptTemplateSpec |
               ExceptConstSpec |
               ExceptTestcaseSpec |
               ExceptAltstepSpec |
               ExceptFunctionSpec |
               ExceptSignatureSpec |
               ExceptModuleParSpec
208. ExceptGroupSpec ::= GroupKeyword (ExceptGroupRefList | AllKeyword)
209. ExceptTypeDefSpec ::= TypeDefKeyword (TypeRefList | AllKeyword)
210. ExceptTemplateSpec ::= TemplateKeyword (TemplateRefList | AllKeyword)
211. ExceptConstSpec ::= ConstKeyword (ConstRefList | AllKeyword)
212. ExceptTestcaseSpec ::= TestcaseKeyword (TestcaseRefList | AllKeyword)
213. ExceptAltstepSpec ::= AltstepKeyword (AltstepRefList | AllKeyword)
214. ExceptFunctionSpec ::= FunctionKeyword (FunctionRefList | AllKeyword)
215. ExceptSignatureSpec ::= SignatureKeyword  (SignatureRefList | AllKeyword)
216. ExceptModuleParSpec ::= ModuleParKeyword  (ModuleParRefList | AllKeyword)
217. ImportSpec ::=     {ImportElement [SemiColon]}
218. ImportElement ::=  ImportGroupSpec |
               ImportTypeDefSpec |
               ImportTemplateSpec |
               ImportConstSpec |
               ImportTestcaseSpec |
               ImportAltstepSpec |
               ImportFunctionSpec |
               ImportSignatureSpec |
               ImportModuleParSpec
219. ImportFromSpec ::= FromKeyword ModuleId [RecursiveKeyword]
220. ModuleId ::=  GlobalModuleId [LanguageSpec]
```

```
/* SÉMANTIQUE STATIQUE – La production LanguageSpec ne peut être omise que si le
module désigné contient de la notation TTCN-3 */
221. LanguageKeyword ::= "language"
222. LanguageSpec ::= LanguageKeyword  FreeText
223. GlobalModuleId ::= ModuleIdentifier [Dot ObjectIdentifierValue]
224. RecursiveKeyword ::= "recursive"
225. ImportGroupSpec ::= GroupKeyword (GroupRefList | AllGroupsWithExcept)
226. GroupRefList ::= FullGroupIdentifier {"," FullGroupIdentifier}
227. AllGroupsWithExcept ::= AllKeyword [ExceptKeyword GroupRefList]
228. FullGroupIdentifier := GroupIdentifier {Dot GroupIdentifier} [ExceptsDef]
229. ExceptGroupRefList ::= ExceptFullGroupIdentifier {","
ExceptFullGroupIdentifier}
230. ExceptFullGroupIdentifier ::= GroupIdentifier {Dot GroupIdentifier}
231. ImportTypeDefSpec ::= TypeDefKeyword (TypeRefList | AllTypesWithExcept)
232. TypeRefList ::= TypeDefIdentifier {"," TypeDefIdentifier}
233. AllTypesWithExcept ::= AllKeyword [ExceptKeyword TypeRefList]
234. TypeDefIdentifier ::=  StructTypeIdentifier |
               EnumTypeIdentifier |
               PortTypeIdentifier |
               ComponentTypeIdentifier |
               SubTypeIdentifier
235. ImportTemplateSpec ::= TemplateKeyword (TemplateRefList |
AllTemplsWithExcept)
236. TemplateRefList ::= TemplateIdentifier {"," TemplateIdentifier}
237. AllTemplsWithExcept ::= AllKeyword [ExceptKeyword TemplateRefList]
238. ImportConstSpec ::= ConstKeyword (ConstRefList | AllConstsWithExcept)
239. ConstRefList ::= ConstIdentifier {"," ConstIdentifier}
240. AllConstsWithExcept ::= AllKeyword [ExceptKeyword ConstRefList]
241. ImportAltstepSpec ::= AltstepKeyword (AltstepRefList |
AllAltstepsWithExcept)
242. AltstepRefList ::= AltstepIdentifier {"," AltstepIdentifier}
243. AllAltstepsWithExcept ::= AllKeyword [ExceptKeyword AltstepRefList]
244. ImportTestcaseSpec ::= TestcaseKeyword (TestcaseRefList |
AllTestcasesWithExcept)
245. TestcaseRefList ::= TestcaseIdentifier {"," TestcaseIdentifier}
246. AllTestcasesWithExcept ::= AllKeyword [ExceptKeyword TestcaseRefList]
247. ImportFunctionSpec ::= FunctionKeyword (FunctionRefList |
AllFunctionsWithExcept)
248. FunctionRefList ::= FunctionIdentifier {"," FunctionIdentifier}
249. AllFunctionsWithExcept ::= AllKeyword [ExceptKeyword FunctionRefList]
250. ImportSignatureSpec ::= SignatureKeyword  (SignatureRefList |
AllSignaturesWithExcept)
251. SignatureRefList ::= SignatureIdentifier {"," SignatureIdentifier}
252. AllSignaturesWithExcept ::= AllKeyword [ExceptKeyword SignatureRefList]
253. ImportModuleParSpec ::= ModuleParKeyword  (ModuleParRefList |
AllModuleParWithExcept)
254. ModuleParRefList ::= ModuleParIdentifier {"," ModuleParIdentifier}
255. AllModuleParWithExcept ::= AllKeyword [ExceptKeyword ModuleParRefList]
```

A.1.6.1.9 Définitions de groupe

```
256. GroupDef ::= GroupKeyword GroupIdentifier
               BeginChar
               [ModuleDefinitionsPart]
               EndChar
257. GroupKeyword ::= "group"
258. GroupIdentifier ::= Identifier
```

A.1.6.1.10 Définitions de fonction externe

```
259. ExtFunctionDef ::= ExtKeyword FunctionKeyword ExtFunctionIdentifier
"("[FunctionFormalParList] ")" [ReturnType]
260. ExtKeyword ::= "external"
261. ExtFunctionIdentifier ::= Identifier
```

A.1.6.1.11 Définitions de constante externe

```
262. ExtConstDef ::= ExtKeyword ConstKeyword Type ExtConstIdentifier
263. ExtConstIdentifier ::= Identifier
```

A.1.6.1.12 Définitions de paramètre de module

```
264. ModuleParDef ::= ModuleParKeyword "{" ModuleParList "}"
265. ModuleParKeyword ::= "modulepar"
266. ModuleParList ::= ModulePar {SemiColon ModulePar}
267. ModulePar ::= ModuleParType  ModuleParIdentifier [AssignmentChar
ConstantExpression] {"," ModuleParIdentifier [AssignmentChar
ConstantExpression]}
/* SÉMANTIQUE STATIQUE – La valeur de la production ConstantExpression doit être
du même type que celui qui est indiqué pour le paramètre */
268. ModuleParType ::= Type
269. ModuleParIdentifier ::= Identifier
```

A.1.6.2 Partie commande

A.1.6.2.0 Généralités

```
270. ModuleControlPart ::=  ControlKeyword
               BeginChar
               ModuleControlBody
               EndChar
               [WithStatement] [SemiColon]
271. ControlKeyword ::= "control"
272. ModuleControlBody ::= [ControlStatementOrDefList]
273. ControlStatementOrDefList ::=    {ControlStatementOrDef [SemiColon]}+
274. ControlStatementOrDef ::=   FunctionLocalInst |
               ControlStatement |
               FunctionLocalDef
275. ControlStatement ::=   TimerStatements |
               BasicStatements |
               BehaviourStatements |
               SUTStatements
```

A.1.6.2.1 Instanciation de variable

```
276. VarInstance ::= VarKeyword Type VarList
277. VarList ::= SingleVarInstance {"," SingleVarInstance}
278. SingleVarInstance ::= VarIdentifier [ArrayDef] [AssignmentChar
VarInitialValue]
279. VarInitialValue ::= Expression
280. VarKeyword ::= "var"
281. VarIdentifier ::= Identifier
282. VariableRef ::= (VarIdentifier | ValueParIdentifier)
[ExtendedFieldReference]
```

A.1.6.2.2 Timer instanciation

```
283. TimerInstance ::= TimerKeyword TimerList
284. TimerList ::= SingleTimerInstance{"," SingleTimerInstance}
285. SingleTimerInstance ::= TimerIdentifier [ArrayDef] [AssignmentChar
TimerValue]
286. TimerKeyword ::= "timer"
287. TimerIdentifier ::= Identifier
288. 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 attribution de valeur de
temporisation par défaut pour des séquences tabulaires de temporisations */
```

```
289. TimerRef ::= TimerIdentifier [ArrayOrBitRef]|
                  TimerParIdentifier [ArrayOrBitRef]
```

A.1.6.2.3 Opérations sur composant

```
290. ConfigurationStatements ::= ConnectStatement |
                  MapStatement |
                  DisconnectStatement |
                  UnmapStatement |
                  DoneStatement |
                  StartTCStatement |
                  StopTCStatement
291. ConfigurationOps ::= CreateOp | SelfOp | SystemOp | MTCOp | RunningOp
292. CreateOp ::= ComponentType Dot CreateKeyword
293. SystemOp ::= SystemKeyword
294. SelfOp ::= "self"
295. MTCOp ::= MTCKeyword
296. DoneStatement ::= ComponentId Dot DoneKeyword
297. ComponentId ::= ComponentIdentifier | (AnyKeyword | AllKeyword)
ComponentKeyword
298. DoneKeyword ::= "done"
299. RunningOp ::= ComponentId Dot RunningKeyword
300. RunningKeyword ::= "running"
301. CreateKeyword ::= "create"
302. ConnectStatement ::= ConnectKeyword PortSpec
303. ConnectKeyword ::= "connect"
304. PortSpec ::= "(" PortRef "," PortRef ")"
305. PortRef ::= ComponentRef Colon Port
306. ComponentRef ::= ComponentIdentifier | SystemOp | SelfOp | MTCOp
307. DisconnectStatement ::= DisconnectKeyword PortSpec
308. DisconnectKeyword ::= "disconnect"
309. MapStatement ::= MapKeyword PortSpec
310. MapKeyword ::= "map"
311. UnmapStatement ::= UnmapKeyword PortSpec
312. UnmapKeyword ::= "unmap"
313. StartTCStatement ::= ComponentIdentifier 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 */
314. StartKeyword ::= "start"
315. StopTCStatement ::= StopKeyword | ComponentIdentifier Dot StopKeyword |
                  AllKeyword ComponentKeyword Dot StopKeyword
316. ComponentIdentifier ::= VariableRef | FunctionInstance
/* SÉMANTIQUE STATIQUE - La variable associée à VariableRef ou le type de retour
associé à FunctionInstance doit impérativement être de type "component" */
```

A.1.6.2.4 Opérations sur les accès

```
317. Port ::= (PortIdentifier | PortParIdentifier) [ArrayOrBitRef]
318. CommunicationStatements ::= SendStatement |
                  CallStatement |
                  ReplyStatement |
                  RaiseStatement |
                  ReceiveStatement |
                  TriggerStatement |
                  GetCallStatement |
                  GetReplyStatement |
                  CatchStatement |
                  CheckStatement |
                  ClearStatement |
                  StartStatement |
```

