

Union internationale des télécommunications

UIT-T

SECTEUR DE LA NORMALISATION
DES TÉLÉCOMMUNICATIONS
DE L'UIT

Z.142

(03/2006)

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

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

**Notation de test et de commande de test
version 3 (TTCN-3): format de présentation
graphique**

Recommandation UIT-T Z.142

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.

Notation de test et de commande de test version 3 (TTCN-3): format de présentation graphique

Résumé

La présente Recommandation définit le format GFT, format de présentation graphique du langage noyau de la notation de test et de commande de test (TTCN-3, *testing and test control notation 3*) défini dans la Rec. UIT-T Z.140. Ce format est utilisé pour présenter graphiquement un comportement de test à l'aide d'un sous-ensemble de diagrammes de séquences de messages définis dans la Rec. UIT-T Z.120 avec des extensions propres aux tests. Il est constitué d'un certain nombre de symboles graphiques permettant de présenter graphiquement des tests élémentaires, des fonctions, des variantes et des parties commande TTCN-3. On peut l'appliquer chaque fois qu'il est nécessaire de définir ou de présenter graphiquement un comportement de test.

La présente Recommandation est fondée sur le langage noyau de la notation TTCN-3 défini dans la Rec. UIT-T Z.140. Elle porte en particulier sur l'affichage de tests sous la forme de diagrammes GFT. Son domaine d'application ne se limite pas à un type particulier de spécification de test.

Source

La Recommandation UIT-T Z.142 a été approuvée le 16 mars 2006 par la Commission d'études 17 (2005-2008) de l'UIT-T selon la procédure définie dans la Recommandation UIT-T A.8.

AVANT-PROPOS

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

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

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

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

NOTE

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

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

DROITS DE PROPRIÉTÉ INTELLECTUELLE

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

A la date d'approbation de la présente Recommandation, l'UIT n'avait pas été avisée de l'existence d'une propriété intellectuelle protégée par des brevets à acquérir pour mettre en œuvre la présente Recommandation. Toutefois, comme il ne s'agit peut-être pas des renseignements les plus récents, il est vivement recommandé aux développeurs de consulter la base de données des brevets du TSB sous <http://www.itu.int/ITU-T/ipr/>.

© UIT 2007

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

TABLE DES MATIÈRES

		<i>Page</i>
1	Domaine d'application	1
2	Références normatives	1
3	Abréviations	1
4	Aperçu général.....	2
5	Concepts associés au langage de format GFT	3
6	Mappage entre le format GFT et le langage noyau TTCN-3	4
7	Structure d'un module.....	5
8	Symboles GFT.....	7
9	Diagrammes GFT	9
	9.1 Propriétés communes	9
	9.2 Diagramme de commande.....	9
	9.3 Diagramme de test élémentaire	10
	9.4 Diagramme de fonction	11
	9.5 Diagramme de variante	12
10	Instances dans des diagrammes GFT	13
	10.1 Instance de commande	13
	10.2 Instances de composante de test	13
	10.3 Instances de port	14
11	Eléments de diagrammes GFT.....	14
	11.1 Règles générales de représentation graphique	14
	11.2 Invocation de diagrammes GFT	15
	11.3 Déclarations	17
	11.4 Instructions de programme de base	19
	11.5 Instructions de programmation comportementales	22
	11.6 Gestion des comportements par défaut	26
	11.7 Opérations de configuration	27
	11.8 Opérations de communication.....	30
	11.9 Opérations de temporisation	46
	11.10 Opérations de verdict de test.....	48
	11.11 Actions externes	49
	11.12 Spécification des attributs	49
Annex A – GFT BNF.....		50
	A.1 Meta-language for GFT	50
	A.2 Conventions for the syntax description	50
	A.3 The GFT grammar.....	50
Annex B – Reference guide for GFT.....		74
Annex C – Examples		97
	C.1 The Restaurant example	97
	C.2 The INRES example	106

Introduction

Le format de présentation graphique de la notation TTCN-3 (GFT) est fondé sur la Rec. UIT-T Z.120 [3] qui définit des diagrammes de séquences de messages (MSC, *message sequence chart*). Le format GFT utilise un sous-ensemble de diagrammes MSC avec des extensions propres aux tests. La plupart des extensions sont uniquement de nature textuelle. Les extensions graphiques sont définies pour faciliter la lecture des diagrammes GFT. Lorsque cela est possible, le format GFT est défini de la même façon qu'un diagramme MSC, de telle sorte que des outils MSC existant peuvent moyennant de légères modifications, être utilisés pour définir graphiquement des tests élémentaires TTCN-3 suivant le format GFT.

Le langage noyau de la notation TTCN-3 est défini dans la Rec. UIT-T Z.140 [1] et fournit une syntaxe textuelle complète, une sémantique statique, une sémantique d'exploitation ainsi que la définition de son utilisation avec la notation ASN.1. Le format de présentation GFT offre un autre moyen d'afficher le langage noyau (voir la Figure 1).

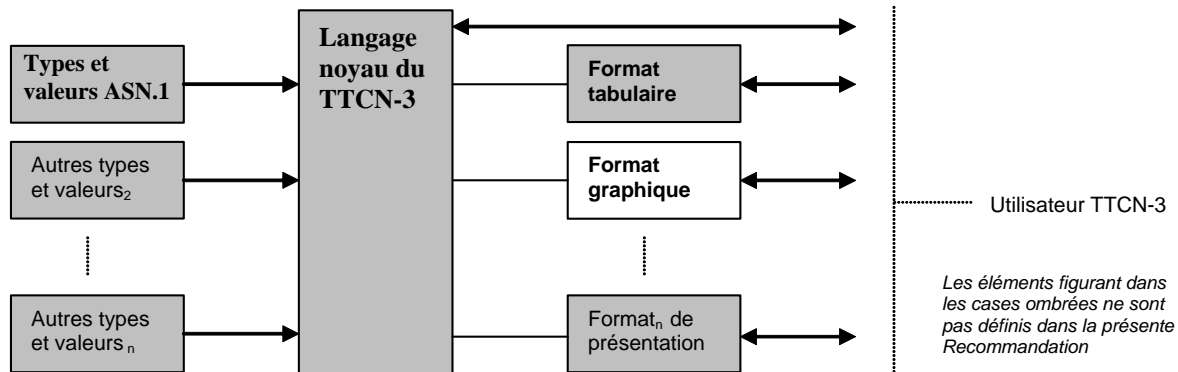


Figure 1/Z.142 – Vue d'utilisateur du langage noyau et des divers formats de présentation

Le langage noyau peut être utilisé indépendamment du format GFT. Toutefois, ce format ne peut pas être utilisé sans le langage noyau. Il doit être utilisé ou implémenté sur la base du langage noyau.

La présente Recommandation définit:

- les concepts de langage du format GFT;
- les lignes directrices relatives à l'utilisation du format GFT;
- la grammaire associée au format GFT;
- le mappage associé au langage noyau de la notation TTCN-3.

L'ensemble de ces caractéristiques constitue le format GFT – format de présentation graphique de la notation TTCN-3.

Notation de test et de commande de test version 3 (TTCN-3): format de présentation graphique

1 Domaine d'application

La présente Recommandation définit le format de présentation graphique du langage noyau de la notation TTCN-3, langage qui est spécifié dans la Rec. UIT-T Z.140 [1]. Ce format de présentation utilise un sous-ensemble de diagrammes de séquences de messages définis dans la Rec. UIT-T Z.120 [3] avec des extensions spécifiques aux tests.

La présente Recommandation est fondée sur le langage noyau TTCN-3 défini dans la Rec. UIT-T Z.140 [1]. Elle porte en particulier sur l'affichage de tests, tels que ceux en format GFT. Son domaine d'application ne se limite pas à un type particulier de spécification de test.

La spécification d'autres formats est hors du domaine d'application de la présente Recommandation.

2 Références normatives

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

- [1] Recommandation UIT-T Z.140 (2006), *Notation de test et de commande de test version 3 (TTCN-3): langage noyau.*
 - [2] Recommandation UIT-T Z.141 (2006), *Notation de test et de commande de test version 3 (TTCN-3): format de présentation tabulaire.*
 - [3] Recommandation UIT-T Z.120 (2004), *Diagramme des séquences de messages.*
 - [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).*
- ISO/CEI 9646-3:1998, *Technologies de l'information – Interconnexion de systèmes ouverts – Essais de conformité – Méthodologie générale et procédures – Partie 3: Notation combinée, arborescente et tabulaire (TTCN).*

3 Abréviations

La présente Recommandation utilise les abréviations suivantes:

BNF	formalisme de Backus-Naur (<i>Backus-Naur form</i>)
CATG	génération de tests informatisés (<i>computer-aided test generation</i>)
GFT	format de présentation graphique de la notation TTCN-3 (<i>graphical presentation format of TTCN-3</i>)
MSC	diagramme de séquences de messages (<i>message sequence chart</i>)
MTC	composante de test principal (<i>main test component</i>)
PTC	composante de test parallèle (<i>parallel test component</i>)
SUT	système sous test (<i>system under test</i>)
TFT	format de présentation tabulaire de la notation TTCN-3 (<i>tabular presentation format of TTCN-3</i>)
TTCN	notation de test et de commande de test (<i>testing and test control notation</i>)

4 Aperçu général

Conformément à la méthodologie des tests de conformité OSI définie dans la Rec. UIT-T X.292 [4], une procédure de test commence généralement par l'identification de ses objectifs. Un objectif de test est défini comme suit:

"Une description textuelle d'un objectif de test bien défini, portant essentiellement sur une prescription de conformité unique ou sur un ensemble de prescriptions de conformité liées, conformément à la spécification OSI appropriée".

Après avoir identifié tous les objectifs de test, on élabore une suite de tests abstraite comprenant un ou plusieurs tests élémentaires abstraits. Un test élémentaire abstrait définit les processus de test nécessaires pour valider une partie (ou l'ensemble) d'un objectif de test.

En appliquant ces termes aux diagrammes des séquences de messages, on peut définir deux catégories pour leur utilisation:

- 1) *utilisation des diagrammes MSC pour la définition des objectifs de test* – Généralement, une spécification de diagramme MSC développé en tant que modèle ou dans le cadre d'une spécification de système peut être considérée comme un objectif de test, c'est-à-dire qu'elle décrit une prescription applicable au système testé sous la forme d'une description de comportement susceptible d'être testé. Par exemple, la Figure 2 correspond à un diagramme MSC simple décrivant l'interaction entre des instances représentant le système testé et ses interfaces A, B et C. Dans le cas d'une implémentation réelle de ce système, les interfaces A, B et C pourraient être mappées à des points d'accès de service ou à des ports. Le diagramme MSC représenté sur la Figure 2 décrit uniquement l'interaction avec le système sous test et ne décrit pas les actions des composants de test nécessaires pour valider le comportement du système sous test (il s'agit de la description d'un objectif de test).

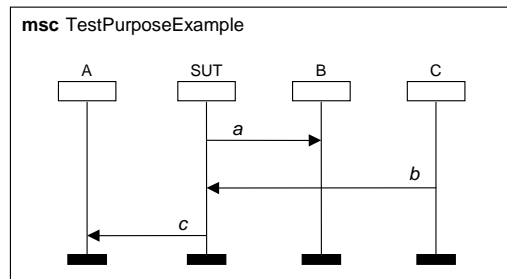


Figure 2/Z.142 – Diagramme MSC décrivant l'interaction entre un système sous test et ses interfaces

- 2) *Utilisation des diagrammes MSC pour la définition de tests élémentaires abstraits* – Une spécification de diagramme MSC décrivant un test élémentaire abstrait permet de spécifier le comportement des composants de test nécessaires en vue de valider un objectif de test associé. La Figure 3 décrit un test élémentaire abstrait de diagramme MSC simple: une composante de test principale (MTC) échange les messages *a*, *b* et *c* avec le système sous test via les ports PortA, PortB et PortC en vue de réaliser l'objectif de test décrit sur la Figure 2. Les messages *a* et *c* sont envoyés par le système sous test via les ports A et B (Figure 2) et reçus par la composante MTC (Figure 3) via les mêmes ports. Le message *b* est envoyé par la composante MTC et reçu par le système sous test.

NOTE – Les exemples des Figures 2 et 3 ne sont que des cas simples qui permettent d'illustrer les différentes utilisations d'un diagramme MSC à des fins de test. Ces diagrammes seront plus compliqués dans le cas d'un système sous test réparti comprenant plusieurs processus ou d'une configuration de test réparti faisant intervenir plusieurs composants de test.

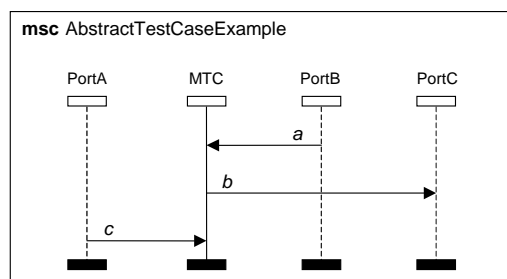


Figure 3/Z.142 – Diagramme MSC décrivant l'interface entre une composante MTC et les interfaces du système sous test

On peut distinguer deux types d'utilisation de diagrammes MSC (voir la Figure 4):

- a) *la génération de tests élémentaires abstraits à partir de descriptions d'objectifs de test de diagrammes MSC* – Le langage noyau TTCN-3 ou le format GFT peuvent être utilisés pour représenter des tests élémentaires abstraits. Il apparaît toutefois que la génération de tests élémentaires à partir d'objectifs de test n'est pas triviale et nécessite l'utilisation et le développement de techniques de génération CATG;
- b) *élaboration d'un format de présentation graphique pour la notation TTCN-3 (GFT) et définition du mappage entre le format GFT et la notation TTCN-3.*

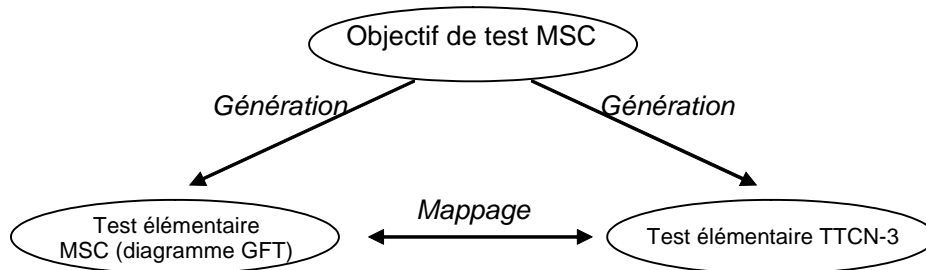


Figure 4/Z.142 – Relations entre la description d'un objectif de test de diagramme MSC, la description d'un test élémentaire de diagramme MSC et la notation TTCN-3

La présente Recommandation porte essentiellement sur le point b): elle définit le format GFT ainsi que son mappage avec le langage noyau TTCN-3.

5 Concepts associés au langage de format GFT

Le format GFT représente graphiquement des aspects comportementaux du langage TTCN-3 tels que le comportement d'un test élémentaire ou d'une fonction. Il ne fournit pas de graphiques sur des points relatifs aux données, tels que la déclaration de types et de modèles.

Le format GFT ne définit pas de représentation graphique relative à la structure d'un module de notation TTCN-3, mais spécifie les prescriptions applicables à cette représentation (voir également le § 7).

NOTE – L'ordre et le groupement des définitions et des déclarations dans la partie définitions d'un module définissent la structure d'un module de notation TTCN-3.

Le format GFT ne définit pas de représentation graphique pour:

- les définitions des paramètres d'un module;
- les définitions des importations;
- les définitions des types;
- les déclarations des signatures;
- les déclarations des modèles;
- les déclarations des constantes;
- les déclarations des constantes externes;
- les déclarations des fonctions externes.

Les définitions et les déclarations en notation TTCN-3 sans présentation de format GFT correspondant peuvent être présentées en langage noyau TTCN-3 ou suivant le format de présentation tabulaire pour la notation TTCN-3 (TFT) (Rec. UIT-T Z.141 [2]).

Le format GFT fournit des graphiques pour les descriptions de comportement de notation TTCN-3. Un diagramme de format GFT permet donc de représenter graphiquement:

- la partie commande d'un module de notation TTCN-3;
- un test élémentaire de notation TTCN-3;
- une fonction de notation TTCN-3;
- une variante de notation TTCN-3.

La relation entre un module de notation TTCN-3 et une présentation de format GFT correspondante est indiquée sur la Figure 5.

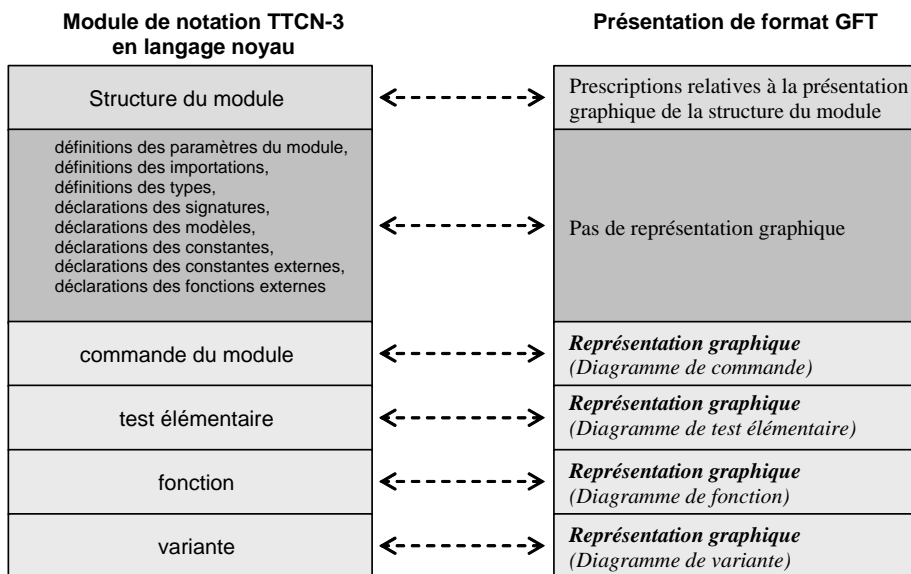


Figure 5/Z.142 – Relation entre le langage noyau TTCN-3 et la description de format GFT correspondante

Le format GFT étant fondé sur un diagramme MSC (Rec. UIT-T Z.120 [3]), un mappage existe entre un diagramme de format GFT et un diagramme MSC. Bien que la plupart des symboles graphiques de diagramme MSC soient utilisés en format GFT, la désignation de certains d'entre eux a été adaptée aux besoins de test; certains nouveaux symboles ont, en outre, été définis pour souligner des aspects spécifiques au test. De nouveaux symboles peuvent toutefois être mappés vers des diagrammes MSC valables.

- la représentation d'instances de port;
- la création de composantes de test;
- le lancement de composantes de test;
- le renvoi d'un appel de fonction;
- la répétition de variantes;
- la surveillance temporelle d'un appel en mode procédure;
- l'exécution de tests élémentaires;
- l'activation et la désactivation de comportements par défaut;
- l'étiquetage et la fonction aller à;
- les temporisateurs au sein des instructions d'appel.

On trouvera dans le § 8 la liste complète de tous les symboles utilisés dans le format GFT.

6 Mappage entre le format GFT et le langage noyau TTCN-3

Le format GFT fournit des moyens graphiques pour définir les comportements en notation TTCN-3. La partie commande et chaque fonction, variante et test élémentaire d'un module de langage noyau TTCN-3 peuvent être mappés vers un diagramme GFT et vice versa. Ainsi:

- la partie commande du module peut être mappée vers un diagramme de commande (voir le § 9.2) et vice versa;
- un test élémentaire peut être mappé vers un diagramme de test élémentaire (voir le § 9.3) et vice versa;

- une fonction en langage noyau peut être mappée vers un diagramme de fonction (voir le § 9.4) et vice versa;
- une variante peut être mappée vers un diagramme de variante (voir le § 9.5) et vice versa.

NOTE 1 – Le format GFT ne fournit pas de représentation graphique pour les définitions des paramètres, types, constantes, signatures, modèles, constantes externes et fonctions externes du module dans la partie définitions du module. Ces définitions peuvent être données directement en langage noyau ou en utilisant un autre format de présentation, par exemple le format de présentation tabulaire.

Chaque déclaration, opération et instruction de la commande de module et chaque test élémentaire, variante ou fonction peuvent être mappés vers la représentation de format GFT correspondante et vice versa.

L'ordre des déclarations, des opérations et des instructions dans une commande de module, un test élémentaire, une variante ou une fonction est identique à l'ordre des représentations de format GFT correspondant dans le diagramme de commande, de test élémentaire, de variante ou de fonction associé.

NOTE 2 – L'ordre des structures GFT dans un diagramme GFT est défini par l'ordre des structures GFT dans l'en-tête du diagramme (qui comprend uniquement des déclarations) et par l'ordre des structures GFT dans l'instance de commande (diagramme de commande) ou l'instance de composante (diagramme de test élémentaire, diagramme de variante ou diagramme de fonction).

7 Structure d'un module

Comme on le voit sur la Figure 6, un module de notation TTCN-3 a une structure arborescente. Il comprend une partie définitions et une partie commande. La partie définitions comprend des définitions et des déclarations qui peuvent être structurées en groupes. La partie commande ne peut pas être organisée en sous-structures: elle définit l'ordre d'exécution et les conditions d'exécution des tests élémentaires.

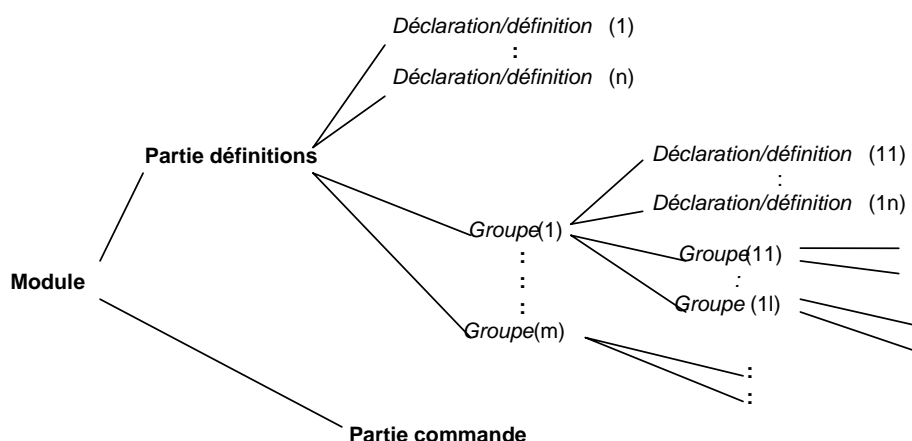


Figure 6/Z.142 – Structure d'un module de notation TTCN-3

Le format GFT fournit des diagrammes pour toutes les branches "comportementales" de la structure arborescente du module, c'est-à-dire pour la partie commande, les fonctions, les variantes et les tests élémentaires. Il ne définit pas concrètement de graphique pour la structure arborescente du module; toutefois, pour fournir un appui approprié en termes d'outils au format GFT, il faut une représentation graphique de la structure d'un module de notation TTCN-3. La structure d'un module de notation TTCN-3 peut être indiquée par le biais d'une vue organisationnelle (Figure 7) ou d'une présentation de type document MSC (Figure 8). Un outil évolué peut même prendre en charge différentes présentations d'un même objet; ainsi, la vue organisationnelle de la Figure 7 indique que certaines définitions sont fournies suivant plusieurs formats de présentation (par exemple, la fonction MySpecialFunction est disponible en langage noyau, sous la forme de tableau TFT et en tant que diagramme GFT).