```
               StopStatement
319. SendStatement ::= Port Dot PortSendOp
320. PortSendOp ::= SendOpKeyword "(" SendParameter ")" [ToClause]
321. SendOpKeyword ::= "send"
322. SendParameter ::= TemplateInstance
323. ToClause ::= ToKeyword AddressRef
324. ToKeyword ::= "to"
325. AddressRef ::= VariableRef | FunctionInstance
/* SÉMANTIQUE STATIQUE – Le retour des productions VariableRef et
FunctionInstance doit impérativement être de type "address" ou "component" */
326. CallStatement ::= Port Dot PortCallOp  [PortCallBody]
327. PortCallOp ::= CallOpKeyword "(" CallParameters ")" [ToClause]
328. CallOpKeyword ::= "call"
329. CallParameters ::= TemplateInstance ["," CallTimerValue]
/* SÉMANTIQUE STATIQUE - Seuls des paramètres de type "out" peuvent être omis ou
spécifiés avec un attribut apparié */
330. CallTimerValue ::= TimerValue | NowaitKeyword
/* SÉMANTIQUE STATIQUE - La production Value doit impérativement être de type
"float" */
331. NowaitKeyword ::= "nowait"
332. PortCallBody ::=   BeginChar
               CallBodyStatementList
               EndChar
333. CallBodyStatementList ::= {CallBodyStatement [SemiColon]}+
334. CallBodyStatement ::= CallBodyGuard  StatementBlock
335. CallBodyGuard ::= AltGuardChar CallBodyOps
336. CallBodyOps ::= GetReplyStatement | CatchStatement
337. ReplyStatement ::= Port Dot PortReplyOp
338. PortReplyOp ::= ReplyKeyword "(" TemplateInstance [ReplyValue]")"
[ToClause]
339. ReplyKeyword ::= "reply"
340. ReplyValue ::= ValueKeyword Expression
341. RaiseStatement ::= Port Dot PortRaiseOp
342. PortRaiseOp ::= RaiseKeyword "(" Signature "," TemplateInstance ")"
[ToClause]
343. RaiseKeyword ::= "raise"
344. ReceiveStatement ::= PortOrAny Dot PortReceiveOp
345. PortOrAny ::= Port | AnyKeyword PortKeyword
346. PortReceiveOp ::=  ReceiveOpKeyword ["(" ReceiveParameter ")"] [FromClause]
[PortRedirect]
/* SÉMANTIQUE STATIQUE - L'option PortRedirect ne peut être présente que si
l'option ReceiveParameter est également présente  */
347. ReceiveOpKeyword ::= "receive"
348. ReceiveParameter ::= TemplateInstance
349. From§ ::= FromKeyword AddressRef
350. FromKeyword ::= "from"
351. PortRedirect ::= PortRedirectSymbol (ValueSpec [SenderSpec] | SenderSpec)
352. PortRedirectSymbol ::= "→"
353. ValueSpec ::= ValueKeyword VariableRef
354. ValueKeyword ::= "value"
355. SenderSpec ::= SenderKeyword VariableRef
/* SEMANTIQUE STATIQUE - La variable ref doit impérativement être de type
"address" ou "component" */
356. SenderKeyword ::= "sender"
357. TriggerStatement ::= PortOrAny Dot PortTriggerOp
358. PortTriggerOp ::= TriggerOpKeyword ["(" ReceiveParameter ")"] [From§]
[PortRedirect]
/* SÉMANTIQUE STATIQUE: L'option PortRedirect ne peut être présente que si
l'option ReceiveParameter est également présente */
359. TriggerOpKeyword ::= "trigger"
360. GetCallStatement ::= PortOrAny  Dot PortGetCallOp
361. PortGetCallOp ::= GetCallOpKeyword ["(" ReceiveParameter ")"] [From§]
[PortRedirectWithParam]
```

```
/* SÉMANTIQUE STATIQUE: L'option PortRedirectWithParam ne peut être présente que
si l'option ReceiveParameter est également présente */
362. GetCallOpKeyword ::= "getcall"
363. PortRedirectWithParam ::= PortRedirectSymbol RedirectSpec
364. RedirectSpec ::= ValueSpec  [ParaSpec] [SenderSpec] |
               ParaSpec [SenderSpec] |
               SenderSpec
365. ParaSpec ::= ParaKeyword ParaAssignmentList
366. ParaKeyword ::= "param"
367. ParaAssignmentList ::= "(" (AssignmentList | VariableList) ")"
368. AssignmentList ::= VariableAssignment {"," VariableAssignment}
369. VariableAssignment ::= VariableRef AssignmentChar ParameterIdentifier
/* SÉMANTIQUE STATIQUE: La production parameterIdentifier doit impérativement
être extraite de la définition de signature correspondante */
370. ParameterIdentifier ::= ValueParIdentifier |
               TimerParIdentifier |
               TemplateParIdentifier |
               PortParIdentifier
371. VariableList ::= VariableEntry  {"," VariableEntry}
372. VariableEntry ::= VariableRef | NotUsedSymbol
373. GetReplyStatement ::= PortOrAny Dot PortGetReplyOp
374. PortGetReplyOp ::=     GetReplyOpKeyword ["(" ReceiveParameter
[ValueMatchSpec] ")"]
               [From§] [PortRedirectWithParam]
/* SÉMANTIQUE STATIQUE: L'option PortRedirectWithParam ne peut être présente que
si l'option ReceiveParameter est également présente */
375. GetReplyOpKeyword ::= "getreply"
376. ValueMatchSpec ::= ValueKeyword TemplateInstance
377. CheckStatement ::= PortOrAny Dot PortCheckOp
378. PortCheckOp ::= CheckOpKeyword ["(" CheckParameter ")"]
379. CheckOpKeyword ::= "check"
380. CheckParameter ::=     CheckPortOpsPresent | FromClausePresent |
RedirectPresent
381. From§Present ::= FromClause [PortRedirectSymbol SenderSpec]
382. RedirectPresent ::= PortRedirectSymbol SenderSpec
383. CheckPortOpsPresent ::= PortReceiveOp | PortGetCallOp | PortGetReplyOp |
PortCatchOp
384. CatchStatement ::= PortOrAny Dot PortCatchOp
385. PortCatchOp ::= CatchOpKeyword ["("CatchOpParameter ")"] [FromClause]
[PortRedirect]
/* SÉMANTIQUE STATIQUE: L'option PortRedirect ne peut être présente que si
l'option CatchOpParameter est également présente */
386. CatchOpKeyword ::= "catch"
387. CatchOpParameter ::= Signature "," TemplateInstance | TimeoutKeyword
388. ClearStatement ::= PortOrAll Dot PortClearOp
389. PortOrAll ::= Port | AllKeyword PortKeyword
390. PortClearOp ::= ClearOpKeyword
391. ClearOpKeyword ::= "clear"
392. StartStatement ::= PortOrAll Dot PortStartOp
393. PortStartOp ::= StartKeyword
394. StopStatement ::= PortOrAll Dot PortStopOp
395. PortStopOp ::= StopKeyword
396. StopKeyword ::= "stop"
397. AnyKeyword ::= "any"
```

A.1.6.2.5 Opérations de temporisation

```
398. TimerStatements ::=    StartTimerStatement | StopTimerStatement |
TimeoutStatement
399. TimerOps ::= ReadTimerOp | RunningTimerOp
400. StartTimerStatement ::= TimerRef Dot StartKeyword ["(" TimerValue ")"]
401. StopTimerStatement ::= TimerRefOrAll Dot StopKeyword
402. TimerRefOrAll ::= TimerRef | AllKeyword TimerKeyword
403. ReadTimerOp ::= TimerRef Dot ReadKeyword
```

```
404. ReadKeyword ::= "read"
405. RunningTimerOp ::= TimerRefOrAny  Dot RunningKeyword
406. TimeoutStatement ::= TimerRefOrAny  Dot TimeoutKeyword
407. TimerRefOrAny ::= TimerRef | AnyKeyword TimerKeyword
408. TimeoutKeyword ::= "timeout"
```

A.1.6.3 Type

```
409. Type ::= PredefinedType | ReferencedType
410. PredefinedType ::= BitStringKeyword |
                BooleanKeyword |
                CharStringKeyword |
                UniversalCharString |
                CharKeyword |
                UniversalChar |
                IntegerKeyword |
                OctetStringKeyword |
                ObjectIdentifierKeyword |
                HexStringKeyword |
                VerdictTypeKeyword |
                FloatKeyword |
                AddressKeyword |
                DefaultKeyword |
                AnyTypeKeyword
411. BitStringKeyword ::= "bitstring"
412. BooleanKeyword ::= "boolean"
413. IntegerKeyword ::= "integer"
414. OctetStringKeyword ::= "octetstring"
415. ObjectIdentifierKeyword ::= "objid"
416. HexStringKeyword ::= "hexstring"
417. VerdictTypeKeyword ::= "verdicttype"
418. FloatKeyword ::= "float"
419. AddressKeyword ::= "address"
420. DefaultKeyword ::= "default"
421. AnyTypeKeyword ::= "anytype"
422. CharStringKeyword ::= "charstring"
423. UniversalCharString ::= UniversalKeyword CharStringKeyword
424. UniversalKeyword ::= "universal"
425. CharKeyword ::= "char"
426. UniversalChar ::= UniversalKeyword CharKeyword
427. ReferencedType ::= [GlobalModuleId Dot] TypeReference
[ExtendedFieldReference]
428. TypeReference ::= StructTypeIdentifier[TypeActualParList] |
                EnumTypeIdentifier |
                SubTypeIdentifier |
                ComponentTypeIdentifier
429. TypeActualParList ::= "(" TypeActualPar {"," TypeActualPar} ")"
430. TypeActualPar ::= ConstantExpression
431. ArrayDef ::= {"[" ArrayBounds [".." ArrayBounds] "]"}+
432. ArrayBounds ::= SingleConstExpression
/* SÉMANTIQUE STATIQUE – ArrayBounds se réduira à une valeur non négative de
type entier (integer) */
```

A.1.6.4 Valeur

```
433. Value ::= PredefinedValue | ReferencedValue
434. PredefinedValue ::= BitStringValue |
                BooleanValue |
                CharStringValue |
                IntegerValue |
                OctetStringValue |
                ObjectIdentifierValue |
                HexStringValue |
                VerdictTypeValue |
```

```
                EnumeratedValue |
                FloatValue |
                AddressValue |
                OmitValue
435. BitStringValue ::= Bstring
436. BooleanValue ::= "true" | "false"
437. IntegerValue ::= Number
438. OctetStringValue ::= Ostring
439. ObjectIdentifierValue ::= ObjectIdentifierKeyword  "{" ObjIdComponentList
"}"
/* La production ReferencedValue doit impérativement être de type "Object
Identifier" */
440. ObjIdComponentList ::= {ObjIdComponent}+
441. ObjIdComponent ::= NameForm |
                NumberForm |
                NameAndNumberForm
442. NumberForm ::= Number | ReferencedValue
/* SÉMANTIQUE STATIQUE – ReferencedValue doit impérativement être de type
"integer" et avoir une valeur non négative */
443. NameAndNumberForm ::= Identifier "(" NumberForm ")"
444. NameForm ::= Identifier
445. HexStringValue ::= Hstring
446. VerdictTypeValue ::= "pass" | "fail" | "inconc" | "none" | "error"
447. EnumeratedValue ::= EnumerationIdentifier
448. CharStringValue ::= Cstring | Quadruple
449. Quadruple ::= CharKeyword "(" Group "," Plane "," Row "," Cell ")"
450. Group ::= Number
451. Plane ::= Number
452. Row ::= Number
453. Cell ::= Number
454. FloatValue ::= FloatDotNotation | FloatENotation
455. FloatDotNotation ::= Number Dot DecimalNumber
456. FloatENotation ::= Number [Dot DecimalNumber] Exponential [Minus] Number
457. Exponential ::= "E"
458. ReferencedValue ::= ValueReference [ExtendedFieldReference]
459. ValueReference ::=     [GlobalModuleId Dot] (ConstIdentifier |
ExtConstIdentifier) |
                ValueParIdentifier |
                ModuleParIdentifier |
                VarIdentifier
460. Number ::= (NonZeroNum {Num}) | "0"
461. NonZeroNum ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
462. DecimalNumber ::= {Num}+
463. Num ::= "0" | NonZeroNum
464. Bstring ::= "'" {Bin} "'" "B"
465. Bin ::= "0" | "1"
466. Hstring ::= "'" {Hex} "'" "H"
467. Hex ::= Num | "A" | "B" | "C" | "D" | "E" | "F"| "a" | "b" | "c" | "d" |
"e" | "f"
468. Ostring ::= "'" {Oct} "'" "O"
469. Oct ::= Hex Hex
470. Cstring ::= """ {Char} """
471. 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 l'ISO/CEI 646. Pour le type
UniversalCharstring, il s'agit d'un caractère extrait de tout jeu de caractères
défini dans l'ISO/CEI 10646 */
472. Identifier ::= Alpha{AlphaNum | Underscore}
473. Alpha ::= UpperAlpha | LowerAlpha
474. AlphaNum ::= Alpha | Num
475. 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"
```

```
476. 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"
477. 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 (0,0,0,33) 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) */
478. FreeText ::= """ {ExtendedAlphaNum} """
479. AddressValue ::= "null"
480. OmitValue ::= OmitKeyword
481. OmitKeyword ::= "omit"
```

A.1.6.5 Paramétrage

```
482. InParKeyword ::= "in"
483. OutParKeyword ::= "out"
484. InOutParKeyword ::= "inout"
485. FormalValuePar ::= [(InParKeyword | InOutParKeyword | OutParKeyword)] Type
ValueParIdentifier
486. ValueParIdentifier ::= Identifier
487. FormalPortPar ::= [InOutParKeyword] PortTypeIdentifier PortParIdentifier
488. PortParIdentifier ::= Identifier
489. FormalTimerPar ::= [InOutParKeyword] TimerKeyword TimerParIdentifier
490. TimerParIdentifier ::= Identifier
491. FormalTemplatePar ::= [InParKeyword] TemplateKeyword Type
TemplateParIdentifier
492. TemplateParIdentifier ::= Identifier
```

A.1.6.6 Instruction "with"

```
493. WithStatement ::= WithKeyword WithAttribList
494. WithKeyword ::= "with"
495. WithAttribList ::= "{" MultiWithAttrib "}"
496. MultiWithAttrib ::= {SingleWithAttrib [SemiColon]}+
497. SingleWithAttrib ::= AttribKeyword  [OverrideKeyword] [AttribQualifier]
AttribSpec
498. AttribKeyword ::=  EncodeKeyword |
              VariationKeyword |
              DisplayKeyword |
              ExtensionKeyword
499. EncodeKeyword ::= "encode"
500. VariationKeyword ::= "variant"
501. DisplayKeyword ::= "display"
502. ExtensionKeyword ::= "extension"
503. OverrideKeyword ::= "override"
504. AttribQualifier ::= "(" DefOrFieldRefList ")"
505. DefOrFieldRefList ::= DefOrFieldRef {"," DefOrFieldRef}
506. DefOrFieldRef ::= DefinitionRef | FieldReference | AllRef | PredefinedType
/* 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 */
507. DefinitionRef ::=  StructTypeIdentifier |
              EnumTypeIdentifier |
              PortTypeIdentifier |
              ComponentTypeIdentifier |
              SubTypeIdentifier |
              ConstIdentifier |
              TemplateIdentifier |
              AltstepIdentifier |
              TestcaseIdentifier |
              FunctionIdentifier |
              SignatureIdentifier |
                VarIdentifier |
```

```
                 TimerIdentifier |
                   PortIdentifier |
                   ModuleParIdentifier |
                 FullGroupIdentifier
508. AllRef ::= ( GroupKeyword AllKeyword [ExceptKeyword BeginChar GroupRefList
EndChar])              |
                ( TypeDefKeyword AllKeyword [ExceptKeyword BeginChar
TypeRefList] EndChar)
|
                ( TemplateKeyword AllKeyword [ExceptKeyword BeginChar
TemplateRefList] EndChar)
|
                ( ConstKeyword AllKeyword [ExceptKeyword BeginChar ConstRefList]
EndChar)
|
                ( AltstepKeyword AllKeyword [ExceptKeyword BeginChar
AltstepRefList] EndChar)
|
                ( TestcaseKeyword AllKeyword [ExceptKeyword BeginChar
TestcaseRefList] EndChar) |
                ( FunctionKeyword AllKeyword [ExceptKeyword BeginChar
FunctionRefList] EndChar) |
                ( SignatureKeyword AllKeyword [ExceptKeyword BeginChar
SignatureRefList] EndChar) |
                ( ModuleParKeyword AllKeyword [ExceptKeyword BeginChar
ModuleParRefList] EndChar)

509. AttribSpec ::= FreeText
```

A.1.6.7 Instructions comportementales

```
510. BehaviourStatements ::=     TestcaseInstance |
               FunctionInstance |
               ReturnStatement |
               AltConstruct |
               InterleavedConstruct |
               LabelStatement |
               GotoStatement |
               RepeatStatement |
                 DeactivateStatement |
               AltstepInstance
/* 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 */
511. VerdictStatements ::= SetLocalVerdict
512. VerdictOps ::= GetLocalVerdict
513. 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
attribuer l'erreur sur la valeur */
514. SetVerdictKeyword ::= "setverdict"
515. GetLocalVerdict ::= "getverdict"
516. SUTStatements ::= ActionKeyword "(" (FreeText | TemplateRefWithParList)
")"
517. ActionKeyword ::= "action"
518. ReturnStatement ::= ReturnKeyword  [Expression]
519. AltConstruct ::= AltKeyword BeginChar AltGuardList EndChar
520. AltKeyword ::= "alt"
521. AltGuardList ::= {GuardStatement [SemiColon]}+  [ElseStatement [SemiColon]]
```

```
522. GuardStatement ::= AltGuardChar (AltstepInstance | GuardOp  StatementBlock)
523. ElseStatement ::= "["ElseKeyword "]" StatementBlock
524. AltGuardChar ::= "[" [BooleanExpression] "]"
525. GuardOp ::=   TimeoutStatement |
                   ReceiveStatement |
                   TriggerStatement |
                   GetCallStatement |
                   CatchStatement |
                   CheckStatement |
                   GetReplyStatement |
                   DoneStatement
/* SÉMANTIQUE STATIQUE – La production GuardOp est utilisée dans la partie d'un
module relative à la commande. Elle ne peut contenir que l'instruction
timeoutStatement */
526. InterleavedConstruct ::= InterleavedKeyword BeginChar InterleavedGuardList
EndChar
527. InterleavedKeyword ::= "interleave"
528. InterleavedGuardList ::= {InterleavedGuardElement [SemiColon]}+
529. InterleavedGuardElement ::= InterleavedGuard InterleavedAction
530. InterleavedGuard ::= "[" "]" GuardOp
531. 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 */
532. LabelStatement ::= LabelKeyword LabelIdentifier
533. LabelKeyword ::= "label"
534. LabelIdentifier ::= Identifier
535. GotoStatement ::= GotoKeyword LabelIdentifier
536. GotoKeyword ::= "goto"
537. RepeatStatement ::= "repeat"
538. ActivateOp ::= ActivateKeyword "(" AltstepInstance ")"
539. ActivateKeyword ::= "activate"
540. DeactivateStatement ::= DeactivateKeyword ["(" Expression ")"]
/* SÉMANTIQUE STATIQUE - Cette expression doit prendre une valeur de type par
défaut */
541. DeactivateKeyword ::= "deactivate"
```

A.1.6.8 Instructions de base

```
542. BasicStatements ::= Assignment | LogStatement | LoopConstruct |
ConditionalConstruct
543. 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 */
544. 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". */
545. FieldExpressionList ::= "{" FieldExpressionSpec {"," FieldExpressionSpec}
"}"
546. FieldExpressionSpec ::= FieldReference AssignmentChar Expression
547. ArrayExpression ::= "{" [ArrayElementExpressionList] "}"
548. ArrayElementExpressionList ::= NotUsedOrExpression  {","
NotUsedOrExpression}
549. NotUsedOrExpression ::= Expression | NotUsedSymbol
550. ConstantExpression ::= SingleConstExpression | CompoundConstExpression
551. 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*/
552. BooleanExpression ::= SingleExpression
/* SÉMANTIQUE STATIQUE – BooleanExpression doit correspondre à une valeur de
type Boolean */
```

```
553. 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". */
554. FieldConstExpressionList ::= "{" FieldConstExpressionSpec {","
FieldConstExpressionSpec} "}"
555. FieldConstExpressionSpec ::= FieldReference AssignmentChar
ConstantExpression
556. ArrayConstExpression ::= "{" [ArrayElementConstExpressionList] "}"
557. ArrayElementConstExpressionList ::= ConstantExpression {","
ConstantExpression}
558. Assignment ::= VariableRef  AssignmentChar Expression
/* SÉMANTIQUE OPÉRATIONNELLE – L'expression située à droite de la production
Assignment doit prendre une valeur explicite du même type que le côté gauche. */
559. SingleExpression ::= SimpleExpression {LogicalOp SimpleExpression}
/* SÉMANTIQUE OPÉRATIONNELLE – Si les deux productions SimpleExpressions et
LogicalOp existent, alors la production SimpleExpressions doit prendre des
valeurs spécifiques de types compatibles */
560. SimpleExpression ::= ["not"] SubExpression
/* SÉMANTIQUE OPÉRATIONNELLE – Les opérandes de l'opérateur NOT doivent être de
type booléen (TTCN ou ASN.1) ou des dérivés de type booléen. */
561. SubExpression ::= PartialExpression [RelOp PartialExpression]
/* SÉMANTIQUE OPÉRATIONNELLE – Si les deux productions PartialExpressions et
RelOp existent, alors la production PartialExpressions doit prendre des valeurs
spécifiques de types compatibles. */
/* SÉMANTIQUE OPÉRATIONNELLE – Si l'opérateur RelOp est "<" | ">" | ">=" | "<=",
alors chaque production SubExpression doit prendre la valeur d'un type
spécifique: integer, enumerated ou float (ces valeurs peuvent être en notation
TTCN ou ASN.1) */
562. PartialExpression ::= Result [ShiftOp Result]
/* SÉMANTIQUE OPÉRATIONNELLE – Chaque résultat doit correspondre à une valeur
spécifique. Si plus d'un seul résultat existe, l'opérande de droite doit être de
type integer. Si l'opérateur de décalage est '<<' ou '>>', alors l'opérande de
gauche doit correspondre au type bitstring, hexstring ou octetstring. Si
l'opérateur de décalage est '<@' ou '@>', alors l'opérande de gauche doit être
de type bitstring, hexstring, charstring ou universal charstring */
563. Result ::= SubResult {BitOp SubResult}
/* SÉMANTIQUE OPÉRATIONNELLE – Si les deux productions SubResults et BitOp
existent, alors la production SubResults doit prendre des valeurs spécifiques de
types compatibles */
564. SubResult ::= ["not4b"] Product
/* SÉMANTIQUE OPÉRATIONNELLE – Si l'opérateur not4b existe, l'opérande doit être
de type bitstring, octetstring ou hexstring. */
565. Product ::= Term {AddOp Term}
/* SÉMANTIQUE OPÉRATIONNELLE – Chaque terme doit correspondre à une valeur
spécifique. Si plus d'un seul terme existe et si l'opérateur d'addition
correspond à StringOp, alors les termes doivent correspondre à un même type qui
doit être: bitstring, hexstring, octetstring, charstring ou universal
charstring. Si plus d'un seul terme existe et que l'opérateur d'additon ne
corresponde pas à StringOp, alors les termes doivent tous les deux correspondre
à un type integer ou float.  */
566. Term ::= Factor {MultiplyOp Factor}
/* SÉMANTIQUE OPÉRATIONNELLE – Chaque production Factor doit correspondre à une
valeur spécifique. Si plus d'une seule production Factor existe, alors ces
productions Factor doivent correspondre à un type integer ou float. */
567. Factor ::= [UnaryOp] Primary
/* SÉMANTIQUE OPÉRATIONNELLE – La production Primary doit correspondre à une
valeur spécifique. Si la production UnaryOp existe et est "not", alors Primary
doit correspondre au type BOOLEAN. Si la production UnaryOp est "+" ou "-",
alors Primary doit correspondre à un type integer ou float. Si la production
UnaryOp correspond à l'opérateur not4b, alors la production Primary doit
correspondre au type bitstring, hexstring ou octetstring. */
568. Primary ::= OpCall | Value | "(" SingleExpression ")"
```