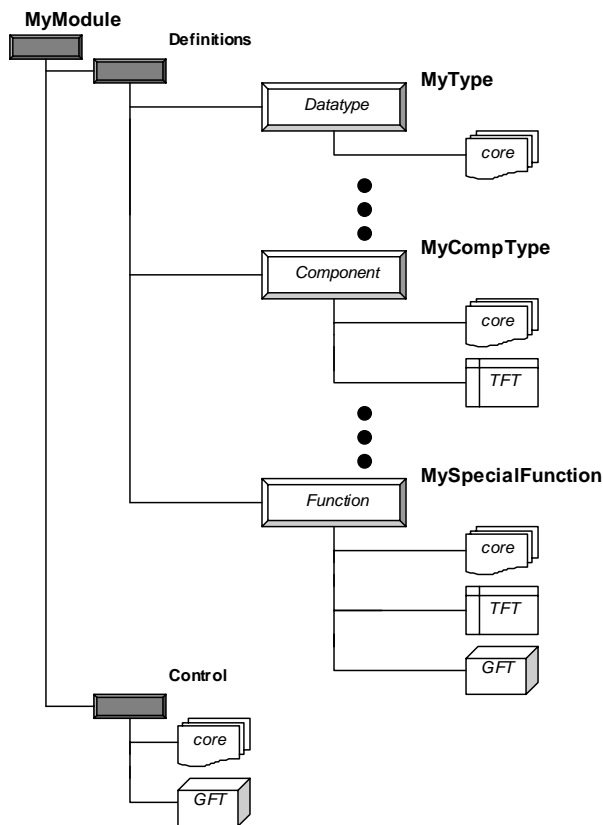


Figure 7/Z.142 – Divers formats de présentation d'une structure de module de notation TTCN-3 au sein d'une vue organisationnelle

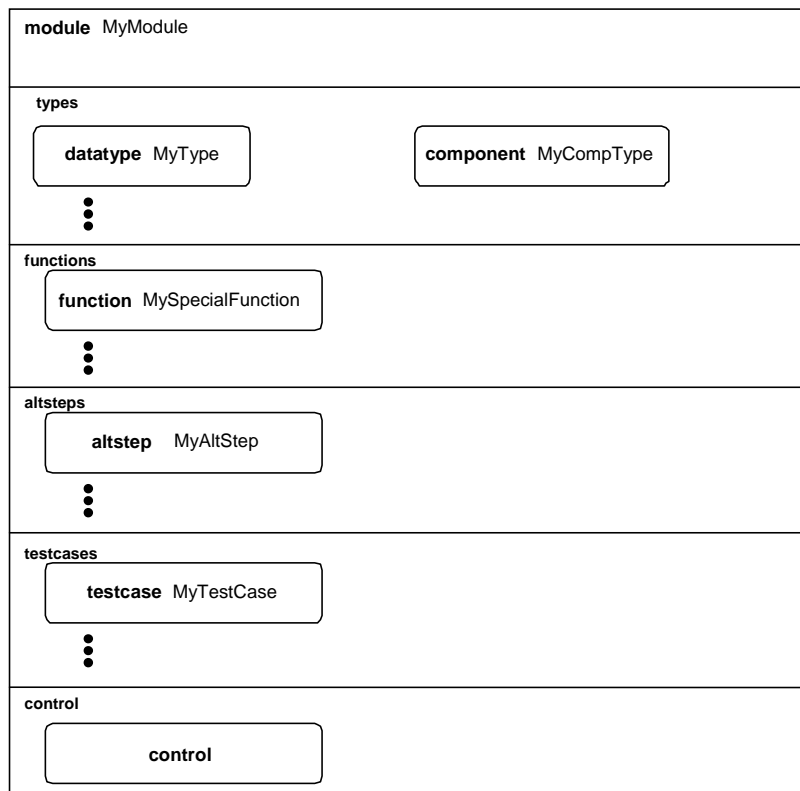


Figure 8/Z.142 – Présentation graphique de type document MSC d'une structure de module de notation

8 Symboles GFT

Le présent paragraphe précise tous les symboles graphiques utilisés dans les diagrammes GFT ainsi que des commentaires sur leur utilisation type.

Tableau 1/Z.142 – Symboles GFT




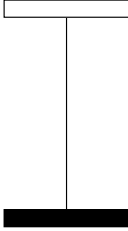


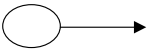

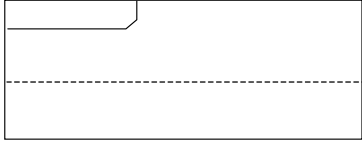




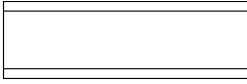










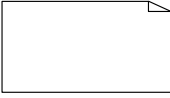
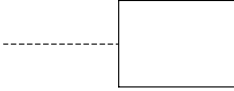
Elément GFT	Symbole	Description
Symbole de cadre		Utilisé pour encadrer les diagrammes de format GFT
Symbole de référence		Utilisé pour représenter l'invocation de fonctions et de variantes
Symbole d'instance de port		Utilisé pour représenter des instances de port
Symbole d'instance de composante		Utilisé pour représenter des composantes de test et l'instance de commande
Symbole de cadre d'action		Utilisé pour les déclarations et les instructions textuelles en notation TTCN-3, à rattacher à un symbole de composante.
Symbole de condition		Utilisé pour les expressions booléennes, les définitions de verdict, les opérations de port (start, stop et clear) textuelles TTCN-3 et pour l'instruction done; à rattacher à un symbole de composante.
Symbole d'étiquette		Utilisé pour les instructions label et goto TTCN-3, à rattacher à un symbole de composante.
Symbole aller à		Utilisé pour les instructions label et goto TTCN-3, à rattacher à un symbole de composante.
Symbole d'expression en ligne		Utilisé pour les instructions if-else, for, while, do-while, alt, call et interleave TTCN-3, à rattacher à un symbole de composante.
Symbole de comportement par défaut		Utilisé pour les instructions activate et deactivate en notation TTCN-3, à rattacher à un symbole de composante.

Tableau 1/Z.142 – Symboles GFT

Elément GFT	Symbole	Description
Symbole d'arrêt		Utilisé pour l'instruction stop TTCN-3, à rattacher à un symbole de composante.
Symbole de retour		Utilisé pour l'instruction return TTCN-3, à rattacher à un symbole de composante.
Symbole de répétition		Utilisé pour l'instruction repeat TTCN-3, à rattacher à un symbole de composante.
Symbole de création		Utilisé pour l'instruction create TTCN-3, à rattacher à un symbole de composante.
Symbole de lancement		Utilisé pour l'instruction start TTCN-3, à rattacher à un symbole de composante.
Symbole de message		Utilisé pour représenter les instructions send, call, reply, raise, receive, getcall, getreply, catch, trigger et check TTCN-3, à rattacher à un symbole de composante et à un symbole de port.
Symbole d'obtention		Utilisé pour représenter les instructions receive, getcall, getreply, catch, trigger et check TTCN-3 depuis un port quelconque, à rattacher à un symbole de composante.
Symbole de région de suspension		Utilisé en association à un appel bloquant, à placer dans une expression en ligne d'appel et à rattacher à un symbole de composante.
Symbole de déclenchement de temporisateur		Utilisé pour l'opération de déclenchement de temporisateur TTCN-3, à rattacher à un symbole de composante.
Symbole d'expiration de temporisateur		Utilisé pour l'opération d'expiration de temporisateur TTCN-3, à associer à un symbole de composante.
Symbole d'arrêt de temporisateur		Utilisé pour l'opération d'arrêt de temporisateur TTCN-3, à rattacher à un symbole de composante.
Symbole de déclenchement de temporisateur implicite		Utilisé pour le déclenchement de temporisateur implicite TTCN-3 en appel de blocage, à placer dans une expression en ligne d'appel et à rattacher à un symbole de composante.
Symbole d'expiration de temporisateur implicite		Utilisé pour l'exception d'expiration de temporisateur TTCN-3, à placer dans une expression en ligne d'appel et à rattacher à un symbole de composante.
Symbole d'exécution		Utilisé pour l'instruction d'exécution de test élémentaire TTCN-3, à rattacher à un symbole d'instance de composante.
Symbole de texte		Utilisé pour les instructions et les commentaires TTCN-3, à placer dans un diagramme GFT.
Symbole de commentaires d'événements		Utilisé pour les commentaires associés aux événements TTCN-3, à rattacher aux événements au niveau des symboles d'instance de composante ou d'instance de port.

9 Diagrammes GFT

Les différents types de diagramme GFT sont les suivants:

- a) *diagramme de commande* pour la représentation graphique d'une partie commande de module TTCN-3;
- b) *diagramme de test élémentaire* pour la représentation graphique d'un test élémentaire TTCN-3;
- c) *diagramme de variante* pour la représentation graphique d'une variante TTCN-3;
- d) *diagramme de fonction* pour la représentation graphique d'une fonction TTCN-3.

Ces différents types de diagramme ont certaines propriétés communes.

9.1 Propriétés communes

Les propriétés que partagent les diagrammes GFT se rapportent à la zone de diagramme, à l'en-tête de diagramme et à la pagination.

9.1.1 Surface du diagramme

Chaque diagramme GFT de commande, de test élémentaire, de variante et de fonction doit avoir un symbole de cadre (également appelé cadre de diagramme) pour définir la surface qu'il occupe. Tous les symboles et textes nécessaires pour définir un diagramme GFT complet et syntaxiquement correct doivent figurer à l'intérieur de la surface du diagramme.

NOTE – Le format GFT ne dispose pas de structures de langage telles que les portes MSC, situées hors du diagramme mais reliées à ce dernier.

9.1.2 En-tête du diagramme

Chaque diagramme GFT doit avoir un en-tête. Celui-ci doit être placé dans son coin supérieur gauche.

L'en-tête du diagramme doit identifier de manière univoque chaque type de diagramme GFT. La règle générale à observer à cet effet est de construire l'en-tête à partir du mot clé **testcase**, **altstep** ou **function** suivi de la signature TTCN-3 du test élémentaire, de la variante ou de la fonction à représenter graphiquement. Dans le cas d'un diagramme de commande GFT, l'en-tête est construit à partir du mot clé **module** suivi du nom du module.

NOTE – En format MSC, le mot clé **msc** précède toujours le nom de diagramme, afin de pouvoir identifier les diagrammes MSC. Il n'existe pas de mot clé commun GFT permettant d'identifier tous les diagrammes GFT.

9.1.3 Pagination

Les diagrammes GFT peuvent être structurés en pages, un grand diagramme GFT pouvant ainsi être réparti sur plusieurs pages. Chaque page d'un diagramme multipage doit avoir un numéro inscrit au coin supérieur droit qui l'identifie de manière univoque. Cette numérotation est facultative si le diagramme n'occupe pas plusieurs pages.

NOTE 1 – On considère que le schéma concret de numérotation est une question d'ordre pratique, qui se situe donc hors du domaine d'application de la présente Recommandation. Un schéma de numérotation simple peut uniquement attribuer un numéro de page, alors qu'un schéma de numérotation évolué peut assurer la reconstruction d'un diagramme en utilisant simplement les informations de numérotation inscrites sur les différentes pages.

NOTE 2 – On considère que les prescriptions de pagination allant au-delà du schéma général de numérotation sont des questions d'ordre pratique, qui se situent donc hors du domaine d'application de la présente Recommandation. A des fins de lisibilité, l'en-tête de diagramme peut figurer sur chaque page; la ligne d'instance d'une instance qui se poursuit sur une autre page peut figurer sur le bord inférieur de la page, et l'en-tête d'instance d'une instance qui se poursuit peut être répété sur la page qui décrit cette suite.

9.2 Diagramme de commande

Un diagramme de commande GFT fournit une représentation graphique de la partie commande d'un module TTCN-3. L'en-tête doit être formé du mot clé **module** suivi du nom du module. Un tel diagramme ne doit comprendre qu'une seule instance de composante (également appelée instance de commande) ayant pour nom d'instance **control** sans aucune information de type. L'instance de commande décrit le comportement de la partie commande du module TTCN-3. Les attributs associés à cette partie commande doivent être spécifiés dans un symbole de texte du diagramme de commande. Le modèle de base d'un diagramme de commande GFT et la description correspondante en langage noyau TTCN-3 sont représentés sur la Figure 9.

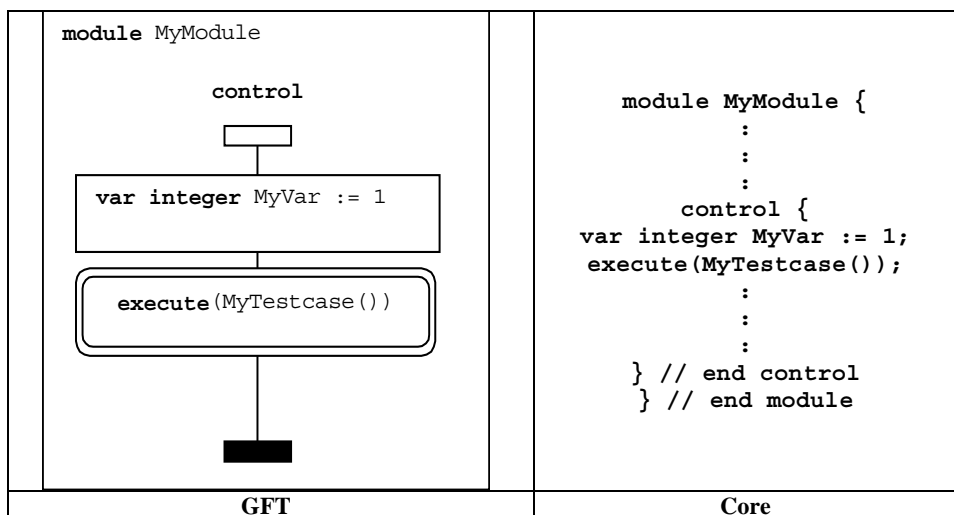


Figure 9/Z.142 – Modèle de base d'un diagramme de commande GFT et langage noyau correspondant

Dans la partie commande, les tests élémentaires peuvent être sélectionnés ou désélectionnés pour l'exécution d'un test élémentaire en utilisant des expressions booléennes. On peut utiliser des expressions, des affectations, des instructions **log**, des instructions **label** et **goto**, des instructions **if-else**, des instructions de boucle **for**, des instructions de boucle **while**, des instructions de boucle **do while**, des instructions d'exécution **stop** et des instructions de temporisation pour commander l'exécution de tests élémentaires. En outre, on peut utiliser des fonctions pour grouper les tests élémentaires et leurs conditions préalables à des fins d'exécution, l'invocation se faisant dans la partie commande du module.

La représentation en format GFT de ces caractéristiques de langage est conforme aux descriptions figurant ci-après dans les sections appropriées, à ceci près que pour la partie commande du module, les symboles graphiques sont rattachés à l'instance de commande et non pas à une instance de composante de test.

On se référera au § 11.4 pour la représentation en format GFT des expressions, des affectations, des instructions **log**, **label**, **goto**, **if-else**, boucle **for**, boucle **while**, boucle **do while** et **stop**, au § 11.9 pour les opérations de temporisation et aux § 9.4 et 11.2.2 pour les fonctions et leur invocation.

9.3 Diagramme de test élémentaire

Un diagramme de test élémentaire GFT fournit une représentation graphique d'un test élémentaire TTCN-3. L'en-tête doit être formé du mot clé **testcase** suivi de la signature complète (c'est-à-dire au moins le nom du test élémentaire et la liste des paramètres) du test élémentaire. La clause **runs on** est obligatoire et la clause **system** est facultative en langage noyau. Si elle est spécifiée dans le langage noyau associé, la clause **system** doit également figurer dans l'en-tête du diagramme de test élémentaire.

Un diagramme de test élémentaire GFT doit comprendre une instance de composante de test décrivant le comportement de la composante **mtc** (également appelée instance mtc) et une instance de port pour chaque port utilisable par la composante **mtc**. Le nom associé à l'instance mtc doit être **mtc**. Le type associé à l'instance mtc est facultatif, mais s'il est indiqué, il doit être identique au type de composante auquel il est fait référence dans la clause **runs on** de la signature de test élémentaire. Les noms associés aux instances de port doivent être identiques aux noms de port définis dans la définition de type de composante de la composante **mtc**. La composante **mtc** et les types de port sont affichés dans le symbole de début d'instance de composante ou de port.

Les attributs associés au test élémentaire représenté en format GFT doivent être spécifiés dans un symbole de texte du diagramme de test élémentaire. Le modèle de base d'un diagramme GFT de test élémentaire et la description correspondante en langage noyau TTCN-3 sont représentés sur la Figure 10.

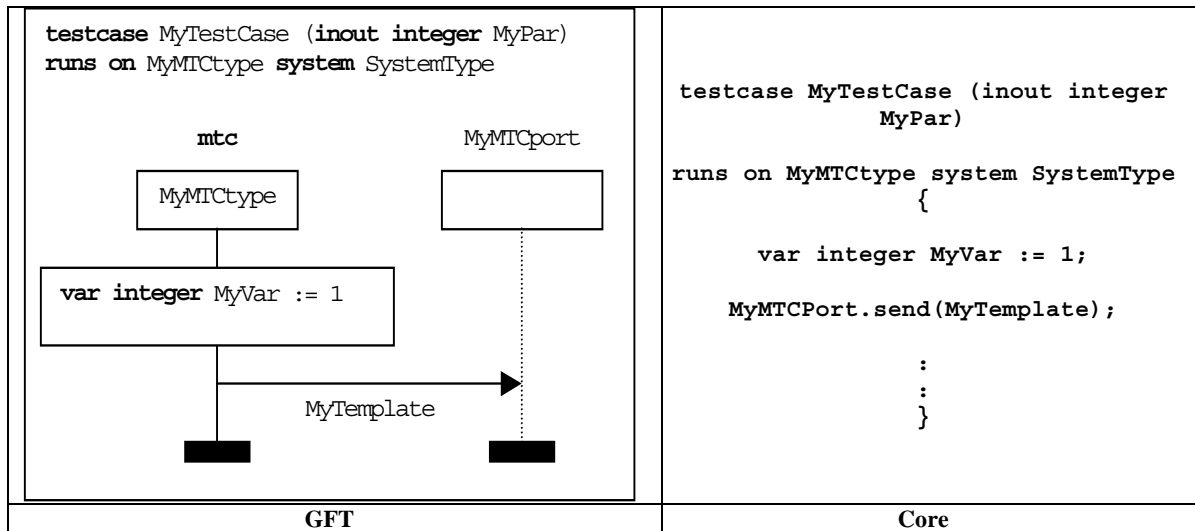


Figure 10/Z.142 – Modèle de base d'un diagramme GFT de test élémentaire et langage noyau correspondant

Un test élémentaire permet de représenter le comportement dynamique du test et de créer des composantes de test. Il peut comprendre des déclarations, des instructions, des opérations de communication et de temporisation ainsi que l'invocation de fonctions ou de variantes.

9.4 Diagramme de fonction

Le format GFT représente des fonctions TTCN-3 au moyen de diagrammes de fonction. L'en-tête d'un diagramme de fonction doit être formé du mot clé **function** suivi de la signature complète (c'est-à-dire au moins le nom de la fonction et la liste des paramètres) de la fonction. La clause **return** et la clause **runs on** sont facultatives dans le langage noyau. Si elles sont spécifiées dans le langage noyau associé, ces clauses doivent également figurer dans l'en-tête du diagramme de fonction.

Un diagramme de fonction GFT doit comprendre une instance de composante de test décrivant le comportement de la fonction et une instance de port pour chaque port utilisable par la fonction.

NOTE – Les noms et les types des ports utilisables par une fonction sont transmis sous forme de paramètres ou correspondent aux noms et types de port définis dans la définition de type de composante référencée dans la clause **runs on**.

Le nom associé à l'instance de composante de test doit être **self**. Le type associé à l'instance de composante de test est facultatif, mais s'il est indiqué, il doit être conforme au type de composante mentionné dans la clause **runs on**.

Les noms et les types associés aux instances de port doivent être conformes aux paramètres de port (si les ports utilisables sont transmis sous forme de paramètres) ou aux déclarations de ports dans la définition de type de composante référencée dans la clause **runs on**. L'information de type pour les instances de port est facultative.

Le nom **self** et le nom de port sont affichés au-dessus du symbole de début d'instance de composante, respectivement d'instance de port. Les types de composante et les types de port sont affichés dans le symbole de début d'instance de composante, respectivement d'instance de port.

Les attributs associés à la fonction représentée en format GFT doivent être spécifiés dans un symbole de texte du diagramme de fonction. Le modèle de base d'un diagramme GFT de fonction et la description correspondante en langage noyau TTCN-3 sont représentés sur la Figure 11.

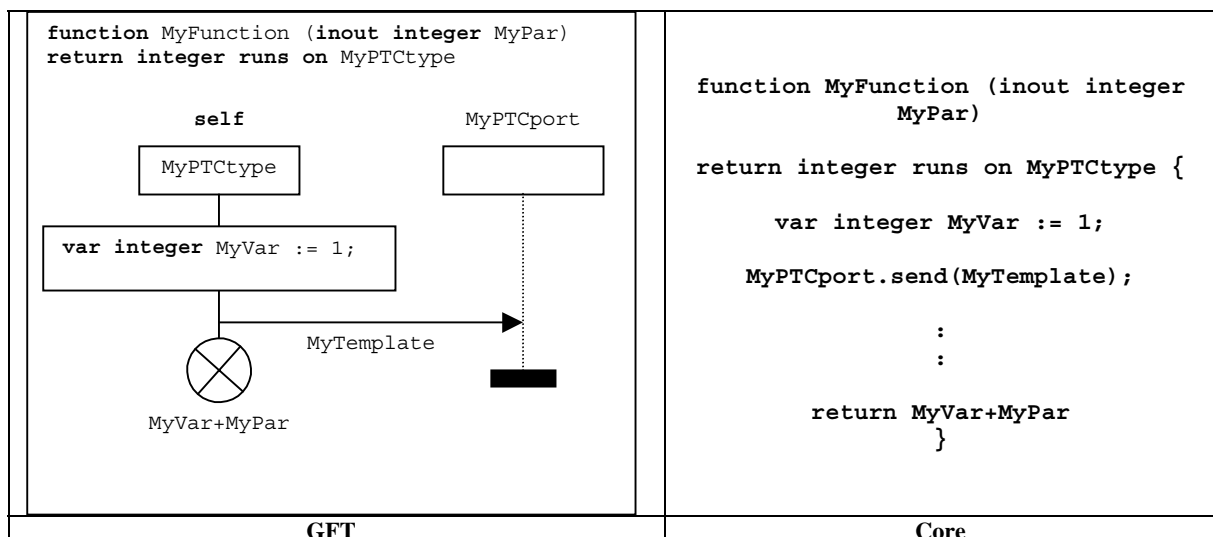


Figure 11/Z.142 – Modèle de base d'un diagramme GFT de fonction et langage noyau correspondant

Une fonction est utilisée pour spécifier et structurer un comportement de test, définir le comportement par défaut ou structurer le calcul dans un module. Elle peut comprendre des déclarations, des instructions, des opérations de communication et de temporisation, l'invocation de fonctions ou de variantes ainsi qu'une instruction de retour facultative.

9.5 Diagramme de variante

Le format GFT représente des variantes TTCN-3 au moyen de diagrammes de variante. L'en-tête d'un diagramme de variante doit être formé du mot clé **altstep** (variante) suivi de la signature complète (c'est-à-dire au moins le nom de la variante et la liste des paramètres) de la variante. La clause **runs on** est facultative en langage noyau. Si elle est spécifiée dans le langage noyau associé, elle doit également figurer dans l'en-tête du diagramme de variante.

Un diagramme de variante GFT doit comprendre une instance de composante de test décrivant le comportement de la variante et une instance de port pour chaque port utilisable par la variante.

NOTE – Les noms et les types des ports utilisables par une variante sont transmis sous forme de paramètres ou correspondent aux noms et types de port définis dans la définition de type de composante référencée dans la clause **runs on**.

Le nom associé à l'instance de composante de test doit être **self**. Le type associé à l'instance de composante de test est facultatif, mais s'il est indiqué, il doit être conforme au type de composante mentionné dans la clause **runs on**.

Les noms et les types associés aux instances de port doivent être conformes aux paramètres de port (si les ports utilisables sont transmis sous forme de paramètres) ou aux déclarations de port dans la définition de type de composante référencée dans la clause **runs on**. L'information de type pour les instances de port est facultative.

Le nom **self** et le nom de port sont affichés au-dessus du symbole de début d'instance de composante, respectivement d'instance de port. Les types de composante et les types de port sont affichés dans le symbole de début d'instance de composante, respectivement d'instance de port.

Les attributs associés à la variante représentée en format GFT doivent être spécifiés dans un symbole de texte du diagramme de variante. Le modèle de base d'un diagramme GFT de variante et la description correspondante en langage noyau TTCN-3 sont représentés sur la Figure 12.