```
569. ExtendedFieldReference ::= {(Dot ( StructFieldIdentifier | ArrayOrBitRef |
TypeDefIdentifier)) | ArrayOrBitRef }+
/* SEMANTIQUE OPERATIONNELLE: 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.
570. OpCall ::=    ConfigurationOps |
               VerdictOps |
               TimerOps |
               TestcaseInstance |
               FunctionInstance |
               TemplateOps |
               ActivateOp
571. AddOp ::= "+" | "-" | StringOp
/* SÉMANTIQUE OPÉRATIONNELLE – Les opérandes des opérateurs "+" ou "-" doivent
être de type integer ou float (c'est-à-dire être prédéfinis en notation TTCN ou
ASN.1) ou être des dérivés du type integer ou float (c'est-à-dire être une sous-
étendue) */
572. MultiplyOp ::= "*" | "/" | "mod" | "rem"
/* SÉMANTIQUE OPÉRATIONNELLE – Les opérandes des opérateurs "*", "/", rem ou mod
doivent être de type integer ou float (c'est-à-dire être prédéfinis en notation
TTCN ou ASN.1) ou être des dérivés du type integer ou float (c'est-à-dire être
une sous-étendue). */
573. UnaryOp ::= "+" | "-"
/* SÉMANTIQUE OPÉRATIONNELLE – Les opérandes des opérateurs "+" ou "-" doivent
être de type integer ou float(c'est-à-dire être prédéfinis en notation TTCN ou
ASN.1) ou être des dérivés du type integer ou float (c'est-à-dire être une sous-
étendue). */
574. RelOp ::= "==" | "<" | ">" | "!=" | ">=" | "<="
/* SÉMANTIQUE OPÉRATIONNELLE – La priorité des opérateurs est définie dans le
Tableau 7. */
575. BitOp ::= "and4b" | "xor4b" | "or4b"
/* SÉMANTIQUE OPÉRATIONNELLE – Les opérandes de l'opérateur and4b, or4b ou xor4b
doivent être de type bitstring, hexstring ou octetstring (TTCN ou ASN.1) ou des
dérivés de ces types. */
576. LogicalOp ::= "and" | "xor" | "or"
/* SÉMANTIQUE OPÉRATIONNELLE – Les opérandes des opérateurs and, or ou xor
doivent être de type booléen (TTCN ou ASN.1) ou des dérivés de type booléen. */
/* SÉMANTIQUE OPÉRATIONNELLE – La priorité des opérateurs est définie dans le
Tableau 7 */
577. StringOp ::= "&"
/* SÉMANTIQUE OPÉRATIONNELLE – Les opérandes des opérateurs de chaîne doivent
être de type bitstring, hexstring, octetstring ou character string */
578. ShiftOp ::= "<<" | ">>" | "<@" | "@>"
579. LogStatement ::= LogKeyword "(" [FreeText] ")"
580. LogKeyword ::= "log"
581. LoopConstruct ::= ForStatement |
               WhileStatement |
               DoWhileStatement
582. ForStatement ::= ForKeyword "(" Initial SemiColon Final SemiColon Step ")"
               StatementBlock
583. ForKeyword ::= "for"
584. Initial ::= VarInstance | Assignment
585. Final ::= BooleanExpression
586. Step ::= Assignment
587. WhileStatement ::=     WhileKeyword "(" BooleanExpression ")"
               StatementBlock
588. WhileKeyword ::= "while"
589. DoWhileStatement ::= DoKeyword StatementBlock
               WhileKeyword "(" BooleanExpression ")"
590. DoKeyword ::= "do"
591. ConditionalConstruct ::=    IfKeyword "(" BooleanExpression ")"
               StatementBlock
               {ElseIfClause}[ElseClause]
592. IfKeyword ::= "if"
```

```
593. ElseIfClause ::= ElseKeyword  IfKeyword "(" BooleanExpression ")"
StatementBlock
594. ElseKeyword ::= "else"
595. ElseClause ::= ElseKeyword  StatementBlock
```

A.1.6.9 Productions diverses

```
596. Dot ::= "."
597. Dash ::= "-"
598. Minus ::= Dash
599. SemiColon ::= ";"
600. Colon ::= ":"
601. Underscore ::= "_"
602. BeginChar ::= "{"
603. EndChar ::= "}"
604. 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 valeurs spécifiques pourraient être
template MyMessageType MyTemplate:=
{
     field1 := 3+2,      // valeur spécifique de type entier
     field2 := "Ma chaîne",  // valeur spécifique de type chaîne
                             // de caractères
     field3 := true,         // valeur spécifique de type booléen
     field4 := {1,2,3}       // valeur spécifique of séquence tabulaire
                             // 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 spécifient énumère 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)        // appariement si 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 s'évalue comme un élément particulier 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 une 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 s'évalue comme 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 s'évalue comme un entier ou un nombre en virgule flottante spécifique.

La limite inférieure doit être mise à gauche de l'étendue, la limite supérieure du côté droit. 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 les limites sont utilisées dans des modèles ou dans des champs de modèle de type **char, universal char, 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 (p. ex. 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 surensemble 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, array.

B.1.3.1 Elément quelconque

Le symbole d'appariement "?" (*AnyElement*) sert à indiquer qu'il remplace des éléments particuliers d'une chaîne (sauf les chaînes de caractères) ou d'un type **record of, set of** ou array. Il ne doit être utilisé qu'à l'intérieur de valeurs de type **string**, **record of**, **set of**, array.

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, comme une valeur d'entier, ou comme un élément quelconque à l'intérieur d'un type **record of**, **set of, array**. Etant donné que les deux interprétations conduisent au 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 array. 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'un type **string, record of**, **set of,** array, il doit être interprété comme une structure séquentielle *AnyElementsOrNone*.

NOTE – Cette règle empêche l'interprétation (sinon possible) de "*" en tant que structure séquentielle *AnyValueOrNone* qui remplace un élément à l'intérieur d'un type **string, record of**, **set of, array**.

B.1.3.2.1 Utilisation de structures génériques à caractères multiples

S'il est nécessaire d'exprimer 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 séquentielle "abc*xyz".

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 valeur de chaîne et le nombre d'éléments contenus dans une structure séquentielle de type **set of, record of, array.** Il ne doit être utilisé que comme attribut des mécanismes suivants: Complément, *AnyValue*, *AnyValueOrNone*, *AnyElement* et *AnyElementsOrNone*. Il peut également être utilisé conjointement avec l'attribut **ifpresent**. La syntaxe de l'attribut **length** peut être trouvée dans les § 6.2.3 et 6.3.3.

Les unités de longueur doivent être interprétées conformément au Tableau 4 dans le cas des valeurs de chaîne. Pour les structures de type **set of, record of, array**, 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'un 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.

Dans le cas d'un champ omis, l'attribut de longueur est toujours considéré comme apparié (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) length (1 .. 6),
     // revient à (1,2,3,6)
     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 ? et * afin d'indiquer respectivement un caractère quelconque et un nombre quelconque de caractères quelconques.

Exemple 1:

```
template charstring MyTemplate:= pattern "ab??xyz*";
```

Ce modèle s'appariera à 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 tout caractères.

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.

Tableau B.1/Z.140 – Liste des métacaractères des structures TTCN-3

Métacaractère	Description
?	S'apparie avec tout caractère
*	S'apparie avec tout caractère zéro ou plusieurs fois
\	Provoque l'interprétation du métacaractère suivant comme un littéral (voir Note)
[ ]	S'apparie avec tout caractère contenu dans l'ensemble spécifié. Voir § B.1.5.1 pour plus de détails
-	Utilisable seulement à l'intérieur d'une paire de crochets ("[" et "]") et permet de spécifier une étendue de caractères. Voir § B.1.5.1 pour plus de détails
^	Utilisable 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 spécifiés par le quadruplet
{référence}	Insère la chaîne définie par l'utilisateur qui est indiquée et l'interprète comme une expressions régulière. Voir § B.1.5.2 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 du jeu C0 (voir ISO/CEI 6429 [13])
\n	S'apparie avec le caractère de commande LF du jeu C0 (voir ISO/CEI 6429 [13])
\r	S'apparie avec le caractère de commande CR du jeu C0 (voir ISO/CEI 6429 [13])
\"	S'apparie avec le caractère de guillemet droit
|	Utilisé afin d'indiquer un choix entre deux expressions
( )	Utilisé afin de grouper une expression
#(n, m)	S'apparie avec l'expression précédente au moins n fois mais au plus m fois. Voir § B.1.5.3 pour plus de détails
NOTE – Par conséquent, le caractère de barre oblique inverse peut être apparié par une paire de caractères de barre oblique inverse sans espace entre eux (\\).	

B.1.5.1 Expression d'un ensemble

L'expression d'un ensemble est délimitée par les symboles '[' ']'. En plus des caractère littéraux, il est possible de spécifier des séries de caractères au moyen du séparateur '-'. L'expression d'un ensemble peut également être rendue négative par insertion du caractère '^' comme premier caractère après le crochet d'ouverture.

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"[A-E][0-9][0-9][0-9]YKE";

// RegExp3 s'appariera avec une chaîne qui commence avec une lettre comprise
//entre A et E, 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 l'énoncé de structure séquentielle, des références à des modèles, constantes ou variables existants. La référence est englobée dans les caractères '{' '}'. La référence doit se réduire à un des types de chaîne de caractères.

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 de caractères quelconques. En effet, toute chaîne de caractères suivant le mot clé **pattern**, soit explicitement ou par référence, sera interprétée 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 de caractères quelconques, suivis par les caractères 'de', suivis par le caractère contenu dans l'ISO/CEI 10646-1 avec groupe=1, plan=1, rangée=65 et cellule=7.

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, la syntaxe '#(n, m)' est utilisée. Elle spécifie que l'expression précédente doit impérativement être appariée au moins n fois mais pas plus de m fois.

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 RegExp5:= pattern"[a-z]#(9)";      // correspond à
                                                       // exactement 9
                                                       // 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
```

Annexe C

Fonctions de notation TTCN-3 prédéfinies

La présente annexe définit les fonctions TTCN-3 prédéfinies.

C.1 Conversion d'entier en caractère

```
int2char(integer value) return char
```

Cette fonction convertit une valeur de type **integer** dans l'étendue de 0 … 127 (codage sur 8 éléments binaires) en valeur de caractère de la Rec. UIT-T T.50 [5]. La valeur d'entier décrit le codage sur 8 éléments binaires du caractère.

La fonction retourne –1 si la valeur de l'argument est un nombre négatif ou supérieur à 127.

C.2 Conversion de caractère en entier

```
char2int(char value) return integer
```

Cette fonction convertit une valeur de type **char** de la Rec. UIT-T T.50 [5] en valeur d'entier dans l'étendue de 0 … 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 char
```

Cette fonction convertit une valeur de type **integer** dans l'étendue de 0 … 2 147 483 647 (codage sur 32 éléments binaires) en valeur de caractère de l'ISO/CEI 10646 [6]. La valeur d'entier décrit le codage sur 32 éléments binaires du caractère.

La fonction retourne le quadruplet (255, 255, 255, 255) si la valeur de l'argument est un nombre négatif ou supérieur à:

2 147 483 647

C.4 Conversion de caractère universel en entier

```
unichar2int(universal char value) return integer
```

Cette fonction convertit une valeur de type **universal char** de l'ISO/CEI 10646 [6] en valeur d'entier dans l'étendue de 0 … 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 poids faible, le bit de gauche est de poids fort. 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 poids faible, le chiffre hexadécimal de gauche est de poids fort. Les chiffres hexadécimaux 0 .. F représentent respectivement les valeurs décimales 0 .. 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 poids faible, le chiffre hexadécimal de gauche est de poids fort. 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 .. F représentent respectivement les valeurs décimales 0 .. 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**. Si la chaîne ne représente pas une valeur valide d'entier, la fonction retourne la valeur zéro (0).

Exemples:

```
str2int("66")      // will return the integer value 66

str2int("-66")     // will return the integer value -66

str2int("abc")     // will return the integer value 0

str2int("0")       // will return the integer value 0
```

C.9 Conversion d'entier en chaîne binaire

```
int2bit(integer value, 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 poids faible, le bit de gauche est de poids fort. Les bits 0 et 1 représentent respectivement les valeurs décimales 0 et 1 . Si la conversion 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. Une erreur de test élémentaire doit apparaître si la valeur est négative ou si le type **bitstring** résultant contient plus de bits que spécifié dans le paramètre `length`.

C.10 Conversion d'entier en chaîne hexadécimale

```
int2hex(integer value, 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 poids faible, le chiffre hexadécimal de gauche est de poids fort. Les chiffres hexadécimaux 0 … F représentent respectivement les valeurs décimales 0 … 15. Si la conversion 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. Une erreur de test élémentaire doit apparaître si la valeur est négative ou si le type résultant **hexstring** contient plus de chiffres hexadécimaux que spécifié dans le paramètre `length` .

C.11 Conversion d'entier en chaîne d'octets

```
int2oct(integer value, 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 poids faible, le chiffre hexadécimal de gauche est de poids fort. 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 .. F représentent respectivement les valeurs décimales 0 .. 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. Une erreur de test élémentaire doit apparaître si la valeur **value** est négative ou si le type résultant **hexstring** contient plus de chiffres hexadécimaux que spécifié dans le paramètre `length` .

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).

Exemples:

```
int2str(66)        // will return the charstring value "66"

int2str(-66)       // will return the charstring value "-66"

int2str(0)              // will return the integer value "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 [6], 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 un type structuré

```
sizeof(structured_type value) return integer
```

Cette fonction retourne le nombre déclaré d'éléments d'un type **record, record of**, **set, set of** ou le nombre effectif d'éléments d'une constante, d'une variable, d'un modèle (**template**) de ces types ou d'une séquence tabulaire (voir Note). Cette fonction ne doit pas être appliquée aux types **record of** ou **set of** sans sous-typage de longueur. Dans le cas de valeurs de type **record of** et **set of** ou de modèles ou de séquences tabulaires, 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
     };
type record of integer MyPDU1;

template MyPDU  MyTemplate
     { field1  omit,
       field2  5
     };

var integer numElements;

// alors
numElements := sizeof(MyPDU);         // retourne 2
numElements := sizeof(MyTemplate);    // retourne 1
numElements := sizeof(MyPDU1);
// retourne une erreur car MyPDU1 n'est pas contraint


// si l'on a
     type record length(0..10) of integer MyRecord;
     var MyRecord 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 retourne la valeur **true** si et seulement si la valeur du champ désigné est présent dans l'instance effective de l'objet de données désigné. L'argument de la fonction **ispresent** doit être une référence à un champ contenu dans un objet de données qui est défini comme étant de type **optional**.

```
// 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 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 séquentielle 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 séquentielle 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 **charstring**. 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 **charstring**. 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) retournera "1110101"

C.19 Conversion de chaîne hexadécimale en chaîne de caractères

hex2str (hextring 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) retournera "AB801"

C.20 Conversion de chaîne d'octets en chaîne de caractères

oct2str (octetstring value**) return charstring**

Cette fonction convertit une valeur de type **octetstring** en valeur de type **charstring**. Le type résultant **charstring** aura la même longueur que le type entrant **octetstring**. Les octets sont interprétés comme des codes de la Rec. UIT-T T.50 [5] (conformément à la version internationale de référence) et les caractères résultants sont mémorisés dans la valeur retournée. Les valeurs d'octet supérieures à 7F doivent provoquer une erreur.

Exemple:

oct2str ('4469707379'H) = "Dipsy"

NOTE – La chaîne de caractères retournée peut contenir des caractères non graphiques, qui ne peuvent pas être présentés entre les guillemets droits.

C.21 Conversion de chaîne de caractères en chaîne d'octets

str2oct (**charstring** value) **return octetstring**

Cette fonction convertit une valeur de type **charstring** en valeur de type **octetstring**. Le type résultant **octetstring** aura la même longueur que le type entrant **charstring**. Chaque octet du type **octetstring** contiendra les codes de la Rec. UIT-T T.50 [5] (conformément à la version internationale de référence) des caractères appropriés du type **charstring**.

Exemple:

```
str2oct ("Tinky-Winky") = '54696E6B792D57696E6B79'H
```

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 devrait 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 (hextring 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** devrait 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 (hextring 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** devrait 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** devrait 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)='000111010111'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(limitesupérieure – limiteinférieure +1)*rnd()) +
limiteinférieure
// 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, integer index, 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 paramère 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 nom est JJ", 11, 2)  // retourne "JJ"
```

Annexe D

Utilisation d'autres types de données avec la notation TTCN-3

D.1 Utilisation de la notation ASN.1 avec la notation TTCN-3

La présente annexe définit l'utilisation facultative de la notation ASN.1 avec la notation TTCN-3.

D.1.0 Généralités

La notation TTCN-3 fournit une interface claire pour l'utilisation, dans des modules TTCN-3, de la version ASN.1 2002 (telle que définie dans la série de Recommandations UIT-T X.680 [7], [8], [9], [10]). Après importation dans un module TTCN-3, l'identificateur linguistique doit être:

– "ASN.1:2002" pour la version ASN.1 2002;

– "ASN.1:1997" pour la version ASN.1 1997;

– "ASN.1:1994" pour la version ASN.1 1994;

– "ASN.1:1988" pour la version du Livre bleu de l'ASN.1.

NOTE 1 – Les identificateurs linguistiques "ASN.1:1997", "ASN.1:1994" et "ASN.1:1988" se rapportent à des versions ASN.1 fondées sur des Recommandations UIT-T remplacées et la seule finalité de leur inclusion dans la présente Recommandation est d'affecter des identificateur uniques si des modules de protocole fondés sur ces versions ASN.1 sont utilisés avec la notation TTCN-3.

NOTE 2 – Quand la version "ASN.1:1988" est prise en charge, les items ASN.1 doivent être importés conformément aux règles syntaxiques et sémantiques de la Rec. UIT-T X.208 (Livre Bleu).

NOTE 3 – Les références relatives aux versions ASN.1:1997, ASN.1:1994 et ASN.1:1988 peuvent être trouvées dans l'Annexe F.

Quand la notation ASN.1 est utilisé avec la notation TTCN-3, les mots clés énumérés au § 11.18/X.680 [7] ne doivent pas être utilisés comme identificateurs dans un module TTCN-3. Les mots clés ASN.1 doivent suivre les exigences de la Rec. UIT-T X.680 [7].

D.1.1 Equivalences entre types ASN.1 et TTCN-3

D.1.1.0 Généralités

Les types ASN.1 énumérés dans le Tableau D.1 sont considérés comme équivalents à leurs contreparties TTCN-3.

Tableau D.1/Z.140 – Liste des équivalents ASN.1 et TTCN-3

Le type ASN.1	s'applique à l'équivalent TTCN-3
BOOLEAN	**boolean**
INTEGER	**integer**
REAL (Note 1)	**float**
OBJECT IDENTIFIER	**objid**
BIT STRING	**bitstring**
OCTET STRING	**octetstring**
SEQUENCE	**record**
SEQUENCE OF	**record of**
SET	**set**
SET OF	**set of**
ENUMERATED	**enumerated**

Tableau D.1/Z.140 – Liste des équivalents ASN.1 et TTCN-3

Le type ASN.1	s'applique à l'équivalent TTCN-3
CHOICE	**union**
VisibleString	**char** (Note 2)**, charstring**
IA5String	**char** (Note 2)**, charstring**
UniversalString	**universal char** (Note 2)**,** **universal charstring**
NOTE 1 – Le type ASN.1 REAL est équivalent au type TTCN-3 **float** jusqu'à ce que la base soit illimitée ou limitée à la base 10 explicitement ou implicitement. La notation ASN.1 permet une restriction explicite, par exemple par un sous-typage interne; mais du point de vue de l'affectation des types ASN.1-TTCN-3, une restriction explicite est une notation ASN.1 de valeur. Une restriction implicite peut être définie par la description textuelle du protocole indiqué, c'est-à-dire à l'extérieur de module(s) ASN.1. Cependant, dans les deux cas, la notation TTCN-3 de valeur peut être utilisée sans avoir à considérer si la base est ASN.1 (voir également la Note 3 dans le § D.1.2.0). NOTE 2 – Seuls les sous-types ASN.1 d'une longueur d'exactement 1 caractère sont équivalent aux types de caractères TTCN-3 de base; p. ex., IA5String (SIZE (1)) est équivalent au type TTCN-3 char mais IA5String (SIZE (0..1)) ne l'est pas.	

En notation TTCN-3, tous les opérateurs, toutes les fonctions, tous les mécanismes d'appariement, toutes les notations de valeur etc. qui peuvent être utilisés avec un type TTCN-3 figurant dans le Tableau D.1 peuvent également être utilisés avec le type ASN.1 correspondant.

D.1.1.1 Identificateurs

Lors de la conversion d'identificateurs ASN.1 en identificateurs TTCN-3, tous les symboles de tiret "-" doivent être remplacés par un soulignement "_".