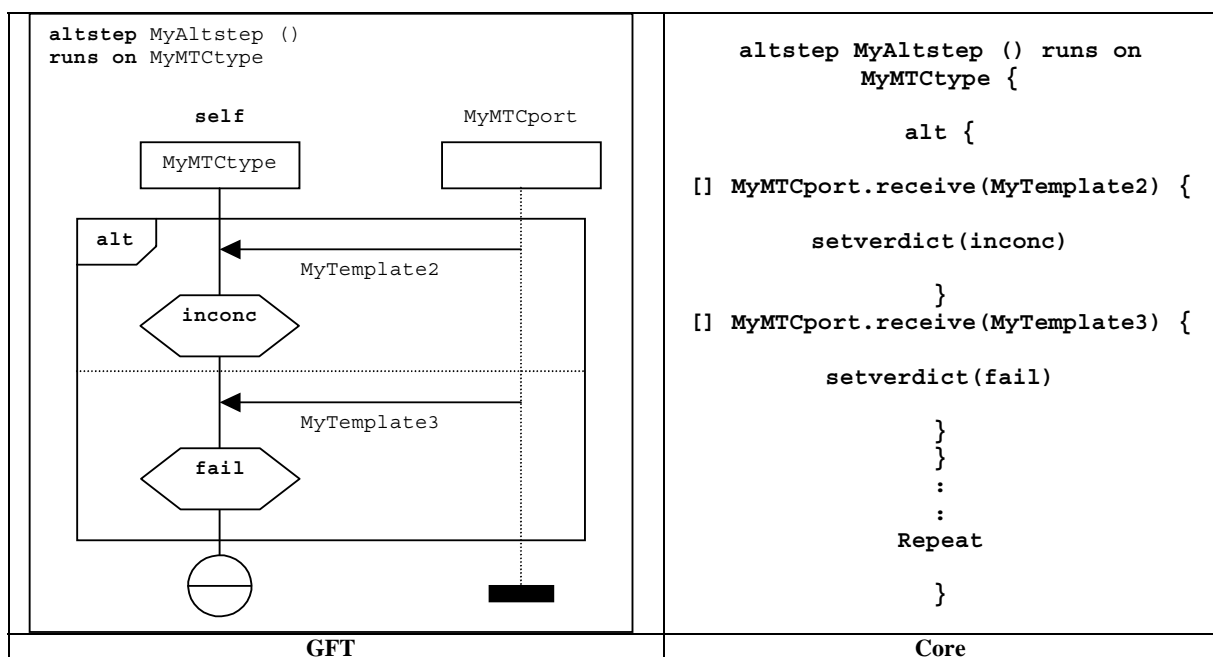


Figure 12/Z.142 – Modèle de base d'un diagramme GFT de variante et langage noyau correspondant

Une variante est utilisée pour spécifier le comportement par défaut ou pour structurer les alternatives d'une instruction **alt**. Elle peut comprendre des instructions, des opérations de communication et de temporisation ainsi que l'invocation de fonctions ou de variantes.

10 Instances dans des diagrammes GFT

Les diagrammes GFT comprennent les types d'instances suivantes:

- les *instances de commande*, qui décrivent le flux de commande dans la partie commande du module;
- les *instances de composante de test*, qui décrivent le flux de commande pour la composante de test qui exécute un test élémentaire, une fonction ou une variante;
- les *instances de port*, qui représentent les ports utilisés par les différentes composantes de test.

10.1 Instance de commande

Seule une instance de commande doit exister dans un diagramme de commande GFT (voir le § 9.2). Une instance de commande décrit le flux de commande de la partie commande d'un module. Une instance de commande GFT doit être décrite graphiquement à l'aide d'un symbole d'instance de composante, le nom obligatoire **control** étant placé au-dessus du symbole de début d'instance. Aucune information de type d'instance n'est associée à une instance de commande. Le modèle de base d'une instance de commande est décrit sur la Figure 13 a).

10.2 Instances de composante de test

Chaque diagramme GFT de test élémentaire, de fonction ou de variante, comprend une instance de composante de test qui décrit le flux de commande de l'instance. Une instance de composante de test doit être décrite graphiquement à l'aide d'une instance de symbole, en respectant les points suivants:

- le nom obligatoire **mtc** est placé au-dessus du symbole de début d'instance dans le cas d'un diagramme de test élémentaire;
- le nom obligatoire **self** est placé au-dessus du symbole de début d'instance dans le cas d'un diagramme de fonction ou de variante.

Le type de composante de test facultatif peut être fourni dans le symbole de début d'instance. Il doit être conforme au type de composante de test donné après le mot clé **runs on** dans l'en-tête du diagramme GFT.

Le modèle de base d'une instance de composante de test dans un diagramme de test élémentaire est représenté sur la Figure 13 b). Le modèle de base d'une instance de composante de test dans un diagramme de fonction ou de variante est représenté sur la Figure 13 c).

10.3 Instances de port

Des instances de port GFT peuvent être utilisées dans des diagrammes de test élémentaire, de variante ou de fonction. Une instance de port représente un port utilisable par une composante de test qui exécute le test élémentaire, la variante ou la fonction spécifié. Une instance de port GFT est décrite graphiquement à l'aide d'un symbole d'instance de composante associé à une ligne d'instance pointillée. Le nom du port représenté est une information obligatoire et doit être placé au-dessus du symbole de début d'instance. Le type de port (facultatif) peut être indiqué dans le symbole de début d'instance. Le modèle de base d'une instance de port est représenté sur la Figure 13 d).

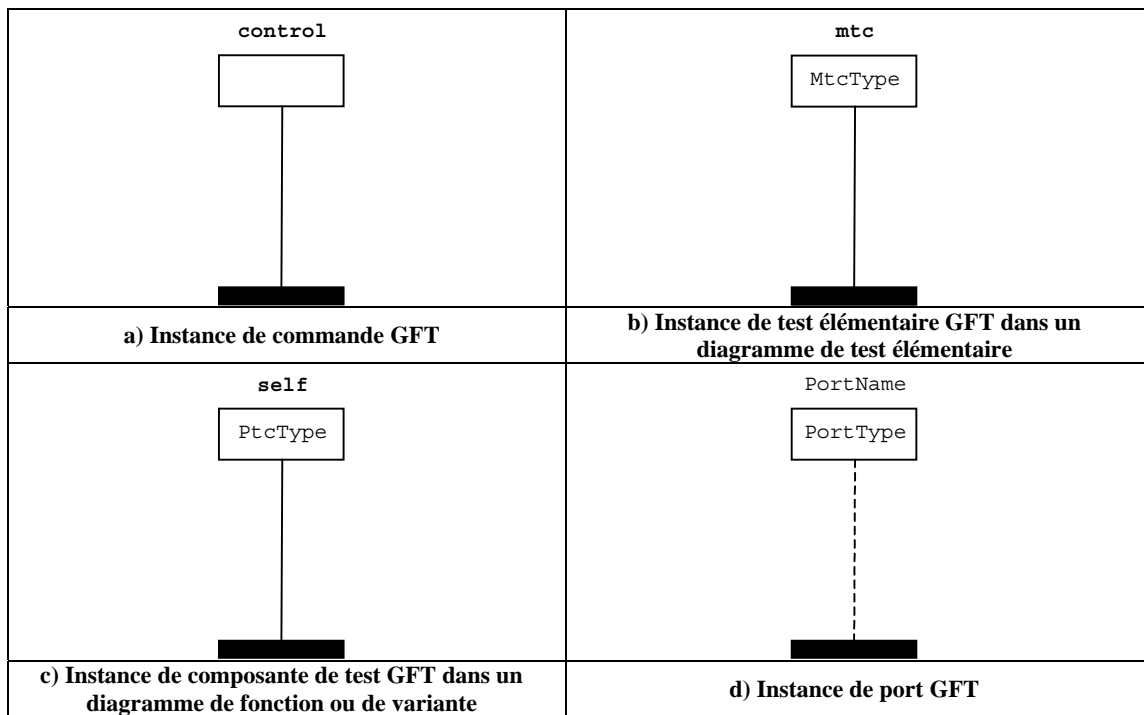


Figure 13/Z.142 – Principaux types d'instance de diagrammes GFT

11 Eléments de diagrammes GFT

Le présent paragraphe définit plusieurs règles graphiques générales relatives à la représentation d'éléments syntaxiques TTCN-3 spécifiques (points-virgules, commentaires). Il décrit la façon d'afficher l'exécution de diagrammes GFT et les symboles graphiques associés aux éléments de langage TTCN-3.

11.1 Règles générales de représentation graphique

Les règles générales de représentation graphique en format GFT se rapportent à l'utilisation des points virgule, des instructions TTCN-3 dans les symboles d'action et des commentaires.

11.1.1 Utilisation des points-virgules

Tous les symboles GFT à l'exception du symbole d'action doivent ne comprendre qu'une seule instruction en langage noyau TTCN-3. Seul un symbole d'action peut comprendre une séquence d'instructions TTCN-3 (voir le § 11.1.2).

L'utilisation d'un point-virgule est facultative si le symbole GFT ne comprend qu'une seule instruction en langage noyau TTCN-3 (voir les Figures 14 a) et Figure 14 b)).

Les points-virgules doivent séparer les instructions d'une séquence figurant dans un symbole d'action. Leur utilisation est facultative pour ce qui est de la dernière instruction de la séquence (Figure 14 c)).

Une séquence de déclarations de variables, de constantes et de temporisateurs peut également être spécifiée dans un langage noyau TTCN-3 simple à la suite de l'en-tête du diagramme GFT. Des points-virgules peuvent également séparer ces déclarations. Leur utilisation est facultative pour ce qui est de la dernière instruction de la séquence.

11.1.2 Utilisation des symboles d'action

Les déclarations, instructions et opérations TTCN-3 suivantes sont spécifiées dans des symboles d'action: les déclarations (moyennant les restrictions définies au § 11.3) et les assignations **log**, **connect**, **disconnect**, **map**, **unmap** et **action**.

Une séquence de déclarations, d'instructions et d'opérations à spécifier dans plusieurs symboles d'action peut être définie dans un même symbole d'action. Il n'est pas nécessaire d'utiliser un nouveau symbole d'action pour chaque déclaration, instruction ou opération.

11.1.3 Commentaires

En format GFT, il existe trois façons de placer des commentaires dans des diagrammes:

- les commentaires peuvent être placés dans des symboles GFT après l'inscription de symbole, en utilisant la syntaxe de commentaires du langage noyau TTCN-3 (Figure 14 d));
- les commentaires conformes à la syntaxe du langage noyau TTCN-3 peuvent être inscrits dans des symboles de texte librement placés dans le diagramme GFT (Figure 14 e));
- le symbole de commentaires peut être utilisé pour associer des commentaires à des symboles GFT. Les commentaires à inscrire dans un symbole de commentaires peuvent être fournis sans contrainte de syntaxe, ce qui signifie qu'il est inutile d'utiliser les délimiteurs de commentaire "/*", "*/" et "//" du langage noyau (Figure 14 f)).

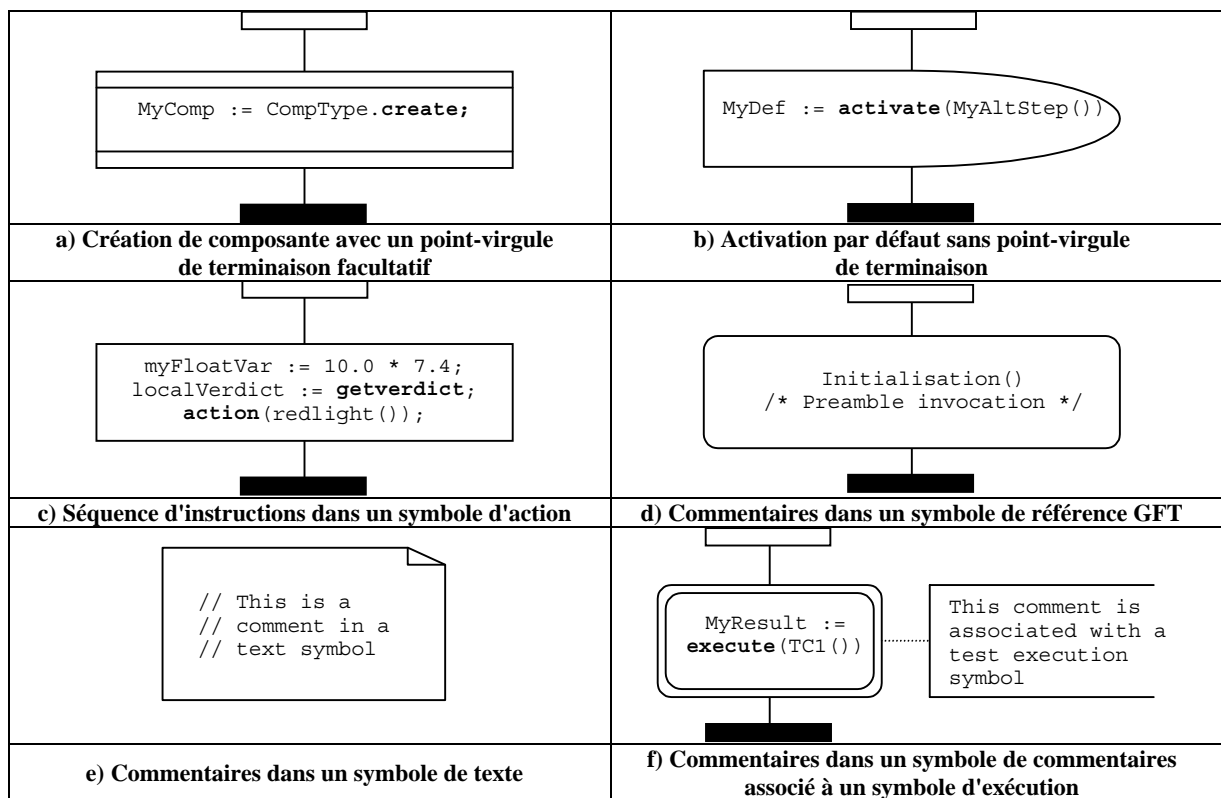


Figure 14/Z.142 – Exemples d'application des règles générales de représentation graphique

11.2 Invocation de diagrammes GFT

Le présent paragraphe décrit la façon dont les différents types de diagrammes GFT sont invoqués. Etant donné qu'il n'y a pas d'instruction relative à l'exécution de la partie commande en notation TTCN-3 (car ceci est comparable à l'exécution d'un programme via le module main et que cela se trouve hors du domaine d'application de la notation TTCN-3), le présent paragraphe porte sur l'exécution de tests élémentaires, de fonctions et variantes.

11.2.1 Exécution de tests élémentaires

L'exécution de tests élémentaires est représentée par l'utilisation du symbole d'exécution de test élémentaire (voir la Figure 15). La syntaxe de l'instruction **execute** est placée dans ce symbole. Ce dernier peut comprendre:

- une instruction **execute** pour un test élémentaire ayant des paramètres facultatifs et une surveillance temporelle;
- l'affectation facultative du verdict renvoyé à une variable **verdicttype**;
- la déclaration en ligne facultative de la variable **verdicttype**.

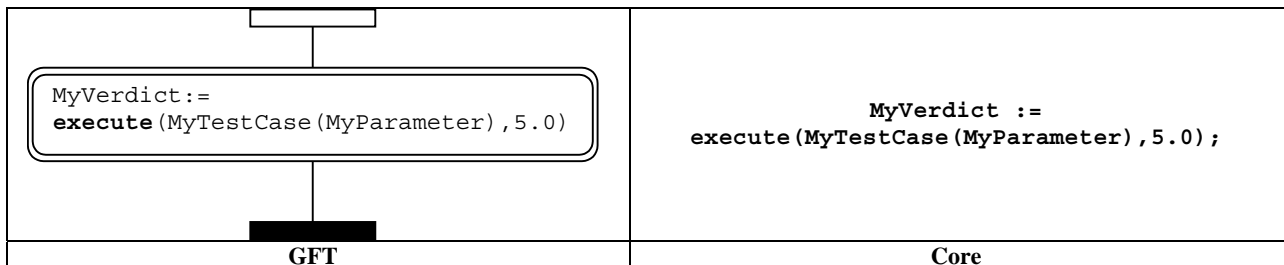


Figure 15/Z.142 – Exécution d'un test élémentaire

11.2.2 Invocation de fonctions

L'invocation de fonctions est représentée par le symbole de référence (Figure 16), sauf lorsqu'il s'agit de fonctions externes et prédéfinies (Figure 17) ou lorsqu'une fonction est appelée dans un élément de langage TTCN-3 ayant une représentation GFT (Figure 18).

La syntaxe de l'invocation de fonction figure dans le symbole de référence. Ce dernier peut comprendre:

- l'invocation d'une fonction ayant des paramètres facultatifs;
- l'affectation facultative de la valeur renvoyée à une variable;
- une déclaration en ligne facultative de la variable.

Le symbole de référence n'est utilisé que pour des fonctions définies par l'utilisateur dans le module courant. Il ne doit pas être utilisé pour des fonctions externes ou des fonctions TTCN-3 prédéfinies, qui doivent être représentées sous leur forme textuelle dans un symbole d'action (Figure 17) ou dans d'autres symboles GFT (voir l'exemple de la Figure 18).

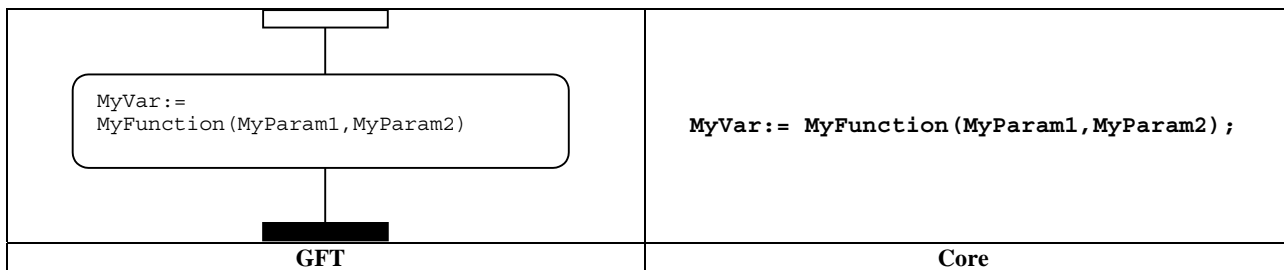


Figure 16/Z.142 – Invocation d'une fonction définie par l'utilisateur

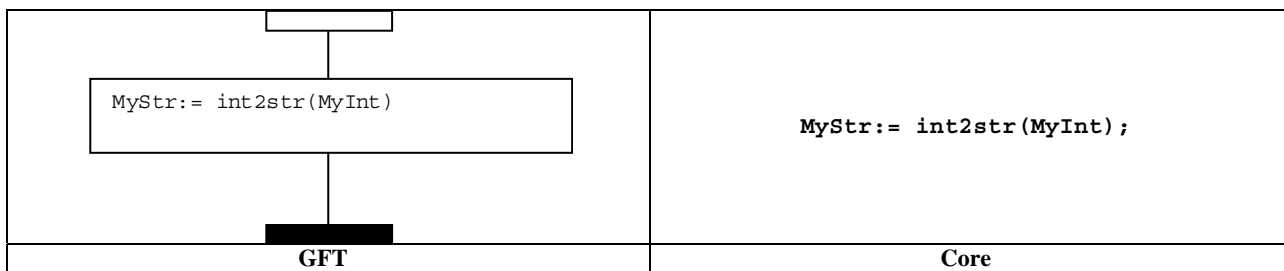


Figure 17/Z.142 – Invocation d'une fonction prédéfinie/externe

Les fonctions appelées dans une structure TTCN-3 et associées à un symbole GFT sont représentées par un texte dans ce symbole.

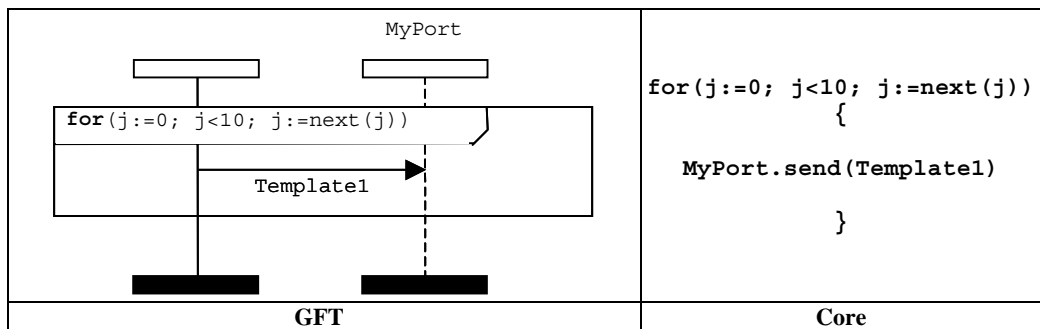


Figure 18/Z.142 – Invocation d'une fonction définie par l'utilisateur dans un symbole GFT

11.2.3 Invocation de variantes

L'invocation de variantes est représentée par l'utilisation du symbole de référence (voir la Figure 19). La syntaxe de l'invocation de variante est placée dans ce symbole. Celui-ci peut comprendre l'invocation d'une variante ayant des paramètres facultatifs. Il doit être utilisé seulement dans le cadre d'un comportement à variantes, l'invocation d'une variante devant être l'un des opérandes des instructions d'alternative (voir également la Figure 32 au § 11.5.2).

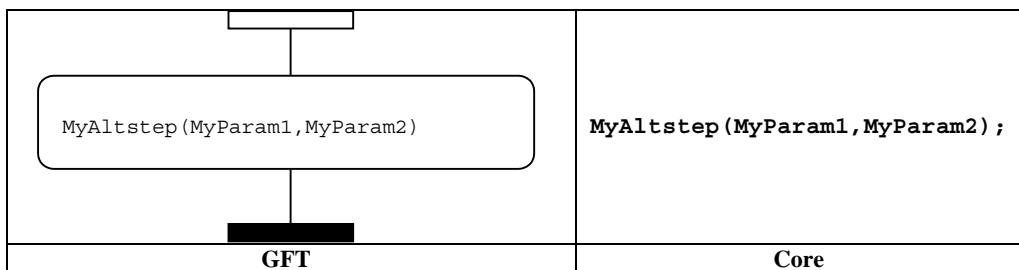


Figure 19/Z.142 – Invocation d'une variante

Une autre possibilité consiste à invoquer implicitement des variantes via l'activation de comportements par défaut. On se référera au § 11.6.2 pour de plus amples détails.

11.3 Déclarations

La notation TTCN-3 permet de déclarer et d'initialiser des temporisateurs, des constantes et des variables au début des blocs d'instructions. Le format GFT utilise la syntaxe du langage noyau TTCN-3 pour les déclarations dans plusieurs symboles. Le type d'un symbole dépend de la spécification de l'initialisation: ainsi, une variable de type **default** initialisé au moyen d'une opération **activate** doit être spécifiée dans un symbole de comportement par défaut (voir le § 11.6).

11.3.1 Déclaration de temporisateurs, de constantes et de variables dans des symboles d'action

Les déclarations suivantes doivent être faites dans des symboles d'action:

- les déclarations de temporisateurs;
- les déclarations de variables sans initialisation;
- les déclarations de variables et de constantes avec initialisation;
 - si l'initialisation n'est pas réalisée par des fonctions comprenant des fonctions de communication; ou
 - si une déclaration est:
 - de type composante non initialisé par une opération **create**;
 - de type **default** non initialisé par une opération **activate**;
 - de type **verdicttype** non initialisé par une instruction **execute**;
 - de type de base simple;
 - de type chaîne de base;

- de type **anytype**;
- de type port;
- de type **address**; ou
- d'un type structuré défini par l'utilisateur et dont les champs sont conformes à toutes les restrictions mentionnées ici sous le point "déclarations de variables et de constantes avec initialisation".

NOTE – On se référera au Tableau 3/Z.140 [1] pour un aperçu général des types en notation TTCN-3.

Une séquence de déclarations à faire dans plusieurs symboles d'action peut figurer dans un seul symbole d'action et ne doit donc pas nécessairement être réalisée dans différents symboles d'action. Les Figures 20 a) et 20 b) donnent des exemples de déclarations faites dans des symboles d'action.

11.3.2 Déclaration de constantes et de variables dans les symboles d'expression en ligne

Les déclarations de constantes et de variables d'un type de composante, initialisées dans une instruction **if-else**, **for**, **while**, **do-while**, **alt** ou **interleave**, doivent figurer dans un même symbole d'expression en ligne.

11.3.3 Déclaration de constantes et de variables dans des symboles de création

Les déclarations de constantes et de variables d'un type de composante, initialisées au moyen d'opérations **create**, doivent être faites dans un symbole de création. Contrairement aux déclarations dans les symboles d'action, chaque déclaration initialisée au moyen d'une opération **create** doit être faite dans un symbole de création distinct. Un exemple de déclaration de variable dans un symbole de création est donné sur la Figure 20 c).

11.3.4 Déclaration de constantes et de variables dans des symboles de comportement par défaut

Les déclarations de constantes et de variables de type **default**, initialisées au moyen d'opérations **activate**, doivent être faites dans un symbole de comportement par défaut. Contrairement au cas des déclarations effectuées dans des symboles d'action, chaque déclaration initialisée au moyen d'une opération **activate** doit être faite dans un symbole de comportement par défaut distinct. Un exemple de déclaration de variable dans un symbole de comportement par défaut est donné sur la Figure 20 d).

11.3.5 Déclaration de constantes et de variables dans des symboles de référence

Les déclarations de constantes et de variables qui sont initialisées au moyen d'une fonction comprenant des opérations de communication doivent être faites dans des symboles de référence. Contrairement au cas des déclarations effectuées dans des symboles d'action, chaque déclaration initialisée au moyen d'une fonction comprenant des opérations de communication doit être faite dans un symbole de référence distinct. Un exemple de déclaration de variable dans un symbole de référence est donné sur la Figure 20 e).

11.3.6 Déclaration de constantes et de variables dans les symboles d'exécution de test élémentaire

Les déclarations de constantes et de variables de type **verdicttype**, qui sont initialisées au moyen d'instructions **execute**, doivent être faites dans des symboles d'exécution de test élémentaire. Contrairement au cas des déclarations effectuées dans des symboles d'action, chaque déclaration initialisée au moyen d'une instruction **execute** doit être faite dans un symbole d'exécution de test élémentaire distinct. Un exemple de déclaration de variable dans un symbole d'exécution de test élémentaire est donné sur la Figure 20 f).

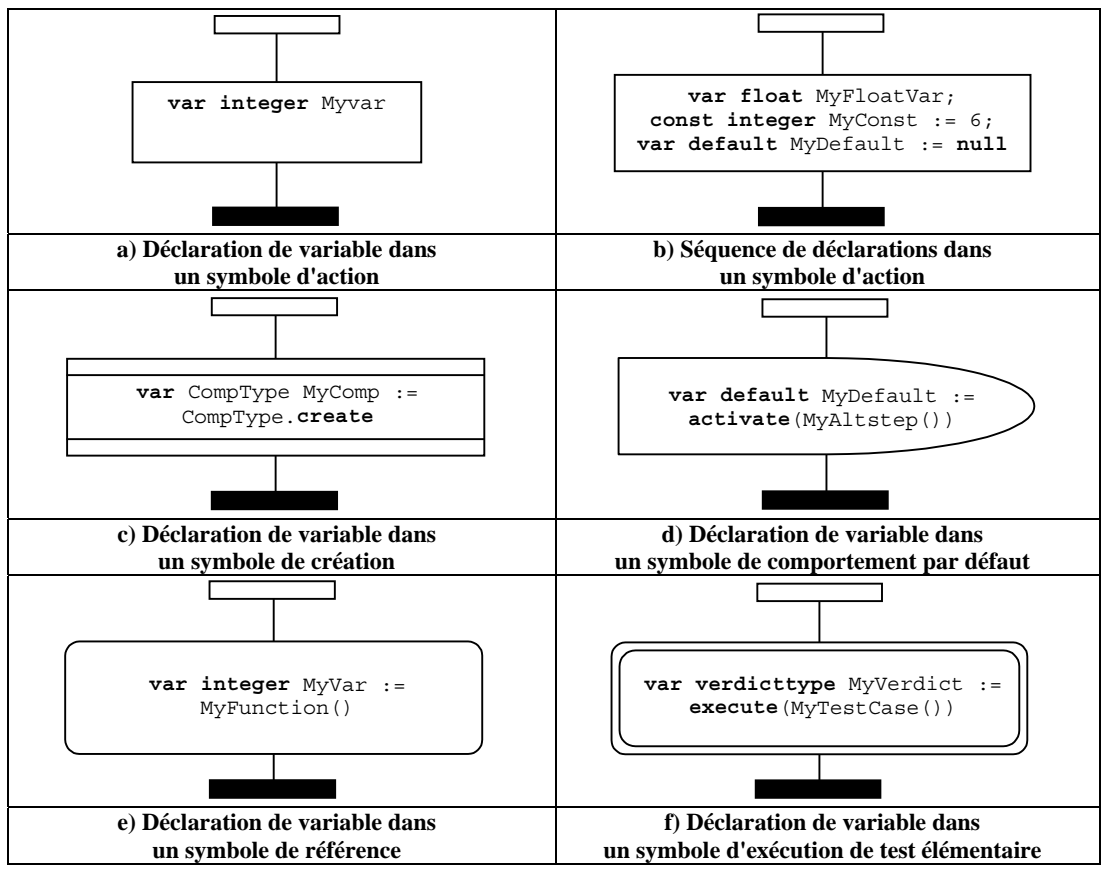


Figure 20/Z.142 – Exemples de déclarations en format GFT

11.4 Instructions de programme de base

Les instructions de programme de base sont des expressions, des affectations, des opérations, des structures de boucle, etc. Toutes les instructions de programme de base peuvent être utilisées dans des diagrammes GFT pour la partie commande, les tests élémentaires, les fonctions et les variantes.

Le format GFT ne fournit aucune représentation graphique pour les expressions et les affectations. Ces dernières sont indiquées par du texte là où elles doivent être utilisées. Des représentations graphiques existent pour les instructions **log**, **label**, **goto**, **if-else**, **for**, **while** et **do-while**.

11.4.1 Instruction log

L'instruction **log** doit être représentée dans un symbole d'action (voir la Figure 21).

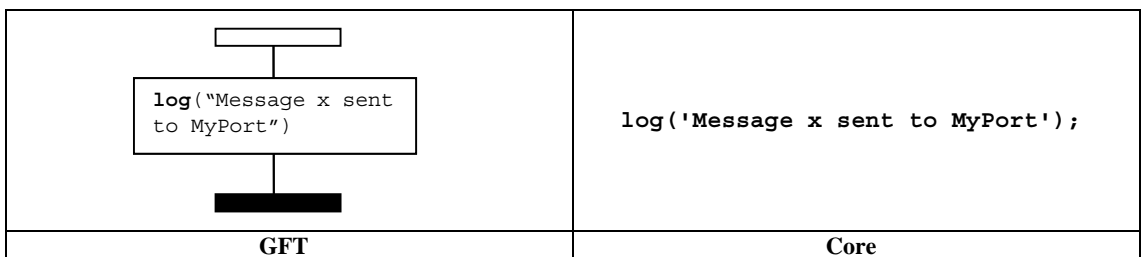


Figure 21/Z.142 – Instruction log

11.4.2 Instruction label

L'instruction **label** doit être représentée à l'aide d'un symbole d'étiquette attaché à une instance de composante. La Figure 22 donne un exemple simple d'étiquette **label** appelée MyLabel.

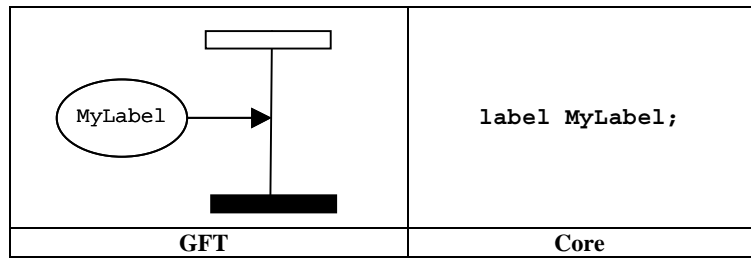


Figure 22/Z.142 – Instruction label

11.4.3 Instruction goto

L'instruction `goto` doit être représentée à l'aide d'un symbole aller à. Elle doit être placée à la fin d'une instance de composante ou à la fin d'un opérande dans un symbole d'expression en ligne. La Figure 23 donne un exemple simple d'instruction `goto`.

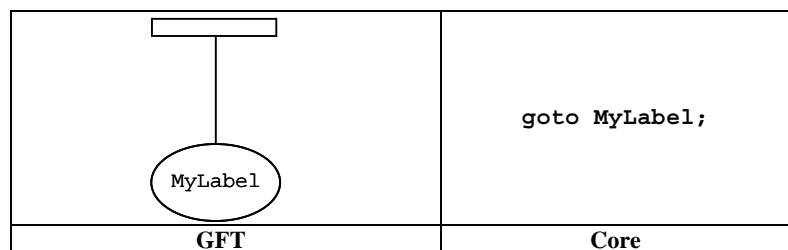


Figure 23/Z.142 – Instruction goto

11.4.4 Instruction if-else

L'instruction `if-else` doit être représentée par un symbole d'expression en ligne étiqueté par le mot clé `if` et une expression booléenne définie au § 19.6/Z.140 [1]. Le symbole d'expression en ligne `if-else` peut contenir un ou deux opérandes, séparés par une ligne pointillée. La Figure 24 donne l'exemple d'une instruction `if` comprenant un seul opérande, exécuté lorsque l'expression booléenne `x>1` est évaluée à Vrai. La Figure 25 illustre le cas d'une instruction `if-else` pour laquelle l'opérande supérieur est exécuté si l'expression booléenne `x>1` est évaluée à vrai, l'opérande inférieur étant exécuté si l'expression booléenne est évaluée à Faux.

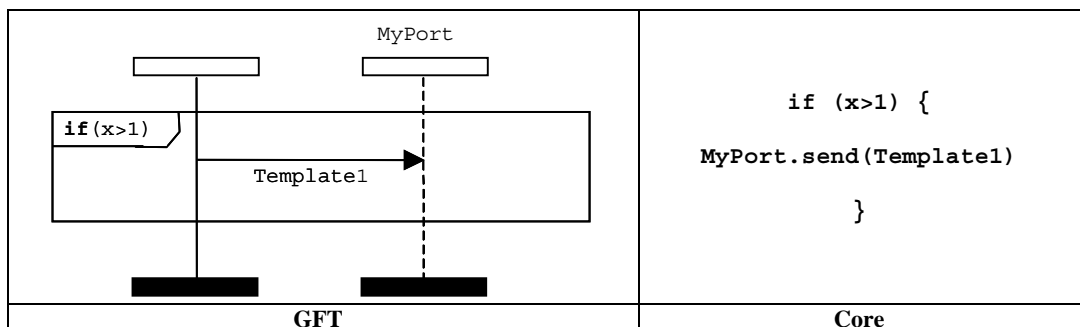


Figure 24/Z.142 – Instruction if

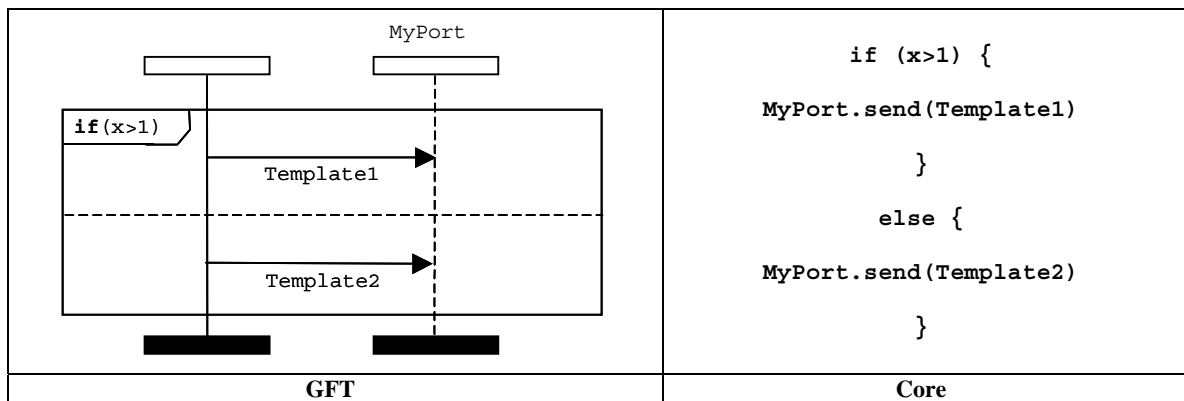


Figure 25/Z.142 – Instruction if-else

11.4.5 Instruction for

L'instruction **for** doit être représentée par un symbole d'expression en ligne étiqueté par le mot clé **for** défini au § 19.7/Z.140 [1]. Le corps de l'instruction **for** doit être constitué par l'opérande du symbole de l'expression en ligne **for**. La Figure 26 représente une boucle **for** simple dans laquelle la variable de boucle est déclarée et initialisée dans l'instruction **for**.

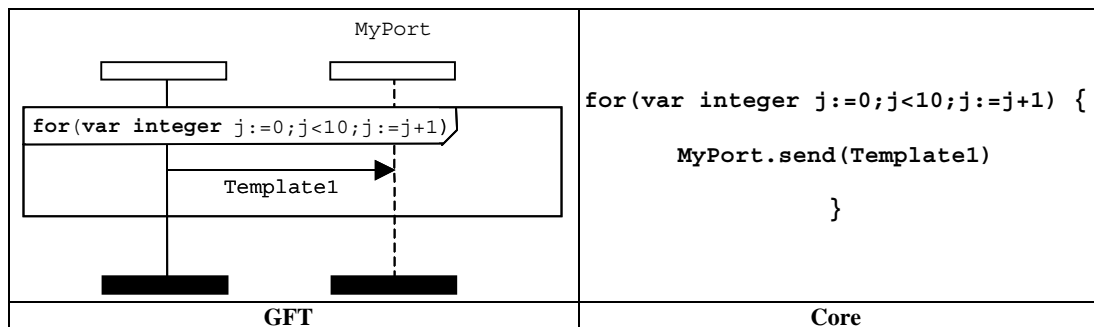


Figure 26/Z.142 – Instruction for

11.4.6 Instruction while

L'instruction **while** doit être représentée par un symbole d'expression en ligne étiqueté par le mot clé **while** défini au § 19.8/Z.140 [1]. Le corps de l'instruction **while** doit être constitué par l'opérande du symbole d'expression en ligne **while**. La Figure 27 donne un exemple d'instruction **while**.

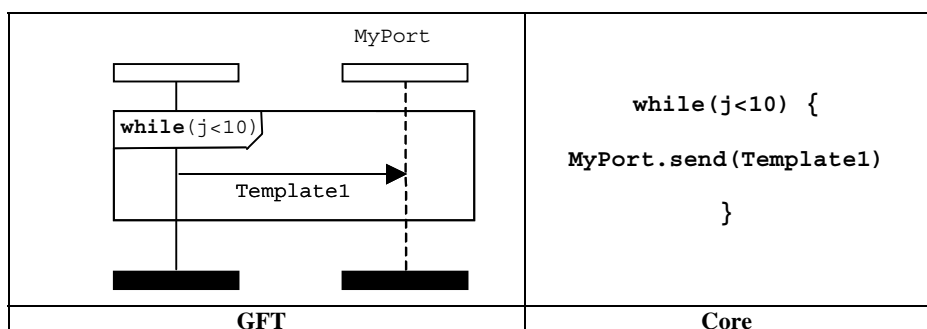


Figure 27/Z.142 – Instruction while

11.4.7 Instruction do-while

L'instruction **do-while** doit être représentée par un symbole d'expression en ligne étiqueté par le mot clé **do-while** défini au § 19.9/Z.140 [1]. Le corps de l'instruction **do-while** doit être constitué par l'opérande du symbole d'expression en ligne **do-while**. La Figure 28 donne un exemple d'instruction **do-while**.

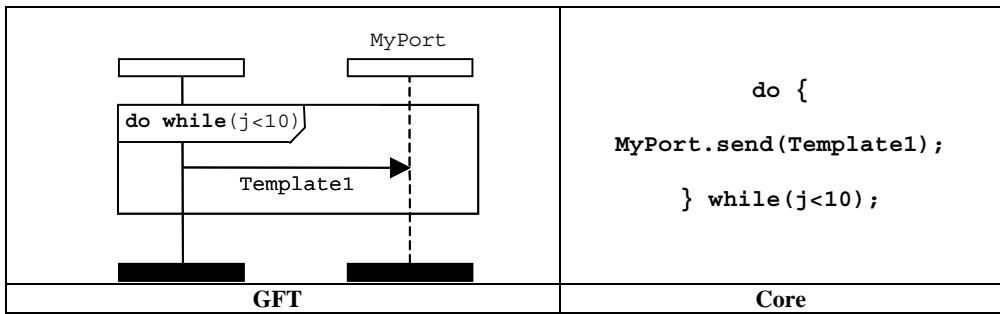


Figure 28/Z.142 – Instruction do-while

11.5 Instructions de programmation comportementales

Les instructions comportementales peuvent être utilisées dans des tests élémentaires, des fonctions et des variantes, la seule exception étant l'instruction de retour, qui ne peut être utilisée que dans des fonctions. Le comportement de test peut être exprimé séquentiellement, sous la forme d'un ensemble de variantes ou à l'aide d'une instruction d'entrelacement. Les instructions return et repeat sont utilisées pour commander le flux de comportement.

11.5.1 Comportement séquentiel

Le comportement séquentiel est représenté par l'ordre des événements placés sur une instance de composante de test. Les événements sont considérés dans l'ordre descendant, les événements les plus proches de la tête du symbole de l'instance de composante étant évalués en premier. La Figure 29 illustre le cas simple d'une composante de test qui évalue d'abord l'expression figurant dans le symbole d'action avant d'envoyer un message au port MyPort.

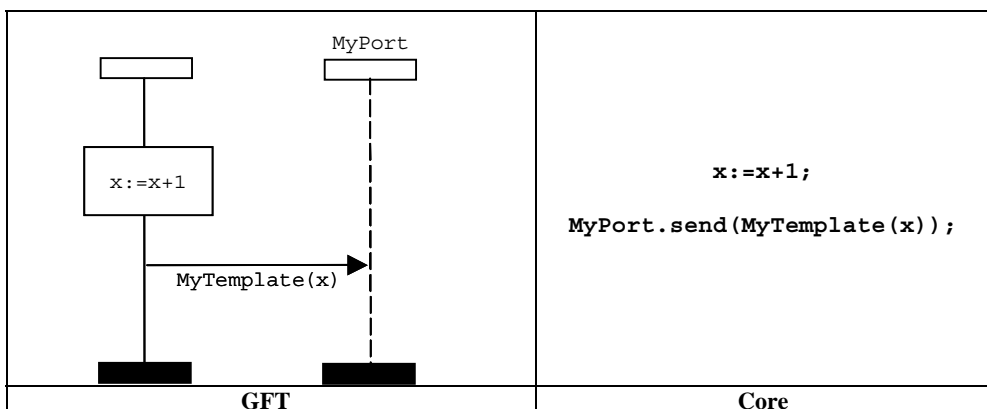


Figure 29/Z.142 – Comportement séquentiel

Le séquençage peut aussi être décrit en utilisant des références à des tests élémentaires, à des fonctions et à des variantes. Dans ce cas, l'ordre suivant lequel les références sont placées le long de l'axe d'instance composante détermine l'ordre d'évaluation. La Figure 30 représente un diagramme GFT simple dans lequel les fonctions MyFunction1 et MyFunction2 sont successivement appelées.

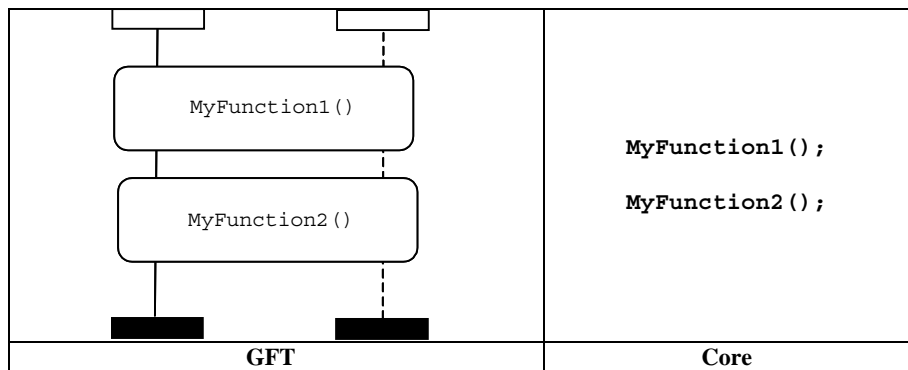


Figure 30/Z.142 – Séquencement utilisant des références

11.5.2 Comportement à variantes

Le comportement à variantes doit être représenté en utilisant un symbole d'expression en ligne comprenant le mot clé **alt** placé dans le coin supérieur gauche. Les opérandes du comportement à variantes doivent être séparés par des lignes pointillées. Ils sont évalués dans l'ordre descendant.

Il convient de noter qu'une expression en ligne de variantes devrait toujours prendre en compte toutes les instances de port, si des opérateurs de communication sont impliqués. La Figure 31 illustre le cas d'un comportement à variantes suivant lequel il y a réception d'un événement message avec la valeur définie par `Template1`, ou réception d'un événement de message avec la valeur définie par `Template2`. L'invocation d'une variante dans l'expression en ligne à variantes est illustrée sur la Figure 32.

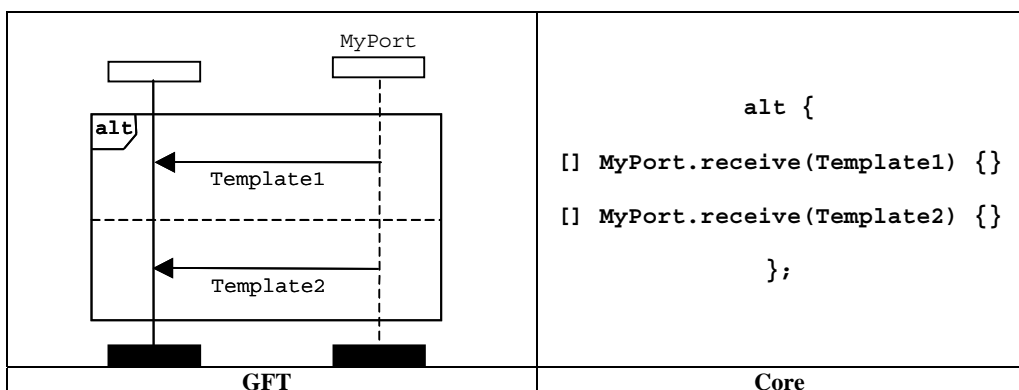


Figure 31/Z.142 – Instruction de comportement à variantes

Il est en outre possible d'appeler une variante en tant que seul événement dans un opérande de variante. Cet appel se fera graphiquement par l'utilisation d'un symbole de référence (voir le § 11.2.3).

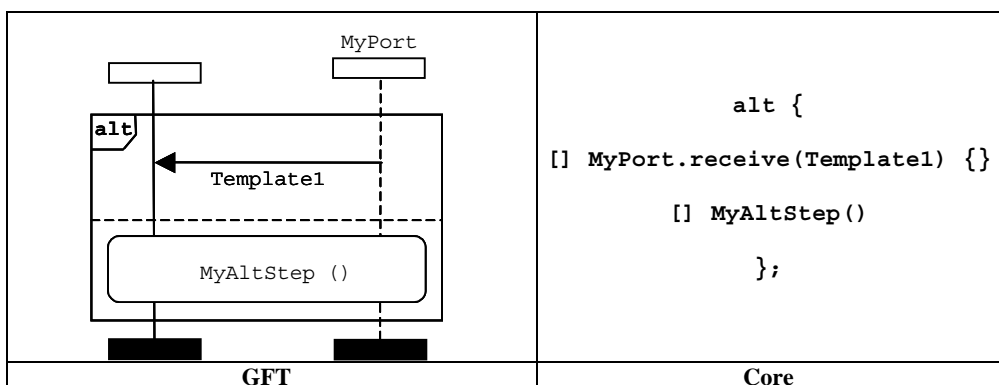


Figure 32/Z.142 – Comportement à variantes avec invocation d'une variante

11.5.2.1 Activer/désactiver une alternative

Il est possible de désactiver/d'activer un opérande de variante à l'aide d'une expression booléenne figurant dans un symbole de condition placé sur l'instance de composante de test. La Figure 33 donne l'exemple d'une instruction d'alternative simple dans laquelle le premier opérande dépend de l'expression $x > 1$ et le second de l'expression $x \leq 1$.