Exemple:

```
MyASN1module DEFINITIONS ::=
BEGIN
         Missleading-ASN1-Name::=     INTEGER    -- Identificateur de type ASN.1
                                                 -- utilisant un tiret "-"

END


module MyTTCNModule
{
         import from MyASN1module language "ASN.1:2002" all;

         const Missleading_ASN1_Name ExampleConst:= 1;      // Référence TTCN-3
                                                            // à un type ASN.1
                                                            // utilisant des
                                                            // soulignements
}
```

D.1.2 Types et valeurs de données ASN.1

D.1.2.0 Généralités

Les types et valeurs ASN.1 peuvent être utilisés dans des modules TTCN-3. Les définitions ASN.1 sont effectuées au moyen d'un module ASN.1 distinct. Les types et valeurs ASN.1 sont désignés par leurs références de type et de valeur, produites conformément aux § 9.3/X.680 [7] et 9.4/X.680 [7] dans le ou les modules ASN.1.

Exemple 1:

```
MyASN1module DEFINITIONS ::=
BEGIN
          Z::= INTEGER              -- Simple définition de type


          BMessage::= SET             -- Définition ASN.1 de type
          {
               name      Name,
               title     VisibleString,
               date      Date
          }

          johnValues Bmessage ::=     -- Définition de valeur ASN.1
          {
               name "John Doe",
               title     "Mr",
               date      "April 12th"
          }

          DefinedValuesForField1 Z ::= {0 | 1} -- Définition ASN.1 de sous-type
END
```

Le module ASN.1 doit être conforme à la syntaxe des Recommandations UIT-T de la série X.680 [7], [8], [9] et [10]. Une fois déclarés, les types et valeurs ASN.1 peuvent être utilisés dans des modules TTCN-3 de la même façon que les types et valeurs TTCN-3 issus d'autres modules TTCN-3 sont utilisés (c'est-à-dire que les définitions requises doivent être importées). Lors de l'importation d'items ASN.1 dans un module TTCN-3, un type ou valeur associé est produit pour chaque item ASN.1 importé. Toutes les définitions ou affectations TTCN-3 fondées sur des items ASN.1 importés doivent être effectuées conformément aux règles imposées par le type ou la valeur associé. Egalement, le mécanisme d'appariement doit utiliser le type associé lors de l'appariement de constantes, variables, modèles ou expressions en ligne fondées sur des déclarations ASN.1.

Les types et valeurs associés sont issus d'items ASN.1 par application des règles de transformation suivantes (l'ordre correspond à celui de l'exécution des transformations individuelles):

1) négliger d'éventuels marqueurs d'extension et spécifications d'exception;
2) négliger d'éventuelles contraintes définies par l'utilisateur (voir § 9/X.682 [9]);
3) négliger d'éventuelles contraintes de contenu (voir § 9/X.682 [9]);
4) négliger d'éventuelles contraintes de structure séquentielle (voir § 48.9/X.680 [7]);
5) créer des sous-types équivalents à partir de tous les types contraints par un sous-typage englobé en remplaçant les types inclus par l'ensemble de valeurs qu'ils représentent;
6) exécuter la transformation COMPONENTS OF conformément au § 24.4/X.680 [7] sur tout type SEQUENCE et conformément au § 26.2/X.680 [7] sur tout type SET contenant les mots clés " COMPONENTS OF";
7) remplacer tout type EMBEDDED PDV par son type associé obtenu par expansion de sous-typage interne dans le type associé au type EMBEDDED PDV (voir § 32.5/X.680 [7]) afin d'obtenir une définition complète de type;
8) remplacer le type EXTERNAL par son type associé, obtenu par expansion de sous-typage interne dans le type associé au type EXTERNAL (voir § 33.5/X.680 [7]) afin d'obtenir une définition complète de type (voir Note 3);
9) remplacer le type CHARACTER STRING par son type associé obtenu par expansion de sous-typage interne dans le type associé au type CHARACTER STRING (voir § 39.5/X.680 [7]) afin d'obtenir une définition complète de type;

10) remplacer le type INSTANCE OF par son type associé obtenu en remplaçant la classe DefinedObjectClass INSTANCE OF par son type ASN.1 associé (voir § C.7/X.681 [8]) et remplacer tous les types ASN.1 par leurs TTCN-3 équivalents conformément au Tableau D.1. Le type résultant est le type TTCN-3 associé;

11) négliger un éventuel sous-typage interne restant (voir Note 4);

12) négliger d'éventuels nombres nommés et bits nommés dans des types ASN.1. Dans les valeurs ASN.1, remplacer tout nombre nommé par sa valeur et remplacer tout bit nommé ou toute séquence de bits nommés par une chaîne binaire sans zéros à droite, où les positions binaires identifiées par des noms présents sont remplacées par des "1" et où les autres positions binaires sont remplacées par des "0";

13) remplacer tout type sélection par le type désigné par ce type sélection; si le type choix indiqué (le "Type" dans le § 29.1/X.680 [7]) est un type contraint, la sélection doit être effectuée sur le type parent du type choix indiqué;

14) convertit tout type ou valeur RELATIVE-OID en un type ou en une valeur **`objid`** (voir Note 5);

15) remplacer l'un quelconque des types chaîne de caractères restreints ci-après par son type associé, obtenu comme suit (voir Note 6):

    BMPString: **`universal charstring`** (char (0,0,0,0) .. char (0,0,255,255));

    UTF8String: **`universal charstring;`**

    NumericString: **`charstring`** contraint à l'ensemble de caractères indiqué dans le § 36.2/X.680 [7];

    PrintableString: **`charstring`** contraint à l'ensemble de caractères indiqué dans le § 36.4/X.680 [7];

    TeletexString et T61String: **`universal charstring`** contraint à l'ensemble de caractères indiqué dans la Rec. UIT-T T.61 (voir bibliographie);

    VideotexString: **`universal charstring`** contraint à l'ensemble de caractères indiqué dans les Recommandations UIT-T T.100 [14] et T.101 [15];

    GraphicString: **`universal charstring;`**

    GeneralString: **`universal charstring;`**

16) remplacer d'éventuels types ou valeurs GeneralizedTime et UTCTime par le type ou la valeur **`charstring`**;

17) remplacer d'éventuels types ou valeurs ObjectDescriptor par le type ou la valeur **`universal charstring`**;

18) remplacer d'éventuelles notations pour les types de champ de classe d'objets (voir § 14/X.681 [8]) par l'item ASN.1 auquel elles font référence; les types ouverts doivent être remplacés par le métatype "OPEN TYPE" aux fins de la transformation (et seulement à cette fin);

19) remplacer toutes les informations issues de notations d'objet (voir § 15/X.681 [8]) par l'item ASN.1 auquel elles font référence;

20) réduire les contraintes tabulaires (voir § 10/X.682 [9]) à un sous-typage énuméré et négliger d'éventuelles contraintes relationnelles (voir Note 7);

21) remplacer toutes les occurrences du type NULL par le type TTCN-3 associé ci-après: **`type enumerated`** <*identificateur*> { NULL }, où <*identificateur*> est la référence ASN.1 de type convertie conformément au § D.1.1.1;

22) remplacer toutes les références à des types ouverts par **`anytype`**;

23) remplacer les types ASN.1 par leurs équivalents conformément au Tableau D.1 et remplacer les valeurs ASN.1 par les valeurs TTCN-3 équivalentes d'après les types associés (voir Note 8). Le métatype "OPEN TYPE" doit être remplacé par **anytype**.

NOTE 1 – Les seuls types associés ne permettent pas le codage correct des valeurs fondées sur les types ASN.1. Cependant, les informations complémentaires qui sont requises par le système afin d'effectuer un codage correct dépendent de l'implémentation et restent cachées pour l'utilisateur. Elle ne sont pas requises afin d'effectuer des déclarations ou affectations correctes sur la base des types et valeurs ASN.1.

NOTE 2 – Lors de l'importation de types ENUMERATED, les nombres entiers affectés par l'utilisateur à des énumérations sont également importés.

NOTE 3 – Le champ de valeur de données du type EXTERNAL peut être codé comme un type ASN1 isolé, être aligné au niveau des octets ou être arbitraire (voir § 8.18.1/X.690 [11]) à la discrétion du codeur. Si l'utilisateur souhaite appliquer une certaine forme de codage ou ne permettre qu'une seule forme de codage spécifique lors de l'appariement, il doit utiliser l'attribut de codage approprié au type ou à la constante, à la variable, au modèle ou au champ de modèle en cause (voir § D.1.5.2).

NOTE 4 – Le sous-typage interne doit être pris en considération par l'utilisateur lors de la définition de valeurs ou modèles TTCN-3 fondés sur un type ASN.1 contraint par sous-typage interne.

NOTE 5 – L'équivalence avec le type **objid** est limitée à la seule syntaxe à utiliser pour les notations de valeur. Dans le cas de valeurs de type **objid,** les deux premières valeurs de l'arborescence sont limitées (voir la Rec. UIT-T X.660 [16]). Cette restriction ne s'applique pas aux valeurs fondées sur le type importé RELATIVE-OID.

NOTE 6 – Les types VisibleString, IA5String et UniversalString ont leur type TTCN-3 équivalent et sont remplacés directement.

NOTE 7 – Les contraintes relationnelles doivent être prises en considération par l'utilisateur lors de la déclaration de valeurs et de modèles (qui peuvent également être manipulés implicitement par des utilitaires).

NOTE 8 – Les champs facultatifs faisant défaut dans des valeurs de types structurés ASN.1 (SET, SEQUENCE, EXTERNAL, etc.) sont équivalents aux champs explicitement omis dans les valeurs structurées de la notation TTCN-3.

Exemple 2:

```
module MyTTCNModule
{
          import from MyASN1module language "ASN.1:2002" all;

          const Bmessage MyTTCNConst:= johnValues;
          const DefinedValuesForField1 Value1:= 1;
}
```

NOTE 9 – Les définitions ASN.1 autres que de types et valeurs (c'est-à-dire les classes ou ensembles d'objets informationnels) ne sont pas directement accessibles à partir de la notation TTCN-3. De telles définitions doivent être réduites à un type ou à une valeur se trouvant dans le module ASN.1 avant de pouvoir être désigné à partir de l'intérieur du module TTCN-3.

D.1.2.1 Portée des identificateurs ASN.1

Les identificateurs ASN.1 importés suivent les mêmes règles de portée que les types et valeurs TTCN-3 importés (voir § 5.3).

D.1.3 Paramétrage en notation ASN.1

Il est permis de faire référence à des définitions paramétrées de type et valeur ASN.1 à partir du module TTCN-3. Cependant, toutes les définitions paramétrées ASN.1 utilisées dans un module TTCN-3 doivent être fournies avec des paramètres effectifs (les types ouverts ne sont pas autorisés) et les paramètres effectifs ainsi fournis doivent pouvoir être résolus au moment de la compilation.

Le langage noyau de la notation TTCN-3 ne prend pas en charge l'importation d'items ASN.1, qui font appel uniquement à des objets ASN.1 spécifiques en tant que paramètre(s) formel(s) ou

effectif(s). Le paramétrage propre à la notation ASN.1, qui implique des objets qui ne peuvent pas être définis directement dans le langage noyau de la notation TTCN-3, doit donc être résolu dans la partie ASN.1 avant d'être utilisé dans la notation TTCN-3. Les objets propres à la notation ASN.1 sont les suivants:

a) classes d'objets informationnels;

b) objets informationnels;

c) ensembles d'objets informationnels.