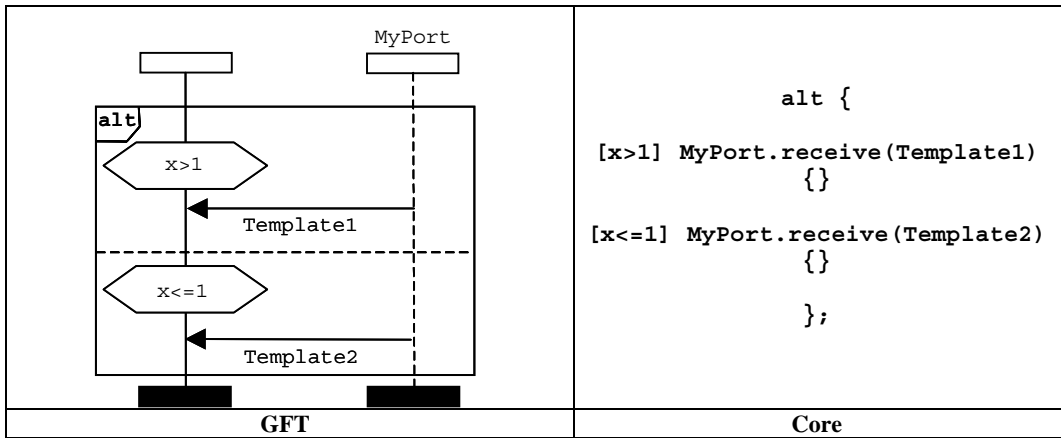


Figure 33/Z.142 – Activation/désactivation d'une alternative

11.5.2.2 Branche else dans des alternatives

La branche **else** doit être indiquée en utilisant un symbole de condition placé sur l'axe de l'instance de composante de test étiqueté par le mot clé **else**. La Figure 34 donne l'exemple d'une instruction de variante simple dans lequel le second opérande est une branche **else**.

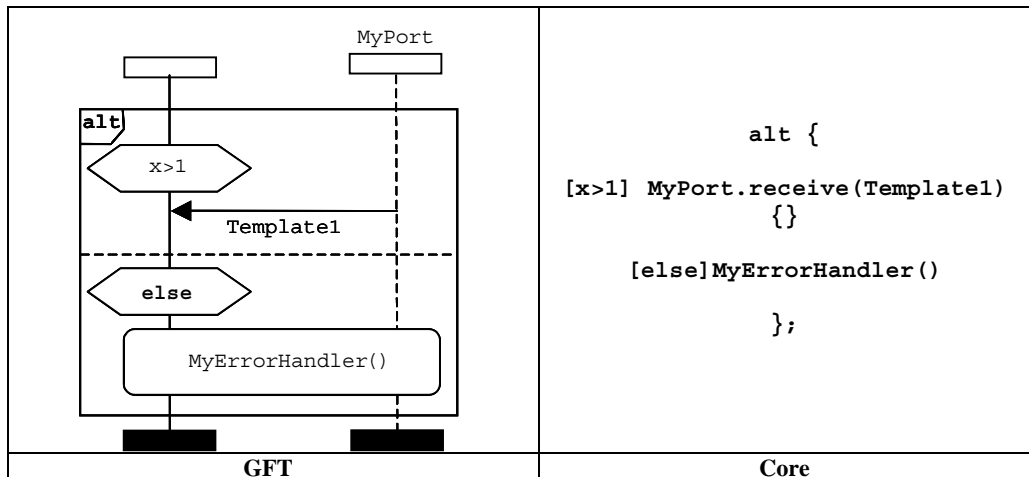


Figure 34/Z.142 – Branche else dans une alternative

Il convient de noter que le symbole de référence dans une branche else devrait toujours prendre en compte toutes les instances de port, si des opérations de communication interviennent.

La réévaluation d'une instruction alt peut être spécifiée en utilisant une instruction repeat, représentée par le symbole repeat (voir le § 11.5.3).

L'invocation des variantes dans des alternatives est représentée en utilisant le symbole de référence (voir le § 11.2.3).

11.5.3 Instruction repeat

L'instruction **repeat** doit être représentée par un symbole de répétition. Ce symbole ne doit être utilisé que comme dernier événement d'un opérande de variante dans une instruction **alt** ou comme dernier événement d'un opérande de variante supérieur dans une définition de variante. La Figure 35 donne l'exemple d'une instruction d'alternative dans laquelle le second opérande, après avoir reçu avec succès un message avec la valeur `Template2` voulue, provoque la répétition de l'alternative.

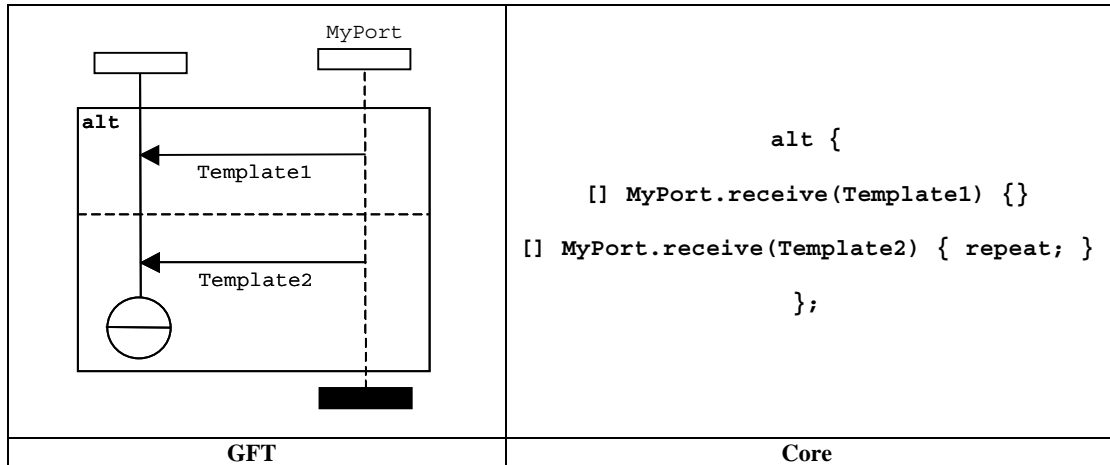
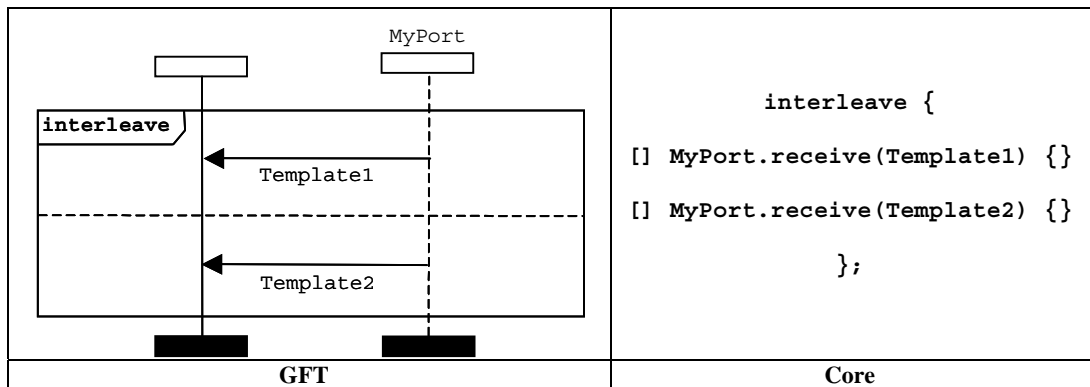


Figure 35/Z.142 – Répétition dans une alternative

11.5.4 Comportement entrelacé

Un comportement d'entrelacement doit être représenté en utilisant un symbole d'expression en ligne ayant le mot clé **interleave** dans le coin supérieur gauche (voir la Figure 36). Les opérandes doivent être séparés par des lignes pointillées. Ils sont évalués suivant l'ordre descendant.



NOTE – Une expression en ligne d'entrelacement devrait toujours prendre en compte toutes les instances de port si des opérateurs de communication sont partie prenante.

Figure 36/Z.142 – Instruction d'entrelacement

11.5.5 Instruction return

L'instruction **return** doit être représentée en utilisant un symbole de retour, qui peut le cas échéant être associé à une valeur de retour. Un symbole de retour ne doit être utilisé que dans un diagramme GFT de fonction. Il ne doit être utilisé que comme dernier événement d'une instance de composante ou comme dernier événement d'un opérande dans un symbole d'expression en ligne. La Figure 37 donne l'exemple d'une fonction simple utilisant une instruction de retour sans retour d'une valeur, la Figure 38 illustrant le cas d'une fonction qui retourne une valeur.

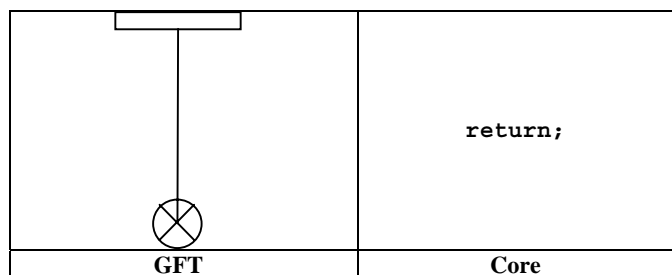


Figure 37/Z.142 – Symbole de retour sans utilisation d'une valeur de retour

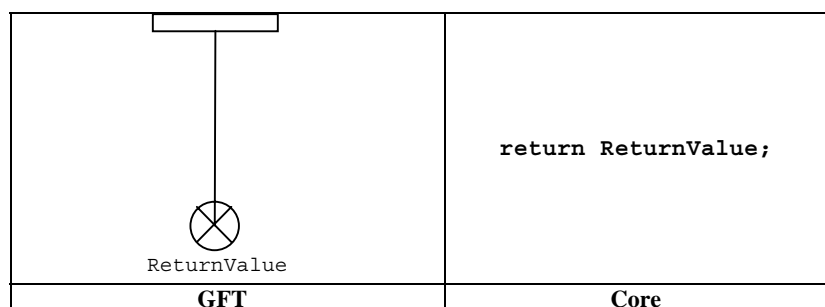


Figure 38/Z.142 – Symbole de retour avec utilisation d'une valeur de retour

11.6 Gestion des comportements par défaut

Le format GFT fournit une représentation graphique pour l'activation et la désactivation des comportements par défaut (voir le § 21/Z.140 [1]).

11.6.1 Références par défaut

Les variables de type **default** peuvent être déclarées dans un symbole d'action ou dans un symbole de comportement par défaut au sein d'une instruction activée. Les paragraphes 11.3.1 et 11.3.4 illustrent la façon dont une variable nommée `MyDefaultType` est déclarée en format GFT.

11.6.2 Opération activée

L'activation de comportements par défaut doit être représentée par le placement de l'instruction **activate** dans un symbole de comportement par défaut (voir la Figure 39).

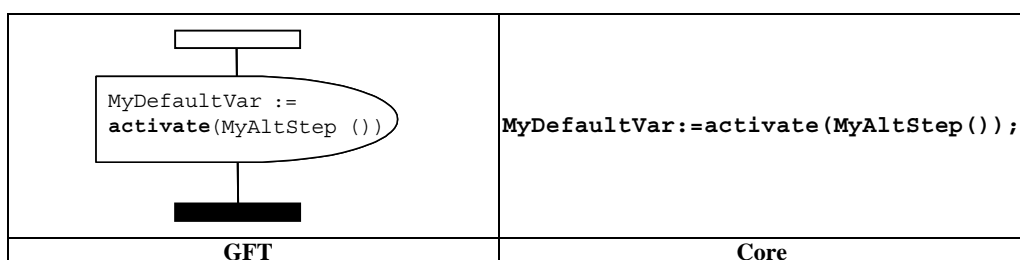


Figure 39/Z.142 – Activation des comportements par défaut

11.6.3 Opération désactivée

La désactivation de comportements par défaut doit être représentée par le placement de l'instruction **deactivate** dans un symbole de comportement par défaut (voir la Figure 40). Si aucun opérande n'est transmis à l'instruction **deactivate**, tous les comportements par défaut sont désactivés.

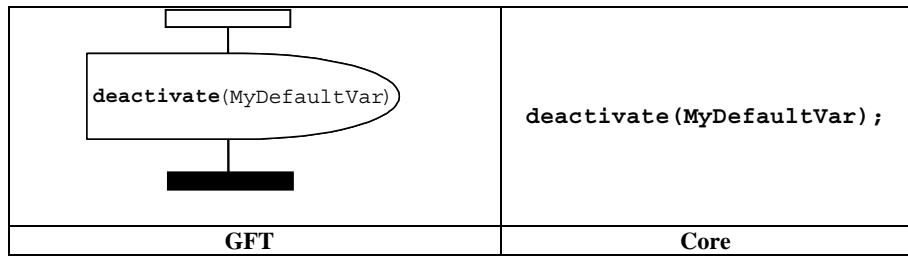


Figure 40/Z.142 – Désactivation de comportements par défaut

11.7 Opérations de configuration

Les opérations de configuration sont utilisées pour mettre en place et commander des composantes de test. Elles ne doivent être utilisées que pour les diagrammes GFT de test élémentaire, de fonction ou de variante.

Les opérations **mtc**, **self** et **system** n'ont pas de représentation graphique; elles sont indiquées sous une forme textuelle là où elles sont utilisées.

Le format GFT ne fournit pas de représentation graphique de l'opération d'exécution (car il s'agit d'une expression booléenne), qui est indiquée par du texte là où elle est utilisée.

11.7.1 Opération de création

L'opération **create** doit être représentée dans un symbole de création, rattaché à l'instance de composante de test qui réalise l'opération (voir la Figure 41). Le symbole de création contient l'instruction **create**.

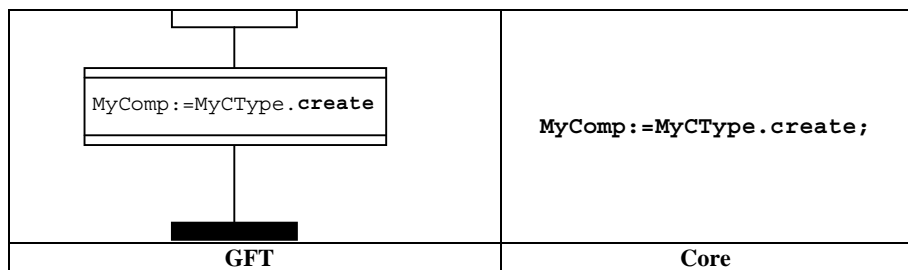


Figure 41/Z.142 – Opération de création

11.7.2 Opérations de connexion et de mappage

Les opérations **connect** et **map** doivent être représentées dans un symbole de cadre d'action, rattaché à l'instance de composante de test qui réalise l'opération (voir la Figure 42). Le symbole de cadre d'action contient l'instruction **connect** ou **map**.

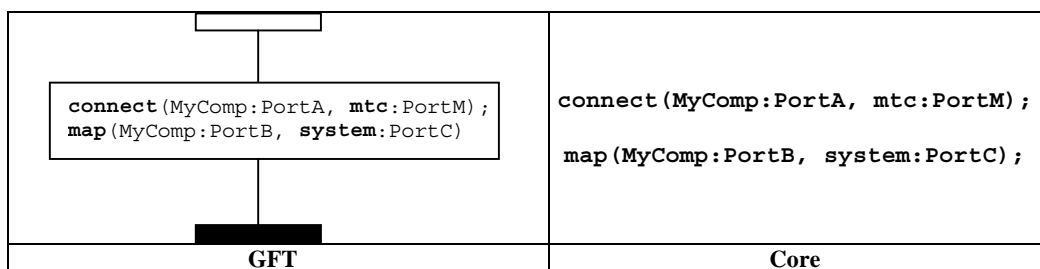


Figure 42/Z.142 – Opérations de connexion et de mappage

11.7.3 Opérations de déconnexion et de démappage

Les opérations **disconnect** et **unmap** doivent être représentées dans un symbole de cadre d'action, rattaché à l'instance de composante de test qui réalise l'opération (voir la Figure 43). Le symbole de cadre d'action contient l'instruction **disconnect** ou **unmap**.

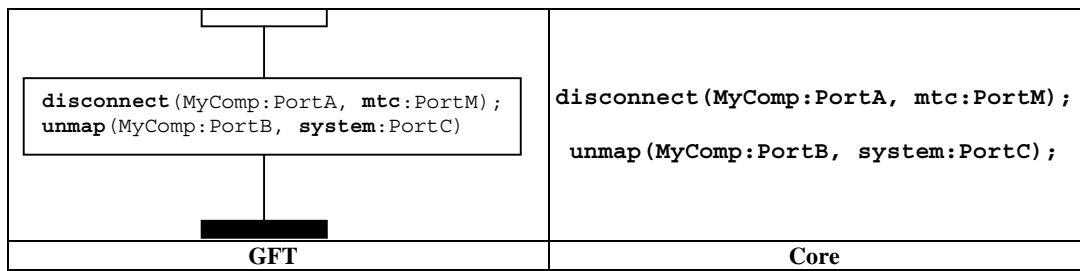


Figure 43/Z.142 – Opérations de déconnexion et de démappage

11.7.4 Opération de lancement de composantes de test

L'opération de lancement de composantes de test **start** doit être représentée dans le symbole de lancement **start**, rattaché à l'instance de composante de test qui réalise l'opération **start** (voir la Figure 44). Le symbole de lancement contient l'instruction **start**.

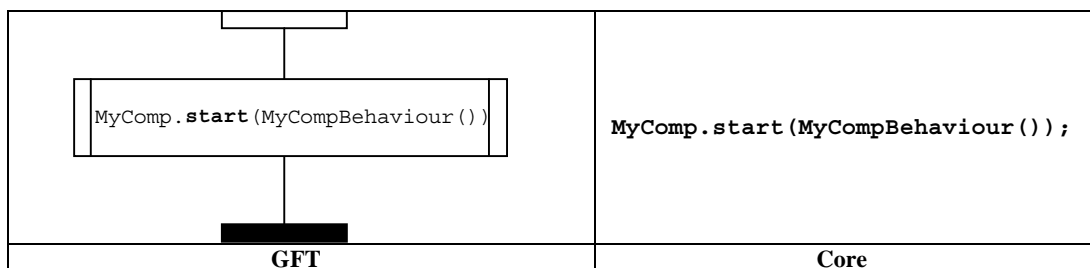


Figure 44/Z.142 – Opération start

11.7.5 Opérations d'arrêt d'exécution et d'arrêt de composantes de test

La notation TTCN-3 définit deux opérations d'arrêt: la partie commande du module et les composantes de test peuvent s'arrêter en utilisant des *opérations d'arrêt d'exécution*, ou une composante de test peut arrêter d'autres composantes de test en utilisant des *opérations d'arrêt de composantes de test*.

L'opération **stop** d'arrêt d'exécution doit être représentée par un symbole d'arrêt, rattaché à l'instance de composante de test qui réalise l'opération **stop** d'arrêt d'exécution (voir la Figure 45). Elle ne doit être utilisée que comme dernier événement d'une instance de composante ou comme dernier événement d'un opérande dans un symbole d'expression en ligne.

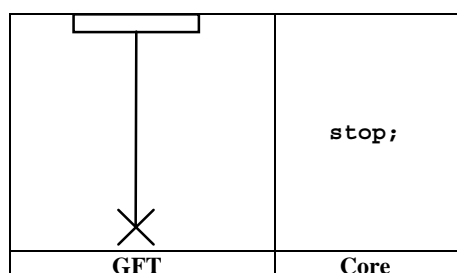


Figure 45/Z.142 – Opération d'arrêt d'exécution

L'opération **stop** d'arrêt de composantes de test doit être représentée par un symbole d'arrêt, rattaché à l'instance de composante de test qui réalise l'opération **stop** d'arrêt. Elle doit avoir une expression associée qui identifie la composante à arrêter (voir la Figure 46). La composante MTC peut arrêter toutes les composantes PTC en une seule étape en utilisant l'opération d'arrêt de composantes avec le mot clé **all** (voir la Figure 47 a)). Une composante PTC peut arrêter l'exécution du test en mettant fin à la composante MTC (voir la Figure 47 b)). L'opération **stop** d'arrêt de composantes de test doit être utilisée comme dernier événement d'une instance de composante ou comme dernier événement d'un opérande dans un symbole d'expression en ligne, si la composante s'arrête elle-même (**self.stop**, par exemple) ou si elle arrête l'exécution du test (**mtc.stop**, par exemple) (voir les Figures 47 c) et d)).

NOTE – Le symbole d'arrêt a une expression associée. Il n'est pas toujours possible de déterminer statistiquement si une opération d'arrêt de composants arrête l'instance qui exécute l'opération d'arrêt ou si elle arrête l'exécution du test.

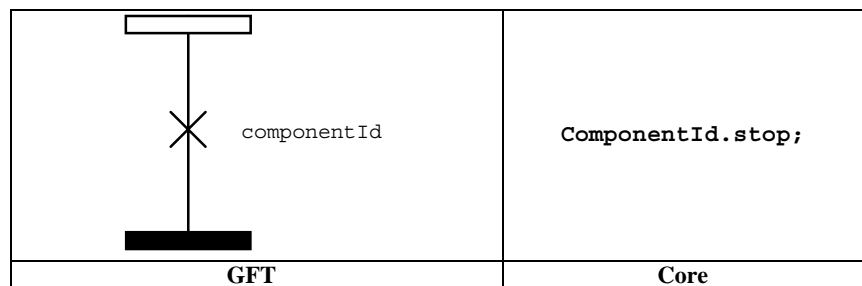


Figure 46/Z.142 – Opération d'arrêt de composants de test

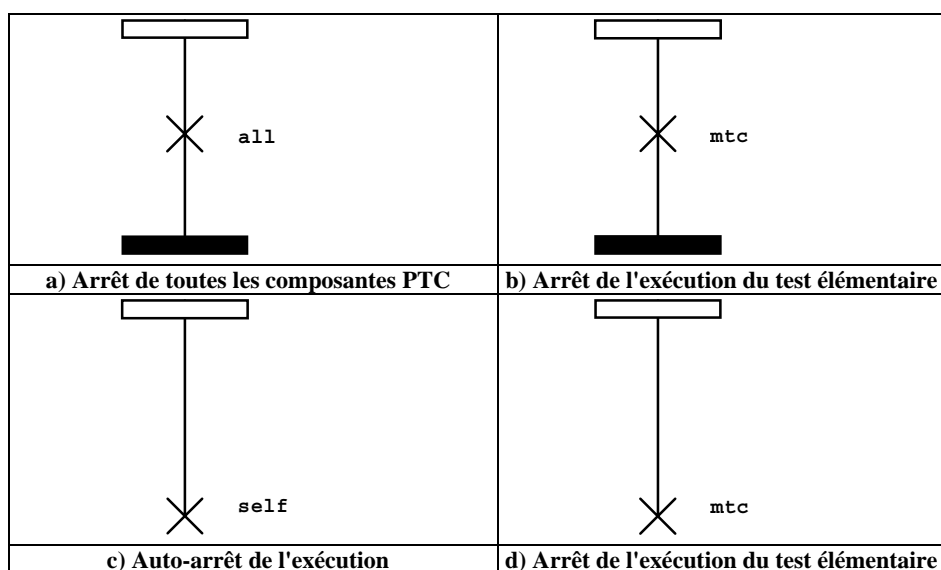


Figure 47/Z.142 – Utilisations particulières de l'opération d'arrêt de composants de test

11.7.6 Opération de fin d'exécution

L'opération **done** doit être représentée dans un symbole de condition, rattaché à l'instance de composante de test qui réalise l'opération **done** (voir la Figure 48). Le symbole de condition contient l'instruction **done**.

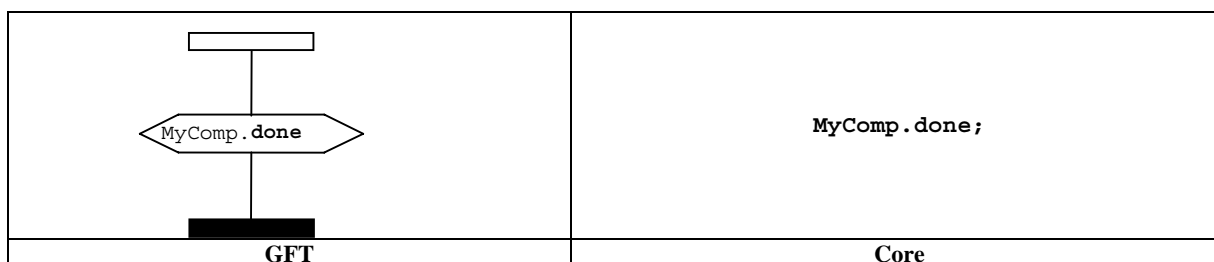


Figure 48/Z.142 – Opération de fin d'exécution

Les mots clés **any** et **all** peuvent être utilisés pour les opérations **running** et **done**, mais seulement à partir d'une instance de composante MTC. Ils n'ont pas de représentation graphique, mais sont indiqués sous une forme textuelle là où ils sont utilisés.

11.8 Opérations de communication

Les opérations de communication sont réparties en deux groupes:

- les *opérations d'envoi*: une composante de test envoie un message (opération **send**), appelle une procédure (opération **call**), répond à un appel accepté (opération **reply**) ou déclenche une exception (opération **raise**);
- les *opérations de réception*: une composante reçoit un message (opération **receive**), accepte un appel de procédure (opération **getcall**), reçoit une réponse provenant d'une procédure précédemment appelée (opération **getreply**) ou traite une exception (opération **catch**).

11.8.1 Format général des opérations d'envoi

Toutes les opérations d'envoi utilisent un symbole de message qui va de l'instance de composante de test réalisant l'opération d'envoi à l'instance de port vers laquelle l'information est transmise (voir la Figure 49).

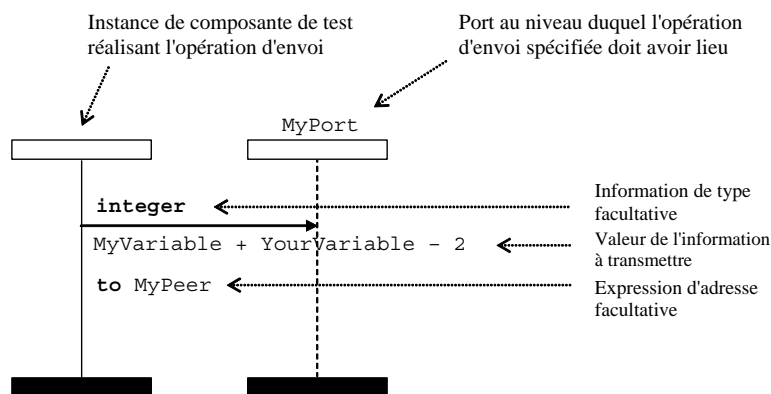


Figure 49/Z.142 – Format général des opérations d'envoi

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

La partie *envoi*:

- spécifie le port au niveau duquel l'opération spécifiée doit être effectuée;
- définit le type facultatif et la valeur de l'information à transmettre;
- fournit une expression d'adresse facultative qui désigne de manière univoque la correspondance de communication en cas de connexion point-multipoint.

Le port doit être représenté par une instance de port. Le nom d'une opération **call**, **reply** ou **raise** doit être indiqué au-dessus du symbole de message avant l'information de type facultative. L'opération **send** est implicite, ce qui signifie que le mot clé **send** ne doit pas être indiqué. La valeur de l'information à transmettre doit être placée sous le symbole de message. L'expression d'adresse facultative (indiquée par le mot clé **to**) doit être placée sous la valeur de l'information à transmettre.

La structure de l'opération **call** est plus spécifique. On se référera au § 11.8.4.1 pour plus de détails.

11.8.2 Format général des opérations de réception

Toutes les opérations de réception utilisent un symbole de message qui va de l'instance de port à l'instance de composante de test qui reçoit l'information (voir la Figure 50).

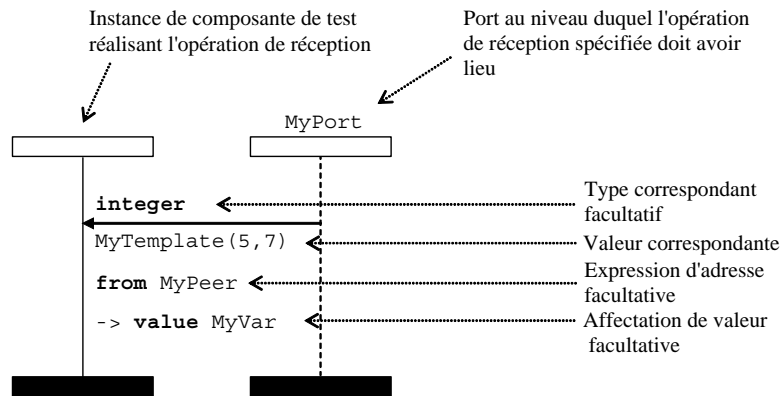


Figure 50/Z.142 – Format général des opérations de réception avec affectation d'adresse et de valeur

Une opération de réception se compose d'une partie *réception* et d'une partie *affectation* facultative.

La partie *réception*:

- spécifie le port au niveau duquel l'opération doit avoir lieu;
- définit une partie correspondante comprenant une information de type facultative et la valeur correspondante qui spécifie l'entrée acceptable satisfaisant à l'instruction;
- donne une expression d'adresse (facultative) qui désigne de manière univoque le correspondant de communication (en cas de connexions point à multipoint).

Le port doit être représenté par une instance de port. Le nom d'une opération **getcall**, **getreply** ou **catch** doit être indiqué au-dessus du symbole de message avant l'information de type (facultative). L'opération **receive** est indiquée de façon implicite, ce qui signifie que le mot clé **receive** ne doit pas être indiqué. La valeur correspondante pour la variable d'entrée acceptable doit être placée sous le symbole de message. L'expression d'adresse (facultative) (indiquée par le mot clé **from**) doit être placée sous la valeur de l'information à transmettre.

La partie affectation (facultative) (indiquée par '->') doit être placée sous la valeur de l'information à transmettre ou sous l'expression d'adresse si celle-ci est présente. Elle peut comprendre plusieurs lignes, de telle sorte, par exemple que les affectations de valeur, de paramètre et d'envoi occupent chacune une ligne (voir la Figure 51).

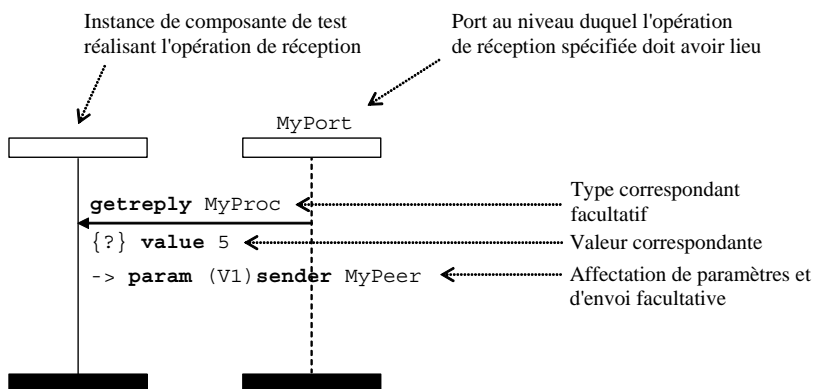


Figure 51/Z.142 – Format général des opérations de réception avec affectation de paramètres et d'envoi facultative

11.8.3 Communication en mode message

11.8.3.1 Opération d'envoi

L'opération d'envoi doit être représentée par un symbole de message sortant allant de la composante de test à l'instance de port. L'information de type facultative doit être placée au-dessus de la flèche de message. Le modèle (en ligne) doit être indiqué sous la flèche (voir les Figures 52 et 53).

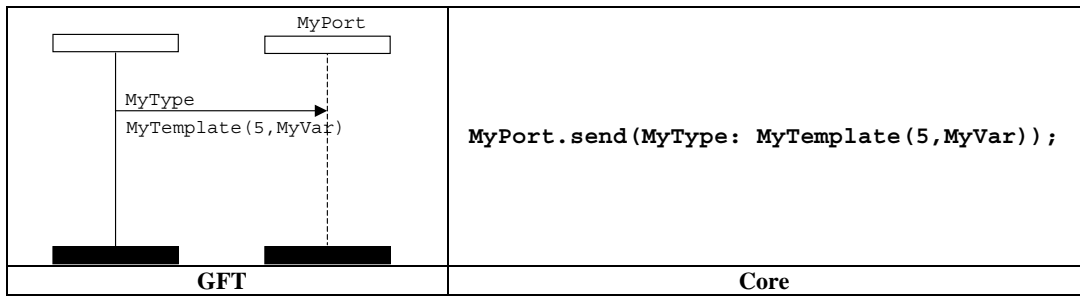


Figure 52/Z.142 – Opération d'envoi avec référence de modèle

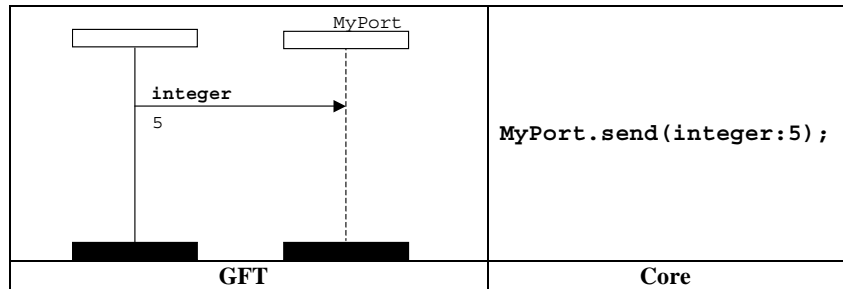


Figure 53/Z.142 – Opération d'envoi avec modèle en ligne

11.8.3.2 Opération de réception

L'opération de réception doit être représentée par une flèche de message entrant allant de l'instance de port à la composante de test. L'information de type facultative doit être placée au-dessus de la flèche de message. Le modèle (en ligne) doit être indiqué sous la flèche (voir les Figures 54 et 55).

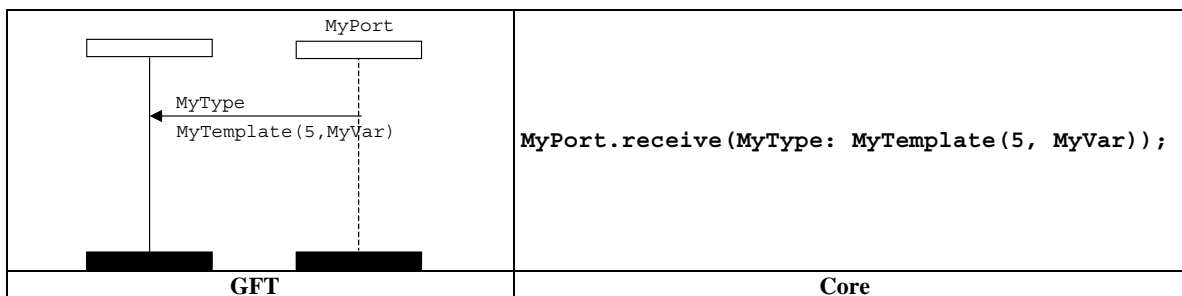


Figure 54/Z.142 – Opération de réception avec référence de modèle

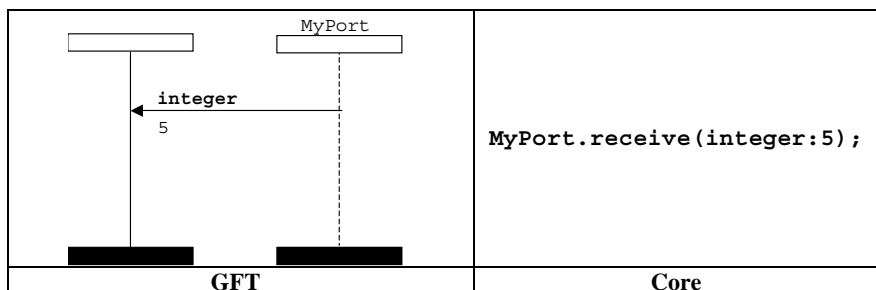


Figure 55/Z.142 – Opération de réception avec modèle en ligne

11.8.3.2.1 Réception de tout message

L'opération réception de tout message doit être représentée par une flèche de message entrant allant de l'instance de port à la composante de test sans adjonction d'aucune autre information (voir la Figure 56).

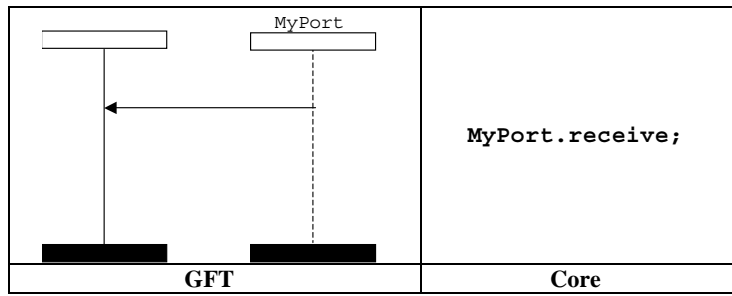


Figure 56/Z.142 – Réception de tout message

11.8.3.2 Réception sur tout port

L'opération réception sur tout port doit être représentée par un symbole d'obtention représentatif d'un port quelconque et dirigé vers la composante de test (voir la Figure 57).

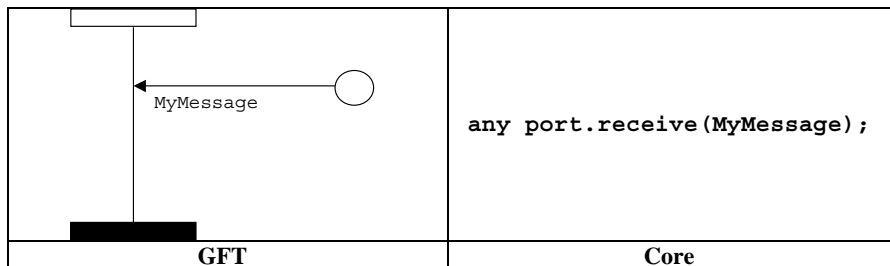


Figure 57/Z.142 – Réception sur tout port

11.8.3.3 Opération de déclenchement

L'opération de déclenchement doit être représentée par une flèche de message entrant allant de l'instance de port à la composante de test, le mot clé **trigger** étant placé au-dessus de la flèche avant l'information de type si celle-ci est présente (l'information de type facultative est donc placée au-dessus de la flèche après le mot clé **trigger**). Le modèle (en ligne) figure sous la flèche (voir les Figures 58 et 59).

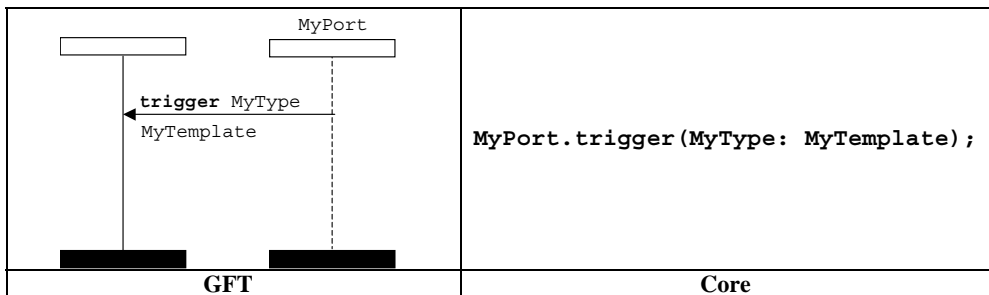


Figure 58/Z.142 – Opération de déclenchement avec modèle de référence

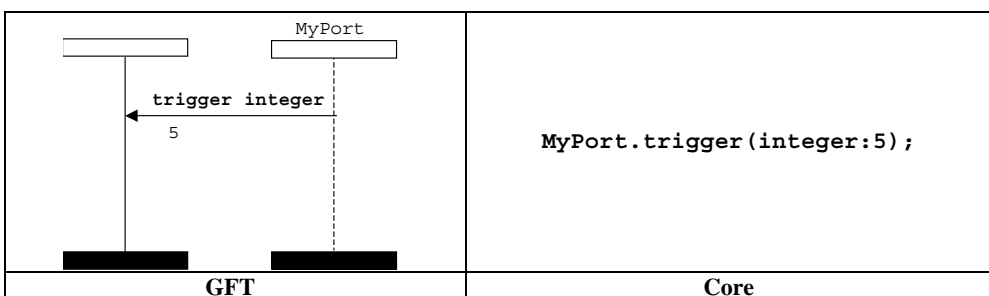


Figure 59/Z.142 – Opération de déclenchement avec modèle en ligne

11.8.3.3.1 Déclenchement à tout message

L'opération déclenchement à tout message doit être représentée par une flèche de message entrant allant de l'instance de port à la composante de test, le mot clé **trigger** étant placé au-dessus de la flèche sans adjonction d'aucune autre information (voir la Figure 60).

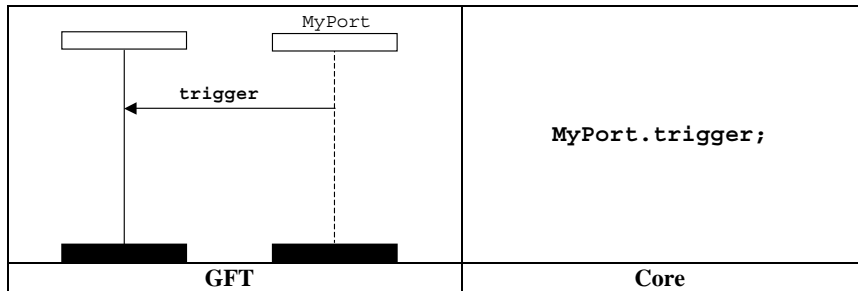


Figure 60/Z.142 – Opération déclenchement à tout message

11.8.3.3.2 Déclenchement pour tout port

L'opération déclenchement pour tout port doit être représentée par un symbole d'obtention représentatif d'un port quelconque et dirigé vers la composante de test (voir la Figure 61).

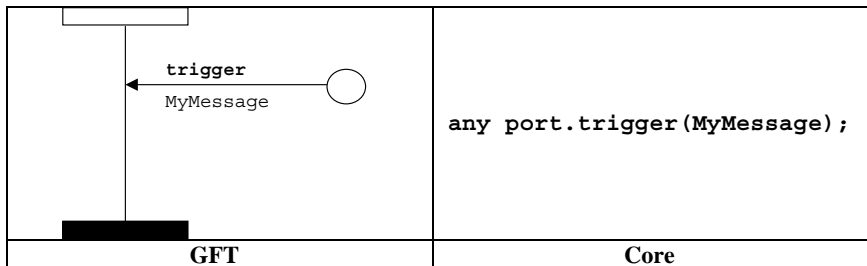


Figure 61/Z.142 – Opération déclenchement pour tout port

11.8.4 Communication en mode procédure

11.8.4.1 Opération d'appel

11.8.4.1.1 Procédures d'appel bloquant

L'opération **call** bloquant est représentée par un symbole de message sortant allant de la composante de test à l'instance de port, suivi d'une région de suspension placée sur la composante de test, le mot clé **call** étant placé au-dessus de la flèche avant la signature si celle-ci est présente. Le modèle (en ligne) figure sous la flèche (voir les Figures 62 et 63).

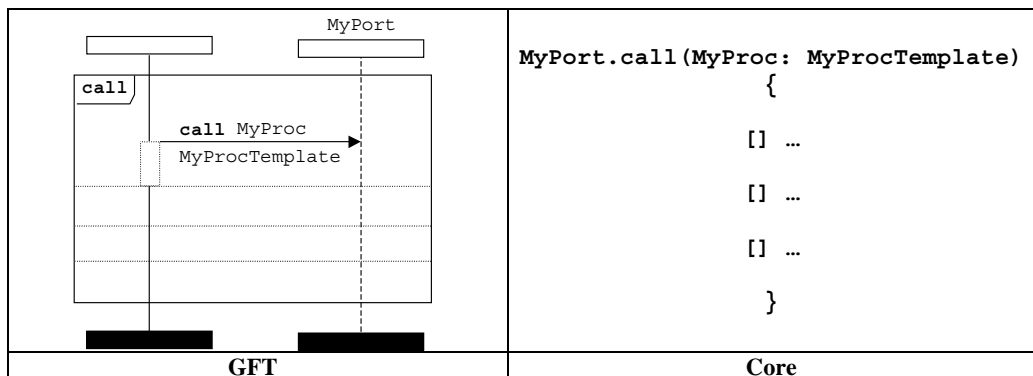


Figure 62/Z.142 – Opération d'appel bloquant avec référence de modèle

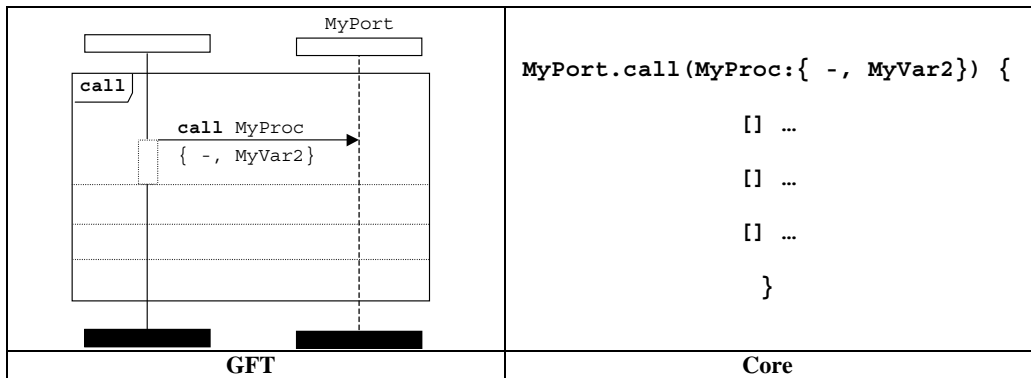


Figure 63/Z.142 – Opération d'appel bloquant avec modèle en ligne

L'expression en ligne d'appel est introduite pour faciliter la spécification des alternatives de réponses possibles à l'opération d'appel bloquant. L'opération d'appel peut être suivie par des alternatives de getreply, catch et timeout. Les réponses à un appel sont spécifiées dans l'expression en ligne d'appel qui suit l'opération d'appel et sont séparées par des lignes pointillées (voir la Figure 64).

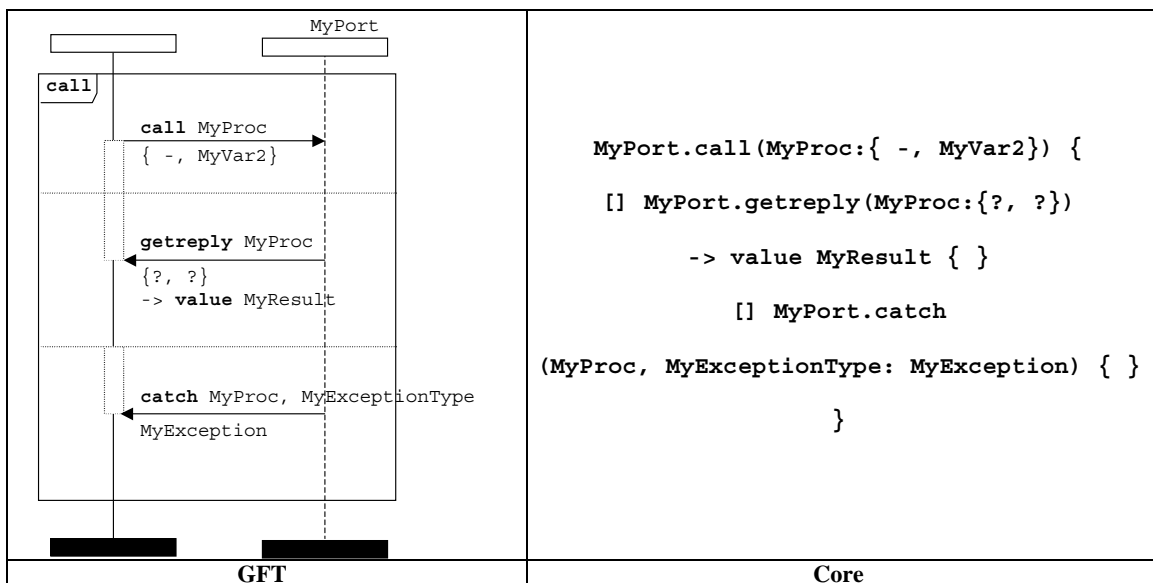


Figure 64/Z.142 – Opération d'appel bloquant suivie des alternatives getreply et catch

L'opération d'appel peut éventuellement comprendre une expiration de temporisateur. Le symbole de déclenchement de temporisateur implicite est utilisé à cet effet pour lancer la période de temporisation. Il est utilisé pour représenter l'exception d'expiration de temporisateur (voir la Figure 65).