Par exemple, la notation qui suit n'est pas légale parce qu'elle définit un type TTCN-3 qui prend un objet ASN.1 en tant que paramètre effectif.

```
MyASN1module DEFINITIONS ::=
BEGIN
     -- Définition de module ASN.1

     -- Information object class definition
     MESSAGE ::= CLASSE { &msgTypeValue    INTEGER UNIQUE,
                                           &MsgFields}

     -- Information object definition
     setupMessage MESSAGE ::= {   &msgTypeValue   1,
                                  &MsgFields      OCTET STRING}

     setupAckMessage MESSAGE ::= {    &msgTypeValue    2,
                                      &MsgFields  BOOLEAN}

     -- Définition d'ensemble d'objets informationnels
     MyProtocol MESSAGE ::= { setupMessage | setupAckMessage}

     -- Type ASN.1 contraint par un ensemble d'objets
     MyMessage{ MESSAGE : MsgSet} ::= SÉQUENCE
     {
          code      MESSAGE.&msgTypeValue({ MsgSet}),
          Type  MESSAGE.&MsgFields({ MsgSet})
}
END

module MyTTCNModule
{
     // Définition de module TTCN-3
     import from MyASN1module langage "ASN.1:2002" all;

     // Type TTCN-3 illégal avec l'ensemble d'objets comme paramètre
     type record Q(MESSAGE MyMsgSet) ::= {Z                  field1,
                                          MyMessage(MyMsgSet)  field2}
}
```

Afin de rendre cette définition légale, le type ASN.1 complémentaire My Message1 doit être défini comme représenté ci-dessous. Cette notation résout le paramétrage d'ensemble d'objets informationnels et peut donc être directement utilisée dans le module TTCN-3.

```
MyASN1module DEFINITIONS ::=
BEGIN
     -- Définition de module ASN.1

     …

     MyProtocol MESSAGE ::= { setupMessage | setupAckMessage}

     -- Type ASN.1 complémentaire afin de supprimer le paramétrage
     -- d'ensemble d'objets
```

```
     MyMessage1 ::= MyMessage{ MyProtocol}
END

module MyTTCNModule
{
     // Définition de module TTCN-3
     import from MyASN1module language "ASN.1:2002" all;

     // Type TTCN-3 légal sans ensemble d'objets comme paramètre
     type record Q := { Z                  field1,
                        MyMessage1         field2}
}
```

D.1.4 Définition des modèles de message ASN.1

D.1.4.0 Généralités

Si des messages sont définis en notation ASN.1 au moyen, par exemple, d'un type SEQUENCE (ou éventuellement SET) alors les messages effectifs, pour les deux événements **send** et **receive**, peuvent être spécifiés au moyen de la syntaxe de valeur ASN.1.

Exemple:

```
MyASN1module DEFINITIONS ::=
BEGIN
     -- Définition de module ASN.1

     -- Définition du message
MyMessageType::= SEQUENCE
     {    field1    [1]  IA5String,         // comme le type TTCN-3
                                            // character string
          field2    [2]  INTEGER OPTIONAL,  // comme le type TTCN-3 integer
          field3    [4]  Field3Type,        // comme le type TTCN-3 record
          field4    [5]  Field4Type         // comme le type TTCN-3 array
     }

  Field3Type::= SEQUENCE {field31 BIT STRING, field32 INTEGER, field33
  OCTET STRING},
     Field4Type::= SEQUENCE OF BOOLEAN


     -- peut avoir la valeur suivante
     myValue MyMessageType::=
     {
          field1        "Une chaîne",
          field2        123,
          field3        {field31 '11011'B, field32 456789, field33 'FF'O},
          field4        {true, false}
     }
END
```

D.1.4.1 Messages de réception ASN.1 utilisant la syntaxe de modèle TTCN-3

Les mécanismes d'appariement ne sont pas pris en charge dans la syntaxe ASN.1 normale. Donc, si l'on souhaite utiliser des mécanismes d'appariement avec un message de réception ASN.1, alors la syntaxe TTCN-3 pour les modèles de réception doit plutôt être utilisée. Noter que cette syntaxe comprend des références de composant afin de pouvoir faire référence aux composants individuels qui sont contenus dans des types ASN.1 SEQUENCE, SET, etc.

Exemple:

```
import from MyASN1module language "ASN.1:2002" {
     type myMessageType
}
```

```
// Un modèle de message utilisant des mécanismes d'appariement en notation
// TTCN-3 pourrait être
template myMessageType  MyValue:=
{
     field1 :=          "A"<?>"tr"<*>"g",
     field2 :=          *,
     field3.field31 :=  '110??'B,
     field3.field32 :=  ?,
     field3.field33 :=  'F?'O,
     field4.[0] :=      true,
     field4.[1] :=      false
}

// la syntaxe suivante est également valide
template myMessageType  MyValue:=
{
     field1 := "A"<?>"tr"<*>"g",      // chaîne avec structures génériques
     field2 := *,                     // tout entier ou aucun que ce soit
     field3 := {'110??'B, ?, 'F?'O},
     field4 := {?, false}
}
```

D.1.4.2 Séquencement des champs de modèle

Quand des modèles TTCN-3 servent à des types ASN.1, la signification de l'ordre des champs dans le modèle dépendra du type de construction ASN.1 utilisée afin de définir le type de message. Par exemple: si le type `SEQUENCE` ou `SEQUENCE OF` est utilisé, alors les champs de message doivent être envoyés ou appariés dans l'ordre spécifié dans le modèle. Si le type `SET` ou `SET OF` est utilisé, alors les champs de message peuvent être envoyés ou appariés dans un ordre quelconque.

D.1.5 Informations de codage

D.1.5.0 Généralités

La notation TTCN-3 permet d'associer à divers éléments linguistiques TTCN-3 des références à des règles de codage et à des variantes situées à l'intérieur de règles de codage. Il est également possible de définir des codages invalides. Ces informations de codage sont spécifiées au moyen de l'instruction **`with`** conformément à la syntaxe suivante:

Exemple:

```
module MyModule
{
      :
     import from MyASN1module language "ASN.1:2002" {
          type myMessageType
     }
     with {
          encode:= "PER-BASIC-ALIGNED:2002"
          // Toutes les instances de MyMessageType devrontt être codées au
          // moyen des règles PER:2002
     }
      :
} // fin du module
with { encode "BER:2002" } // Le codage par défaut pour le module entier
                           // (suite de tests) est BER:2002
```

D.1.5.1 Attributs de codage ASN.1

Les chaînes suivantes sont les attributs de codage prédéfinis (normalisés) pour la notation ASN.1:

a) "BER:2002" signifie: codé conformément à La Rec. UIT-T X.690 (BER) [11];

b) "CER:2002" signifie: codé conformément à La Rec. UIT-T X.690 (CER) [11];

c) "DER:2002" signifie: codé conformément à La Rec. UIT-T X.690 (DER) [11].

d) "PER-BASIC-UNALIGNED:2002" signifie: codé conformément à la Rec. UIT-T X.691 [12] (règles PER non alignées);

e) "PER-BASIC-ALIGNED:2002" signifie: codé conformément à la Rec. UIT-T X.691 (règles PER alignées) [12];

f) "PER-CANONICAL-UNALIGNED:2002" signifie: codé conformément à la Rec. UIT-T X.691 [12] (règles PER canoniques non alignées);

g) "PER-CANONICAL-ALIGNED:2002" signifie: codé conformément à la Rec. UIT-T X.691 (règles PER canoniques alignées) [12].

D.1.5.2 Attributs ASN.1 de type "variant"

Les chaînes suivantes sont des attributs prédéfinis (normalisés) de type "variant". Ils n'ont une signification prédéfinie que quand ils sont appliqués conjointement avec des attributs ASN.1 de codage prédéfinis (voir § D.1.5.1). La manipulation de ces attributs prédéfinis quand ils sont appliqués conjointement avec d'autres attributs ou appliqués à un objet TTCN-3 sans attribut est hors du domaine d'application de la présente Recommandation (voir Note 1):

a) "forme de longueur 1" signifie que la valeur indiquée ne doit être codée et décodée qu'au moyen de la forme courte des octets de longueur (voir § 8.1.3/X.690 [11]) dans le cas de codages selon les règles BER, CER et DER ou qu'au moyen du déterminant de longueur d'un seul octet (voir § 10.9/X.691 [12]) dans le cas d'une forme quelconque du codage par règles PER (voir Note 2).

b) "forme de longueur 2" signifie que la valeur indiquée ne doit être codée et décodée qu'au moyen de la forme longue des octets de longueur (voir § 8.1.3/X.690 [11]) dans le cas de codages selon les règles BER, CER et DER ou qu'au moyen du déterminant de longueur de deux octets (voir § 10.9/X.691 [12]) dans le cas d'une forme quelconque du codage PER (voir Note 2).

c) "forme de longueur 3" signifie que la valeur indiquée ne doit être codée et décodée qu'au moyen de la forme indéfinie des octets de longueur (voir § 8.1.3/X.690 [11]) dans le cas de codages BER, CER et DER.

d) "REAL base 2" signifie que la valeur indiquée doit être codée ou appariée conformément à la forme de codage binaire REAL. Cet attribut ne peut être utilisé que sur des constantes, des variables ou des modèles. Quand il est appliqué à une sorte quelconque de groupage (p. ex. à des groupes ou à toute l'instruction d'importation), cet attribut ne doit avoir d'effet que sur ces objets TTCN-3.

e) Les expressions "type ASN1 isolé", "aligné au niveau des octets" et "arbitraire" signifient que la valeur indiquée d'après un type ASN.1 externe doit être codée au moyen de la forme de codage spécifiée par l'attribut ou doit être appariée si elle n'est reçue qu'avec le choix spécifié (voir § 8.18/X.690 [11]). Cet attribut ne peut être appliqué qu'à des types EXTERNAL, à des constantes, à des variables, à des modèles ou à des champs de modèle fondés sur ces types, importés de la notation ASN.1; quand cet attribut est appliqué à une sorte quelconque de groupage (p. ex. à groupes ou à toute l'instruction d'importation), il ne doit avoir d'effet que sur ces objets TTCN-3. Si les conditions fixées dans les § 8.18.6/X.690 à 8.18.8/X.690 [11] et l'attribut spécifié ne correspondent pas, cela doit provoquer une erreur d'exécution.

f) "TeletexString" signifie que la valeur indiquée doit être codée et décodée comme le type ASN.1 TeletexString (voir § 8.20/X.690 [11] et 26/X.691 [12]).

g) "VideotexString" signifie que la valeur indiquée doit être codée et décodée comme le type ASN.1 VideotexString (voir § 8.20/X.690 [11] et 26/X.691 [12]).

h) "GraphicString" signifie que la valeur indiquée doit être codée et décodée comme le type ASN.1 GraphicString (voir § 8.20/X.690 [11] et 26/X.691 [12]).

i) "GeneralString" signifie que la valeur indiquée doit être codée et décodée comme le type ASN.1 GeneralString (voir § 8.20/X.690 [11] et 26/X.691 [12]).

NOTE 1 – Ces attributs peuvent être réutilisés dans des règles de codage propres à l'implémentation avec une signification différente de celle qui est spécifié dans le présent paragraphe, peuvent être négligés ou une indication d'avertissement/d'erreur peut être donnée. Cependant, la stratégie à appliquer dépend de l'implémentation.

NOTE 2 – L'application de ces attributs de type "variant" peut conduire à un codage ASN.1 non valide (p. ex. si l'on applique la forme indéfinie de longueur à des valeurs de primitive dans les règles BER ou si l'on n'utilise pas le nombre minimal nécessaire d'octets de longueur). Cela est autorisé intentionnellement et les utilisateurs doivent attribuer avec précaution ces attributs de type "variant" à des constantes, des variables, des modèles ou des champs de modèle utilisés pour la réception.

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 contenus 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 –32768 à 32767 pour le type signé et de 0 à 65535 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 –2147483648 à 2147483647 pour le type signé et de 0 à 4294967295 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 –9223372036854775808 à 9223372036854775807 pour le type signé et de 0 à 18446744073709551615 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 Annexe F) 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 (p. ex. 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 [6]. 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 [6]. 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 (p. ex. 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 [6]). 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 [6]). Ses valeurs distinctives sont zéro, un, ou plusieurs caractères extraits de ce jeu. Les valeurs de ce type doivent entièrement (p. ex. 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 [6]. 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 Annexe F). Ses valeurs distinctives sont zéro, un, ou plusieurs caractères extraits du jeu de caractères ISO/CEI 8859. Les valeurs de ce type doivent entièrement (p. ex. 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 ISO/CEI 646. 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 tout alphabet 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.

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 Annexe F). 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 appelée "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" };
```

Annexe F (informative)

Bibliographie

- Recommandation UIT-T T.61 (1988), *Répertoire de caractères et jeux de caractères codés pour le service international télétex*.
- Recommandation UIT-T X.208 (1988), *Spécification de la syntaxe abstraite numéro un (ASN.1)* (Statut: supprimée).
- Recommandation UIT-T X.680 (1994) | ISO/CEI 8824-1:1995, *Technologies de l'information – Notation de syntaxe abstraite numéro un: spécification de la notation de base* (Statut: remplacée).
- Recommandation UIT-T X.681 (1994) | ISO/CEI 8824-2:1995, *Technologies de l'information – Notation de syntaxe abstraite numéro un: spécification des objets informationnels* (Statut: remplacée).
- Recommandation UIT-T X.682 (1994) | ISO/CEI 8824-3:1995, *Technologies de l'information – Notation de syntaxe abstraite numéro un: spécification des contraintes* (Statut: remplacée).
- Recommandation UIT-T X.683 (1994) | ISO/CEI 8824-4:1995, *Technologies de l'information – Notation de syntaxe abstraite numéro un: paramétrage des spécifications de la notation de syntaxe abstraite numéro un* (Statut: remplacée).
- Recommandation UIT-T X.680 (1997) | ISO/CEI 8824-1:1998, *Technologies de l'information – Notation de syntaxe abstraite numéro un: spécification de notation de base* (Statut: remplacée).
- Recommandation UIT-T X.681 (1997) | ISO/CEI 8824-2:1998, *Technologies de l'information – Notation de syntaxe abstraite numéro un: spécification des objets informationnels* (Statut: remplacée).
- Recommandation UIT-T X.682 (1997) | ISO/CEI 8824-3:1998, *Technologies de l'information – Notation de syntaxe abstraite numéro un: spécification des contraintes* (Statut: remplacée).
- Recommandation UIT-T X.683 (1997) | ISO/CEI 8824-4:1998, *Technologies de l'information – Notation de syntaxe abstraite numéro un: paramétrage des spécifications de la notation de syntaxe abstraite numéro un* (Statut: remplacée).
- ETSI ES 201 873-4 (V2.2.1), *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 4: TTCN-3 Operational Semantics (MTS)*.
- IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmeti*c.
- ISO/CEI 8859-1:1998, *Technologies de l'information– Jeux de caractères graphiques codés sur un seul octet – Partie 1: Alphabet latin n° 1*.

– OMG Document formal/2001-12-01, *The Common Object Request Broker: Architecture and Specification, Section 3, IDL Syntax and Semantics*, Version 2.6.

Annexe G (informative)

Utilisation d'expressions régulières et de mécanismes d'appariement en notation TTCN-3

La présente annexe vise à offrir des directives additionnelles sur l'utilisation d'expressions régulières et de mécanismes d'appariement comme défini dans la présente Recommandation et dans la Rec. UIT-T Z.141, Version 3 de la notation combinée arborescente et tabulaire: format de présentation tabulaire (TFT, *tabular presentation format*).

Les notations TTCN-3 et ASN.1 utilisent toutes les deux des expressions régulières et mécanismes d'appariement qui peuvent servir à spécifier ou à vérifier des valeurs. Les mécanismes d'appariement de la notation TTCN-3 servent à remplacer des valeurs de champs individuels ou à remplacer tout le contenu d'un modèle.

Les mécanismes d'appariement pris en charge sont représentés dans le Tableau 6 avec leurs symboles associés et leur domaine d'application. Ce tableau énumère tous les types TTCN-3 et ASN.1 équivalents tels que définis dans les Recommandations UIT-T de la série X.680, auxquels ces mécanismes d'appariement s'appliquent. Une description complète de chaque mécanisme d'appariement peut être trouvée dans le § B.1.5.

Il reste quelques différences entre la façon dont les mécanismes d'appariement sont utilisés en notation TTCN-3 et en notation ASN.1. Cela est dû à une légère divergence au cours de la longue période de mise au point des deux normes. Il n'y a cependant aucune contradiction fondamentale entre elles.

Les explications complémentaires ci-après visent à préciser l'utilisation des expressions régulières en notation TTCN-3 lorsque des différences avec la notation ASN.1 ne peuvent pas être évitées.

G.1 Est-ce que le caractère qui suit le symbole "\" est un métacaractère?

Un caractère qui suit le symbole "\" n'est pas autorisé en tant que métacaractère. Le comportement n'est pas explicitement défini. Cette combinaison devrait donc se traduire par une erreur. La raison en est la rétrocompatibilité. Par exemple, si "\a" est introduit en remplacement de "a", la suite de tests écrite antérieurement va avoir un comportement différent.

G.2 Est-ce que l'astérisque "*" correspond à la plus courte ou à la plus longue séquence de caractères possible?

La question est inappropriée. L'astérisque "*" s'apparie à un nombre quelconque de caractères. L'appariement à une structure séquentielle donne un résultat booléen. Mais le résultat sera déterminé par la partie restante de la séquence, non par l'astérisque lui-même. Un seul "*" s'apparie à une chaîne quelconque. Noter que "*" a un signification sémantique différente en notation TTCN-3 par rapport à la notation ASN.1. Pour plus de détails et d'explications, voir le § G.10. L'appariement de la structure séquentielle "abc*xyz" produira la valeur "Vrai" pour toute séquence de caractères commençant par les caractères "abc" et se terminant par les caractères "xyz". Ce qui se trouve entre ces deux séquences n'a pas d'intérêt. Pour un nombre quelconque d'itérations du trois premiers et des trois derniers caractères, le résultat sera vrai. Dans un autre exemple, la séquence "abc*xyz*" s'appariera également à la séquence "abcxyzxyz", de deux façons différentes: c'est soit le premier soit le second astérisque qui s'appariera avec un nombre de caractères égal à zéro.

G.3 Est-ce que les métacaractères "?" et "*" correspondent à des fins de ligne?

Les notations ASN.1 et TTCN-3 utilisent différemment les métacaractères "?" et "*". Voir § G.10.

G.4 Quel est le comportement des métacaractères non produits par échappement mais "illégaux"? Est-ce que "ab]" correspond à a-b-right-crochet-droit?

Il n'y a pas de métacaractères "illégaux". Une PAIRE de crochets est définie de façon à avoir une signification spécifique. Tout caractère qui n'est pas défini comme ayant une signification spécifique est juste un caractère littéral. Un seul crochet "[" ou "]" dans une séquence de caractères ne possède donc aucune signification spécifique et est juste un caractère. Par exemple, "ab]" signifie a-b-crochet-droit. Si une paire de crochets est requise comme caractères littéraux, ces crochets sont écrits sous la forme "\[ab]".

G.5 Si foo := "ab", est-ce que la structure séquentielle "{foo}#(2)" correspond à abb ou abab?

Ces expressions ne devraient pas être comparées. Il n'y a aucune chaîne désignée en ASN.1 et aucun caractère (ou jeu de caractères) nommé en notation TTCN-3. Essayer de les comparer n'a donc aucune raison d'être. Avoir une chaîne désignée en notation TTCN-3 est inévitable. Cela est requis, par exemple, quand une chaîne de caractères reçue est à utiliser pour l'appariement d'une autre chaîne. Il n'y a aucune autre possibilité que de sauvegarder la première chaîne dans une variable et de faire référence au nom de cette variable dans la structure d'appariement de la seconde chaîne. Dans l'exemple : foo := "ab", structure "{foo}#2", la seule chaîne qui correspond est "abab". Une meilleure solution serait bienvenue. L'expression \N{} a une fonctionnalité différente en notation ASN.1. Inclure l'expression \N dans la notation TTCN-3 nécessiterait une contribution technique.

G.6 Comment le caractère "^" est-il manipulé quand il n'est pas le premier caractère d'un groupe?

Dans ce cas, le comportement est le même qu'en notation ASN.1. Il ne possède aucune signification particulière quand NOT est le premier caractère faisant suite au crochet d'ouverture d'un groupe et il doit être manipulé exactement comme un caractère littéral. Il est défini en notation TTCN-3. Le Tableau B.1 indique:

> "^Utilisable 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"

B.1.5.1 indique: "L'expression d'un ensemble peut également être rendue négative par insertion du caractère '^' comme premier caractère après le crochet d'ouverture." Aucune autre utilisation n'est définie pour "^".

G.7 Est-ce que les métacaractères sont autorisés dans un ensemble précédé par un "\"?

La notation TTCN-3 définit clairement que seuls les caractères littéraux et les étendues sont autorisés dans des ensembles. Voir le § B.1.5.1, qui indique: "L'expression d'un ensemble est délimitée par les symboles '[' ']'. En plus des caractère littéraux, il est possible de spécifier des séries de caractères au moyen du séparateur '-'." A cet égard, la notation TTCN-3 est plus restrictive que la notation ASN.1 qui permet un quadruplet, \N, \d, \t, \w, \r, \n, \s et \b à l'intérieur d'une paire de crochets.

G.8 Des ensembles peuvent-ils être imbriqués?

La réponse est la même que pour G.7. Comme un ensemble expression correspond en une valeur isolée de type seul caractère position, le [a-m[n-z]] serait: le même que [a-z] (mais il ne sont pas autorisées) ou [a-m[o-z]] est le même que [a-mo-z].

G.9 Une expression de référence peut-elle être utilisée à l'intérieur d'un ensemble?

La réponse est la même que pour G.7.

G.10 Comment les caractères "?" et "*" sont-ils utilisés dans les expressions régulières de la notation TTCN-3?

Les caractères "?" et "*" sont utilisés différemment dans les expressions régulières ASN.1 et TTCN-3. L'aspect qui importe en notation TTCN-3 est que les caractères "?" et "*" servent à apparier des expressions régulières de la même façon que lors de l'appariement de toute autre valeur de remplacement ou interne. Les mécanismes d'appariement sont hérités de la notation TTCN-2. En ASN.1, les suffixes "?" et "*" sont également des items hérités; il faudra s'accommoder de ces différences. Le caractère TTCN-3 "?" et le caractère ASN.1 "." ne devraient pas être comparés. Les caractères TTCN-3 "?" (et "*") "englobent" des caractères quelconques y compris les caractères de commande comme LF, CR, HT, espace, etc., ce qui n'est pas le cas du caractère ASN.1 ".". Inclure le caractère "." dans la notation TTCN-3 serait une modification technique.

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 nouvelle génération
Série Z	**Langages et aspects généraux logiciels des systèmes de télécommunication**

Imprimé en Suisse
Genève, 2006