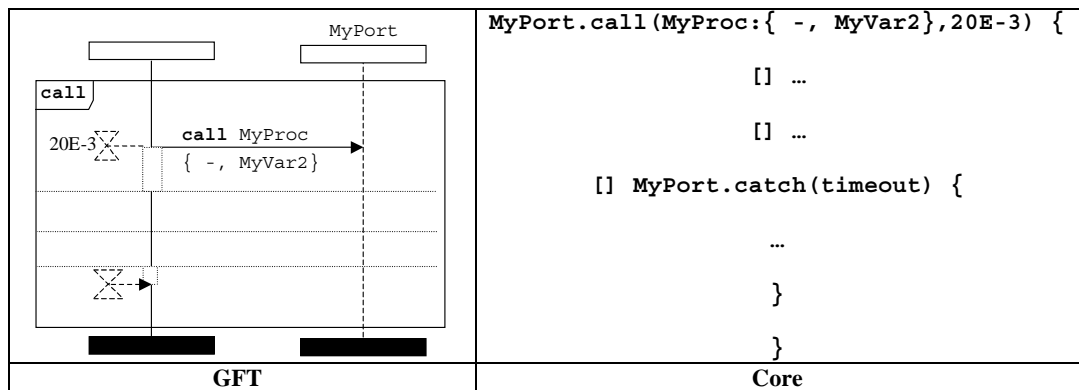


Figure 65/Z.142 – Opération d'appel bloquant suivie d'une exception d'expiration de temporisateur

11.8.4.1.2 Procédures d'appel non bloquant

L'opération d'appel non bloquant doit être représentée par un symbole de message sortant allant de la composante de test au port, le mot clé **call** étant placé au-dessus de la flèche de message et avant la signature. Il ne doit pas y avoir de symbole de région de suspension associé au symbole de message. La signature facultative figure au-dessus de la flèche de message. Le modèle (en ligne) est placé sous la flèche (voir les Figures 66 et 67).

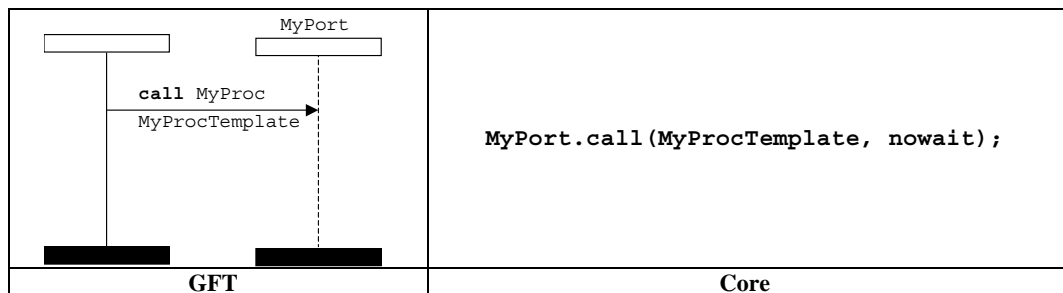


Figure 66/Z.142 – Opération d'appel non bloquant avec modèle de référence

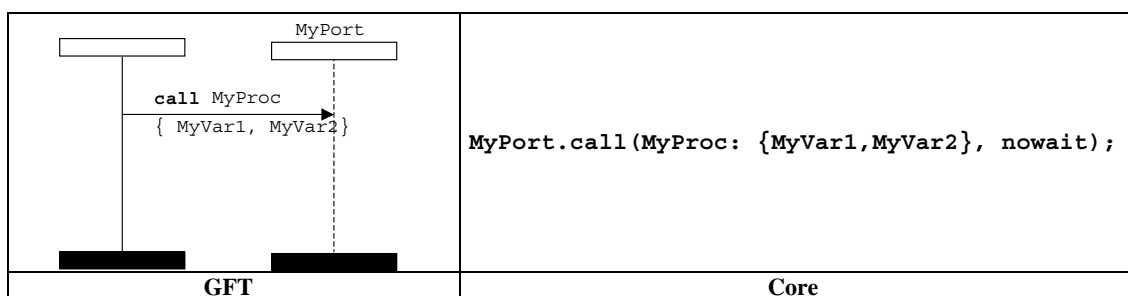


Figure 67/Z.142 – Opération d'appel non bloquant avec modèle en ligne

11.8.4.2 Opération d'obtention d'appel

L'opération d'obtention d'appel doit être représentée par une flèche de message entrant allant de l'instance de port à la composante de test, le mot clé **getcall** étant placé au-dessus de la flèche avant la signature (la signature est donc indiquée au-dessus de la flèche après le mot clé **getcall**). Le modèle (en ligne) est placé sous la flèche (voir les Figures 68 et 69).

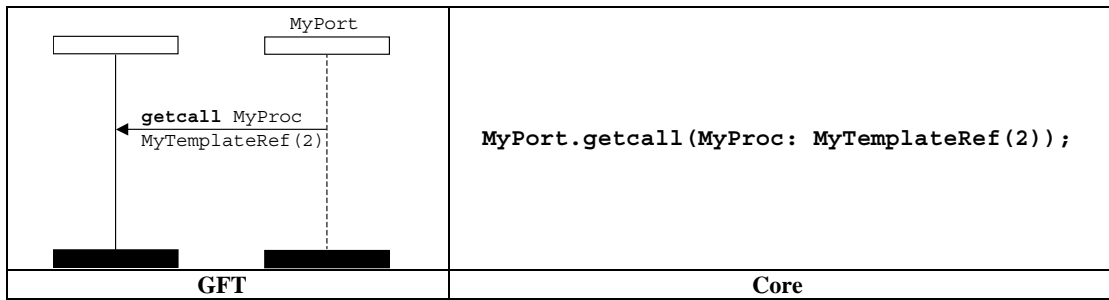


Figure 68/Z.142 – Opération d'obtention d'appel avec modèle de référence

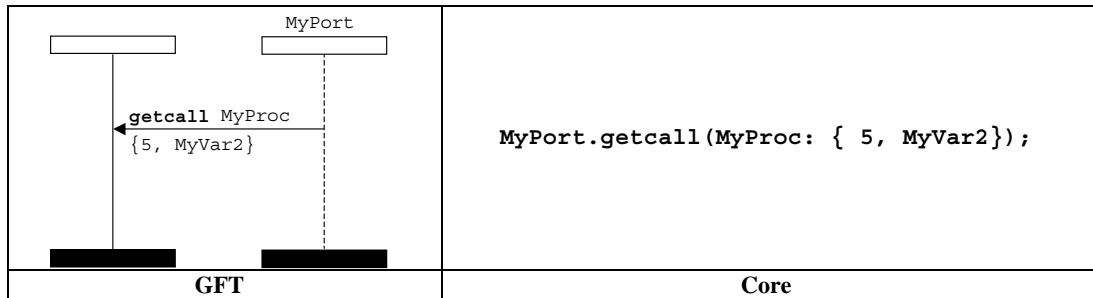


Figure 69/Z.142 – Opération d'obtention d'appel avec modèle en ligne

11.8.4.2.1 Acceptation de tout appel

L'opération acceptation de tout appel doit être représentée par une flèche de message entrant allant de l'instance de port à la composante de test, le mot clé `getcall` étant placé au-dessus de la flèche. Aucune autre information ne doit être associée au symbole de message (voir la Figure 70).

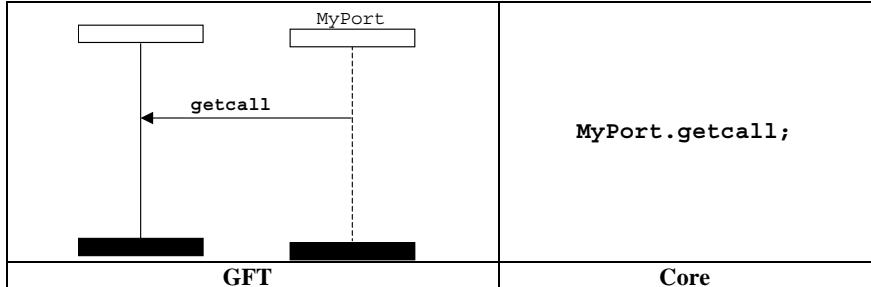


Figure 70/Z.142 – Opération acceptation de tout appel

11.8.4.2.2 Obtention d'appel pour tout port

L'opération obtention d'appel pour tout port est représentée par un symbole d'obtention représentatif d'un port quelconque et dirigé vers la composante de test, le mot clé `getcall` étant placé au-dessus de la flèche de message avant la signature si celle-ci est présente. Le modèle (en ligne), s'il existe, doit être placé sous la flèche (voir la Figure 71).

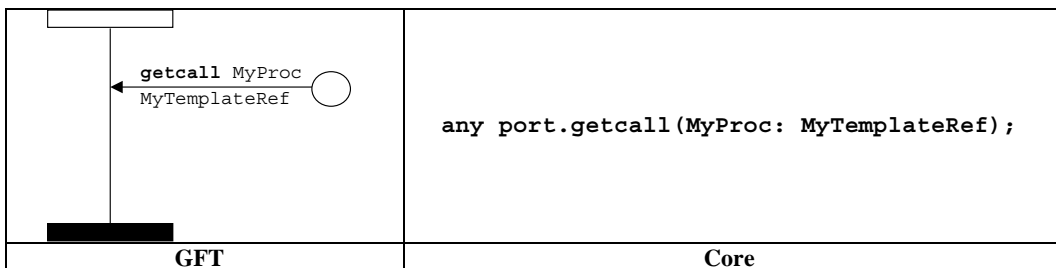


Figure 71/Z.142 – Opération obtention d'appel pour tout port avec modèle de référence

11.8.4.3 Opération de réponse

L'opération de réponse doit être représentée par un symbole de message sortant allant de la composante de test à l'instance de port, le mot clé **reply** étant placé au-dessus de la flèche de message avant la signature. La signature doit donc figurer au-dessus de la flèche après le mot clé **reply**. Le modèle (en ligne) doit être placé sous la flèche (voir les Figures 72 et 73).

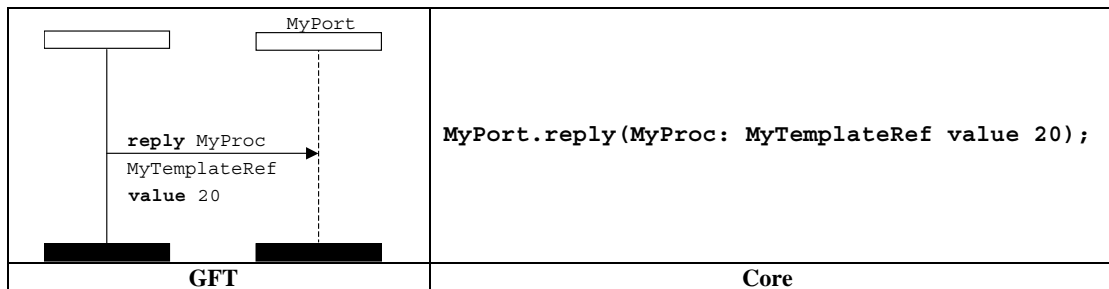


Figure 72/Z.142 – Opération de réponse avec modèle de référence

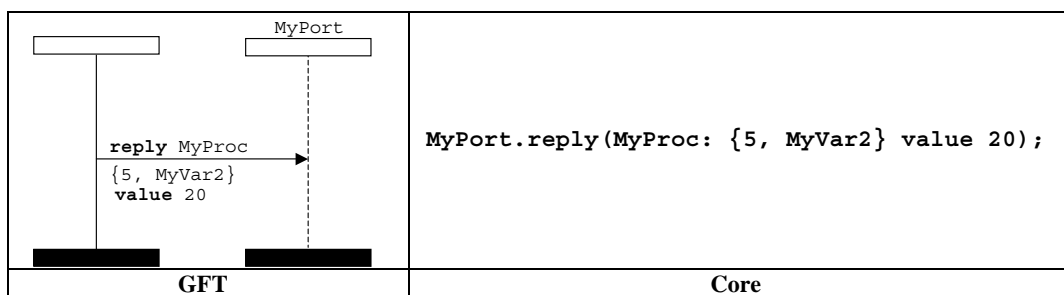


Figure 73/Z.142 – Opération de réponse avec modèle en ligne

11.8.4.4 Opération d'obtention de réponse

L'opération d'obtention de réponse doit être représentée par une flèche de message entrant allant de l'instance de port à la composante de test, le mot clé **getreply** étant placé au-dessus de la flèche avant la signature. Dans un symbole d'appel, l'extrémité de la flèche de message doit être rattachée à une région de suspension qui la précède sur la composante de test (voir les Figures 74 et 75). Hors d'un symbole d'appel, l'extrémité de la flèche de message ne doit pas être rattachée à une région de suspension qui la précède sur la composante de test (voir les Figures 76 et 77).

La signature doit être placée au-dessus de la flèche après le mot clé **getreply**. Le modèle (en ligne) doit être placé sous la flèche.

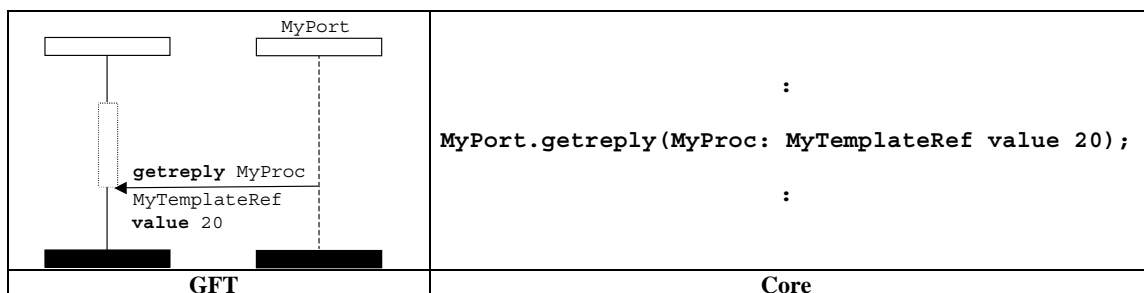


Figure 74/Z.142 – Opération d'obtention de réponse avec référence de modèle (dans un symbole d'appel)

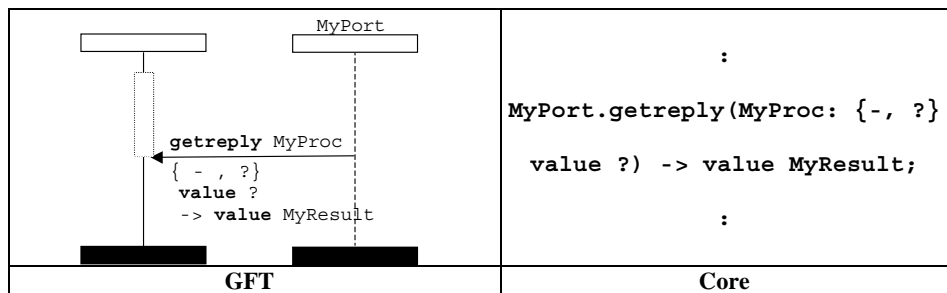


Figure 75/Z.142 – Opération d'obtention de réponse avec modèle en ligne (dans un symbole d'appel)

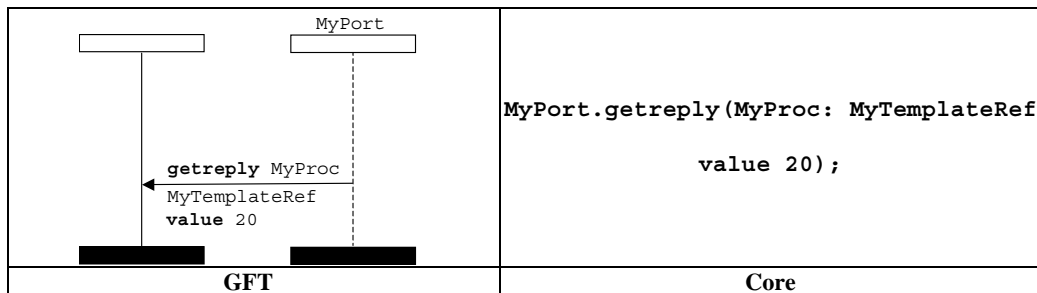


Figure 76/Z.142 – Opération d'obtention de réponse avec référence de modèle (hors d'un symbole d'appel)

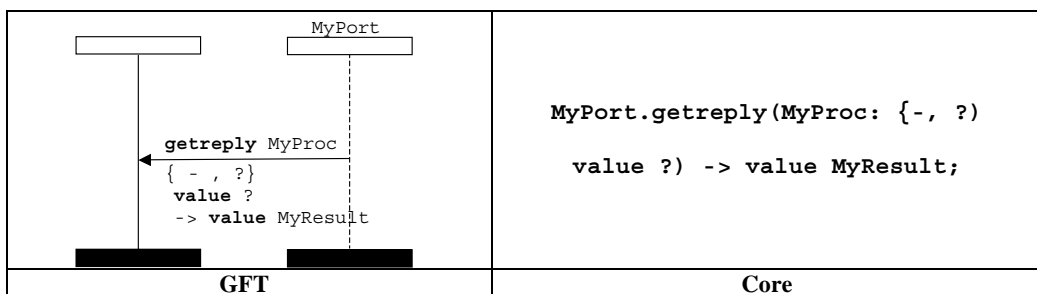


Figure 77/Z.142 – Opération d'obtention de réponse avec modèle en ligne (hors d'un symbole d'appel)

11.8.4.4.1 Acceptation de toute réponse à tout appel

L'opération d'acceptation de toute réponse à tout appel doit être représentée par une flèche de message entrant allant de l'instance de port à la composante de test, le mot clé **getreply** étant placé au-dessus du message. Ce mot clé **getreply** ne doit pas être suivi d'une signature. Dans un symbole d'appel, l'extrémité de la flèche de message doit être rattachée à une région de suspension qui la précède sur la composante de test (voir la Figure 78). Hors d'un symbole d'appel, l'extrémité de la flèche de message ne doit pas être rattachée à une région de suspension qui la précède sur la composante de test (voir la Figure 79).

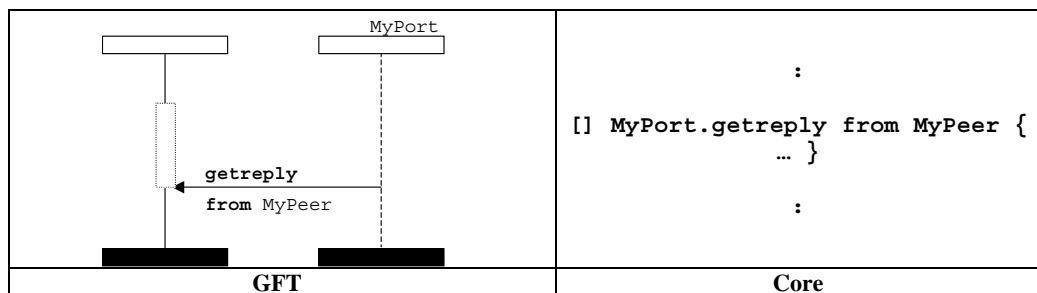


Figure 78/Z.142 – Opération d'obtention de réponse à tout appel (dans un symbole d'appel)

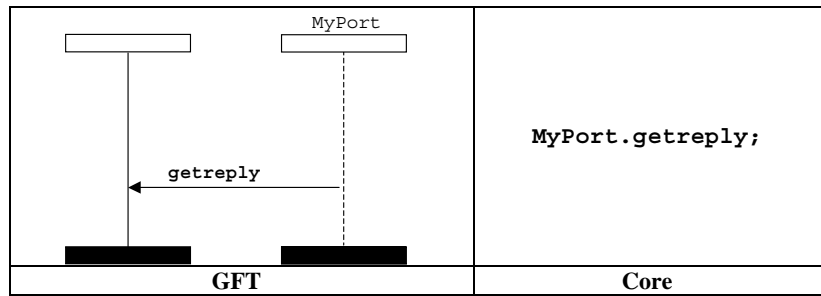


Figure 79/Z.142 – Opération d'obtention de réponse à tout appel (hors d'un symbole d'appel)

11.8.4.4.2 Obtention de réponse pour tout port

L'opération d'obtention de réponse pour tout port est représentée par un symbole d'obtention représentatif d'un port quelconque et dirigé vers la composante de test. Le mot clé **getreply** doit être placé au dessus de la flèche de message avant la signature si celle-ci est présente. Dans un symbole d'appel, l'extrémité de la flèche de message doit être rattachée à une région de suspension qui la précède sur la composante de test (voir la Figure 80). Hors d'un symbole d'appel, l'extrémité de la flèche de message ne doit pas être rattachée à une région de suspension qui la précède sur la composante de test (voir la Figure 81).

La signature, si elle est présente, doit être placée au-dessus de la flèche après le mot clé **getreply**. Le modèle (en ligne) est placé sous la flèche.

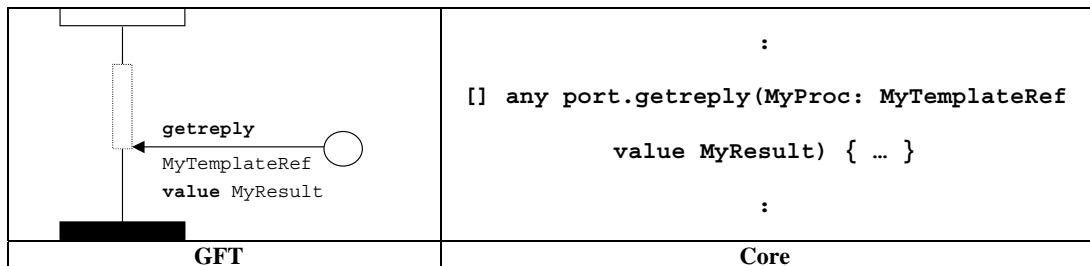


Figure 80/Z.142 – Obtention de réponse pour tout port (dans un symbole d'appel)

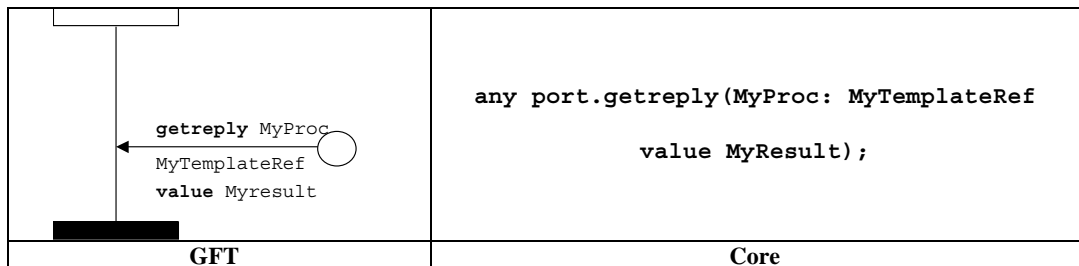


Figure 81/Z.142 – Obtention de réponse pour tout port (hors d'un symbole d'appel)

11.8.4.5 Opération de déclenchement d'une exception

L'opération de déclenchement d'une exception doit être représentée par un symbole de message sortant allant de la composante de test à l'instance de port. Le mot clé **raise** doit être placé au-dessus de la flèche de message avant la signature et le type d'exception, que sépare une virgule. Le modèle (en ligne) doit être placé sous la flèche (voir les Figures 82 et 83).

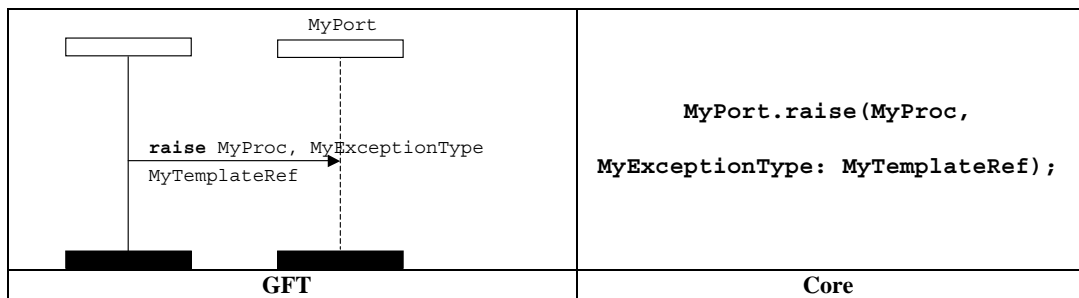


Figure 82/Z.142 – Opération de déclenchement d'une exception avec référence de modèle

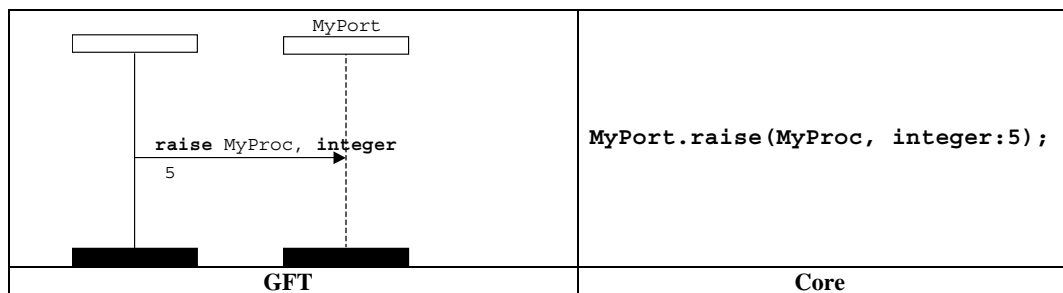


Figure 83/Z.142 – Opération de déclenchement d'une exception avec modèle en ligne

11.8.4.6 Opération d'acquisition

L'opération d'acquisition doit être représentée par une flèche de message entrant allant de l'instance de port à la composante de test, le mot clé **catch** étant placé au-dessus de la flèche avant la signature et le type d'exception (s'il est présent). Dans un symbole d'appel, l'extrémité de la flèche de message doit être rattachée à une région de suspension qui la précède sur la composante de test (voir les Figures 84 et 85). Hors d'un symbole d'appel, l'extrémité de la flèche de message ne doit pas être rattachée à une région de suspension qui la précède sur la composante de test (voir les Figures 86 et 87).

La signature et l'information de type d'exception facultative figurent au-dessus de la flèche après le mot clé **catch** et sont séparées par une virgule si le type d'exception est indiqué. Le modèle (en ligne) est placé sous la flèche.

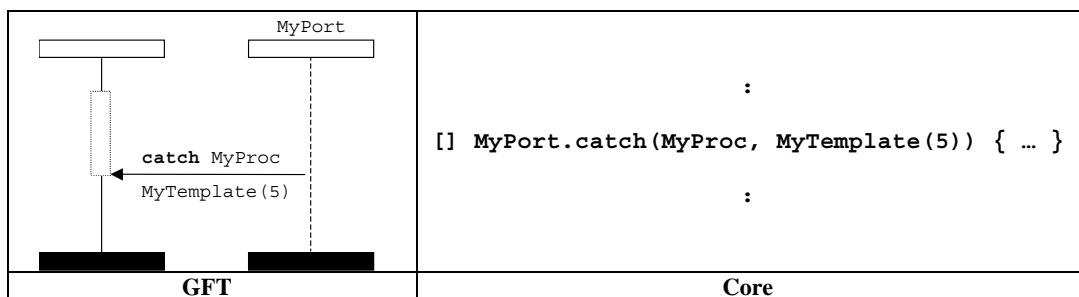


Figure 84/Z.142 – Opération d'acquisition avec référence de modèle (dans un symbole d'appel)

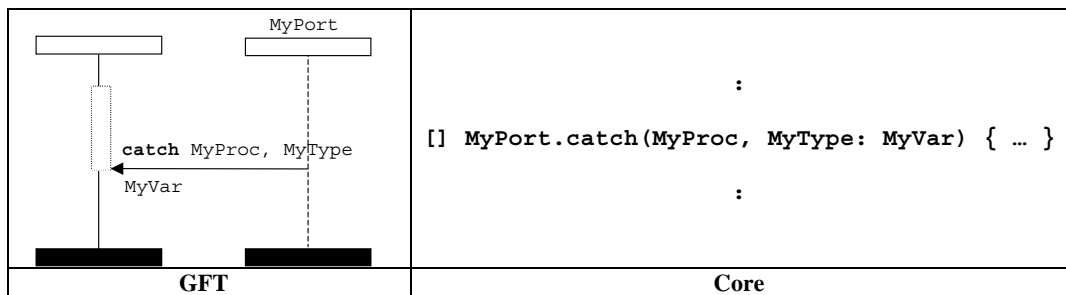


Figure 85/Z.142 – Opération d'acquisition avec modèle en ligne (dans un symbole d'appel)

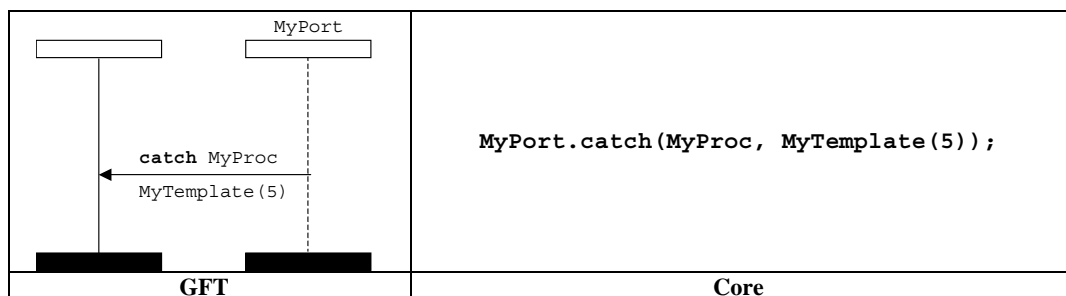


Figure 86/Z.142 – Opération d'acquisition avec référence de modèle (hors d'un symbole d'appel)

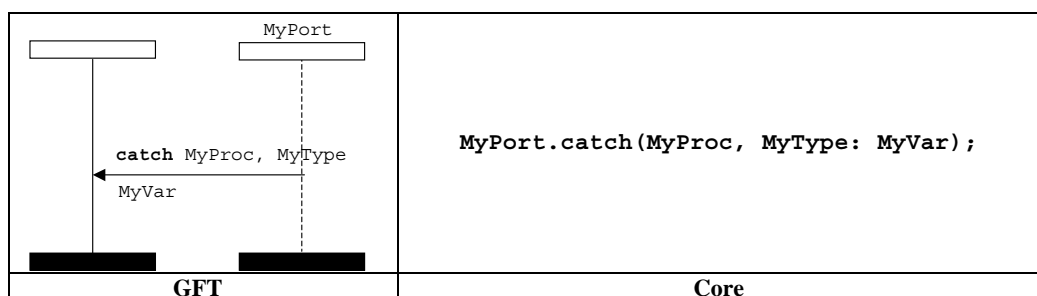


Figure 87/Z.142 – Opération d'acquisition avec modèle en ligne (hors d'un symbole d'appel)

11.8.4.6.1 Exception d'expiration de temporisateur

L'opération d'exception d'expiration de temporisateur doit être représentée par un symbole d'expiration de temporisateur dont la flèche est rattachée à la composante de test (voir la Figure 88). Aucune autre information ne doit être associée à ce symbole. Cette opération ne doit être utilisée que dans un symbole d'appel. L'extrémité de la flèche de message doit être rattachée à une région de suspension qui la précède sur la composante de test.

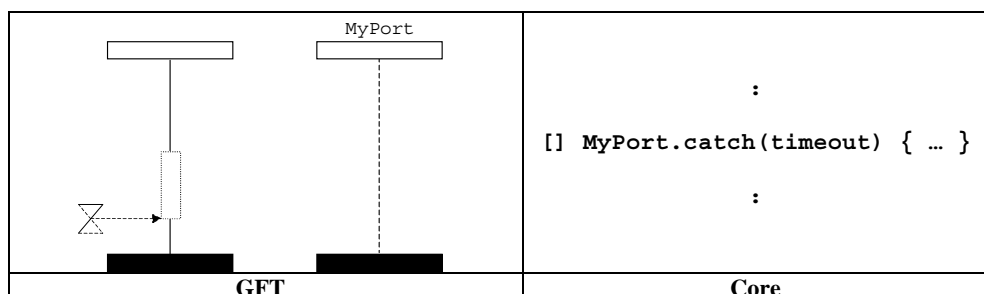


Figure 88/Z.142 – Exception d'expiration de temporisateur (dans un symbole d'appel)

11.8.4.6.2 Acquisition de toute exception

L'opération d'acquisition de toute exception doit être représentée par une flèche de message entrant allant de l'instance de port à la composante de test, le mot clé `catch` étant placé au-dessus de la flèche. Dans un symbole d'appel,

l'extrémité de la flèche de message doit être rattachée à une région de suspension qui la précède sur la composante de test (voir la Figure 89). Hors d'un symbole d'appel, l'extrémité de la flèche de message ne doit pas être rattachée à une région de suspension qui la précède sur la composante de test (voir la Figure 90). L'opération d'acquisition de toute exception n'est pas associée à un modèle ou à un type d'exception.

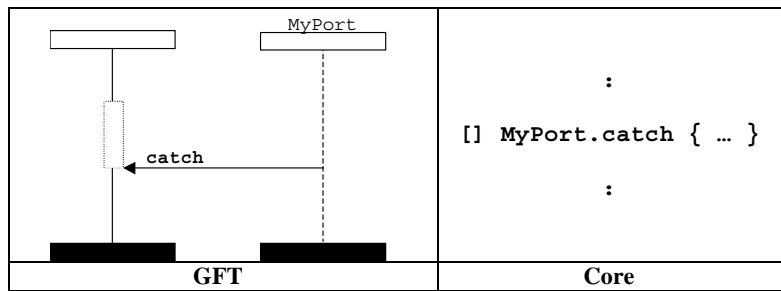


Figure 89/Z.142 – Acquisition de toute exception (dans un symbole d'appel)

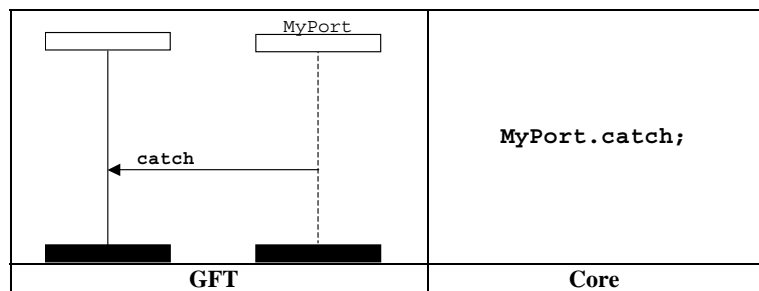


Figure 90/Z.142 – Acquisition de toute exception (hors d'un symbole d'appel)

11.8.4.6.3 Acquisition pour tout port

L'opération acquisition pour tout port est représentée par un symbole d'obtention représentatif d'un port quelconque et dirigée vers la composante de test, le mot clé `catch` étant placé au-dessus de la flèche de message. Dans un symbole d'appel, l'extrémité de la flèche de message doit être rattachée à une région de suspension qui la précède sur la composante de test (voir la Figure 91). Hors d'un symbole d'appel, l'extrémité de la flèche de message ne doit pas être rattachée à une région de suspension qui la précède sur la composante de test (voir la Figure 92). Le modèle, s'il est présent, est placé sous la flèche.

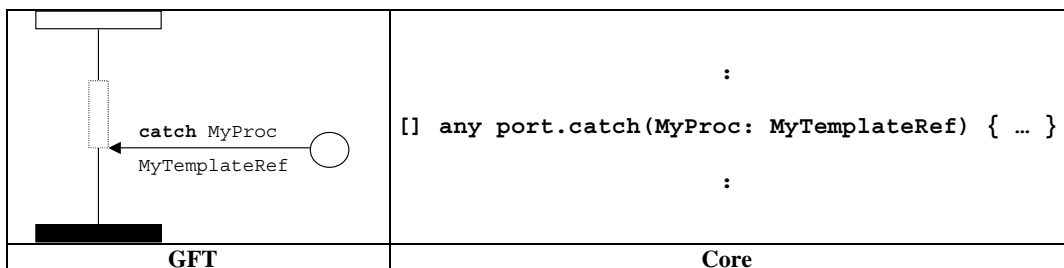


Figure 91/Z.142 – Acquisition pour tout port (dans un symbole d'appel)

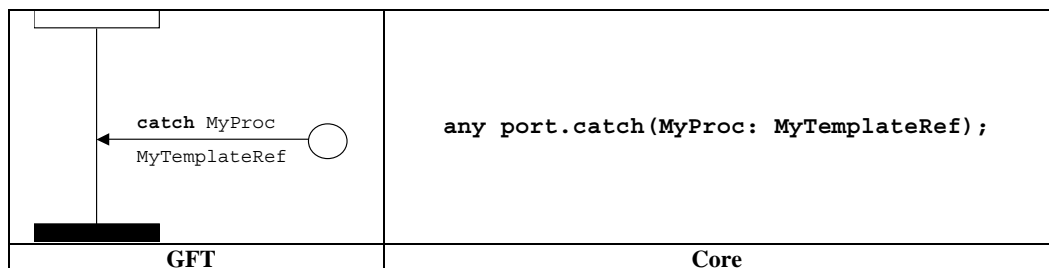


Figure 92/Z.142 – Acquisition pour tout port (hors d'un symbole d'appel)

11.8.5 Opération de vérification

L'opération de vérification doit être représentée par une flèche de message entrant allant de l'instance de port à la composante de test. Le mot clé **check** doit être placé au-dessus de la flèche. Les informations relatives aux opérations **receive** (voir la Figure 93), **getcall**, **getreply** (voir les Figures 94 et 95) et **catch** suivent le mot clé **check** et sont conformes aux règles applicables à la représentation de ces opérations.

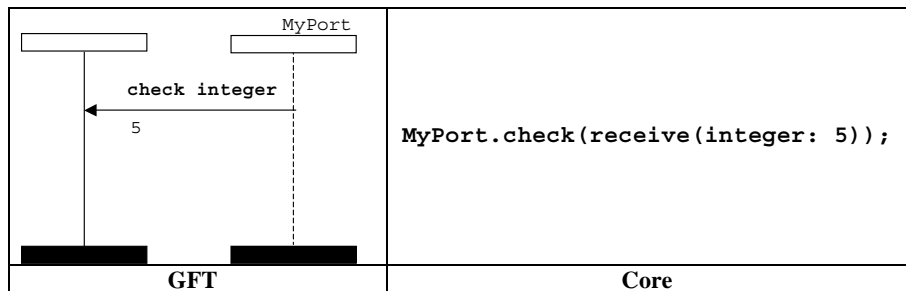


Figure 93/Z.142 – Vérification d'une opération de réception avec modèle en ligne

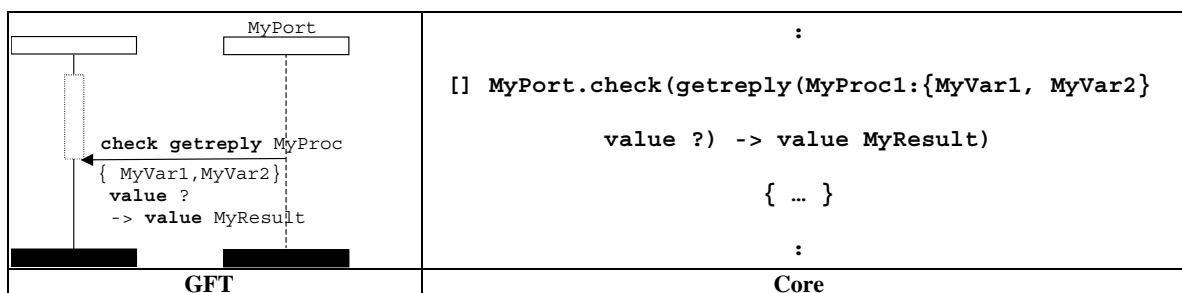


Figure 94/Z.142 – Vérification d'une opération d'obtention de réponse (dans un symbole d'appel)



Figure 95/Z.142 – Vérification d'une opération d'obtention de réponse (hors d'un symbole d'appel)

11.8.5.1 Opération de vérification totale

L'opération de vérification totale doit être représentée par une flèche de message entrant allant de l'instance de port à la composante de test, le mot clé **check** étant placé au-dessus de la flèche (voir la Figure 96). Elle ne doit être associée à aucun mot clé d'opération de réception, aucun type ou aucun modèle. Une information d'adresse et l'indication mémorisée de l'expéditeur peuvent le cas échéant être mentionnées.

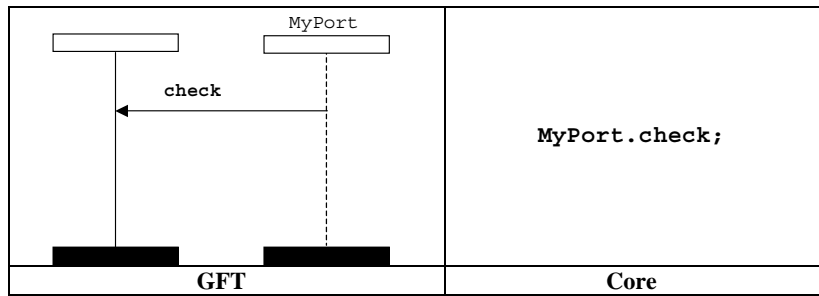


Figure 96/Z.142 – Opération de vérification totale

11.8.5.2 Vérification sur tout port

L'opération de vérification sur tout port est représentée par un symbole d'obtention représentatif d'un port quelconque et dirigé vers la composante de test, le mot clé **check** étant placé au-dessus de la flèche (voir la Figure 97). Les informations relatives aux opérations **receive**, **getcall**, **getreply** et **catch** suivent le mot clé check et sont conformes aux règles applicables à la représentation de ces opérations.

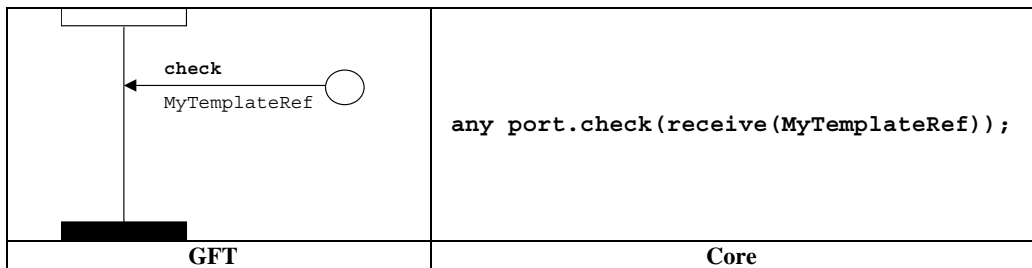


Figure 97/Z.142 – Vérification d'une opération de réception sur tout port

11.8.6 Opérations de commande des ports de communication

11.8.6.1 Opération de libération de port

L'opération de libération de port doit être représentée par un symbole de condition comprenant le mot clé **clear**. Ce symbole est rattaché à l'instance de composante de test, qui réalise l'opération, et au port qui est libéré (voir la Figure 98).

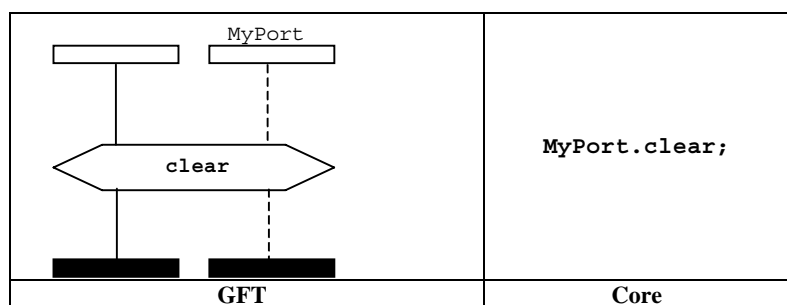


Figure 98/Z.142 – Opération de libération de port

11.8.6.2 Opération d'ouverture de port

L'opération d'ouverture de port doit être représentée par un symbole de condition avec le mot clé **start**. Ce symbole est rattaché à l'instance de composante de test, qui réalise l'opération, et au port qui est ouvert (voir la Figure 99).

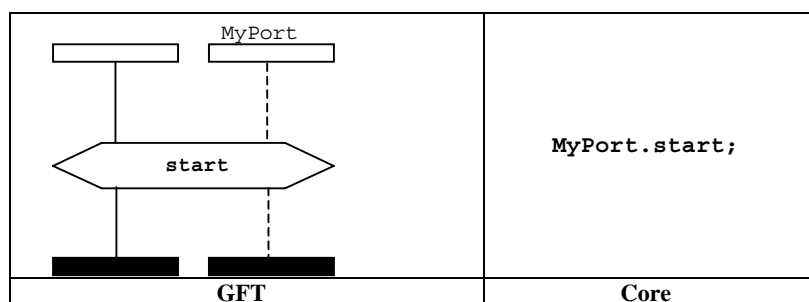


Figure 99/Z.142 – Opération d'ouverture de port

11.8.6.3 Opération de fermeture de port

L'opération de fermeture de port doit être représentée par un symbole de condition avec le mot clé **stop**. Elle est rattachée à l'instance de composante de test, qui réalise cette opération, et au port qui est fermé (voir la Figure 100).

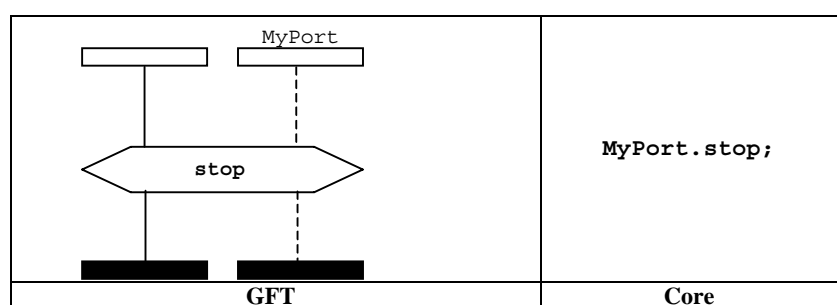


Figure 100/Z.142 – Opération de fermeture de port

11.8.6.4 Utilisation des mots clés *any* et *all* avec les ports

La représentation en format GFT du mot clé **any** pour des ports dans le cas d'une opération **receive**, **trigger**, **getcall**, **getreply**, **catch** ou **check** est expliquée dans les sous-sections pertinentes du § 11.8.

La représentation du mot clé **all** pour les ports dans le cas d'une opération **clear**, **start** ou **stop** se fait en rattachant le symbole de condition contenant l'opération **clear**, **start** ou **stop** à toutes les instances de port représentées dans le diagramme GFT pour un test élémentaire, une fonction ou une variante.

11.9 Opérations de temporisation

En format GFT, il existe deux symboles de temporisation différents, suivant qu'il s'agisse d'un temporisateur identifié ou d'un temporisateur d'appel (voir la Figure 101). Ils se distinguent par leur apparence, un temporisateur identifié étant dessiné en traits pleins et un temporisateur d'appel en traits pointillés. Un temporisateur identifié doit avoir son nom rattaché à son symbole, alors qu'un temporisateur d'appel n'a pas de nom. Les temporisateurs identifiés sont décrits dans le présent paragraphe. Les temporisateurs d'appel sont décrits dans le § 11.8.4.



Figure 101/Z.142 – Temporisateur identifié et temporisateur d'appel

Le format GFT ne fournit pas de représentation graphique pour l'opération de temporisation **running** (qui est une expression booléenne). Cette opération est indiquée par du texte lorsqu'elle est utilisée.

11.9.1 Opération de déclenchement de temporisateur

Pour l'opération de déclenchement de temporisateur, le symbole de déclenchement de temporisateur doit être rattaché à l'instance de composante. Un nom de temporisateur et une valeur de durée facultative (indiquée entre parenthèses) peuvent lui être associés (voir la Figure 102).

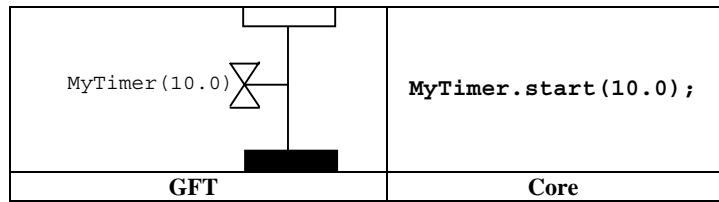


Figure 102/Z.142 – Opération de déclenchement de temporisateur

11.9.2 Opération d'arrêt de temporisateur

Pour l'opération d'arrêt de temporisateur, le symbole d'arrêt de temporisateur doit être rattaché à l'instance de composante. Un nom de temporisateur facultatif peut lui être associé (voir la Figure 103).

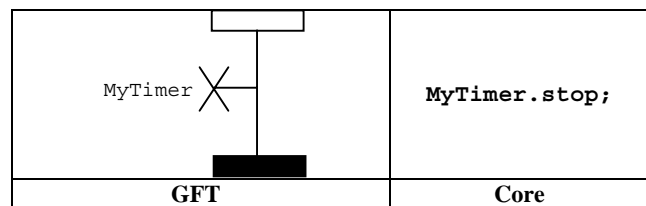


Figure 103/Z.142 – Opération d'arrêt de temporisateur

Les symboles de déclenchement et d'arrêt d'un temporisateur peuvent être reliés par une ligne verticale. Dans ce cas, il suffit de spécifier l'identificateur du temporisateur à côté du symbole de déclenchement de temporisateur (voir la Figure 104).

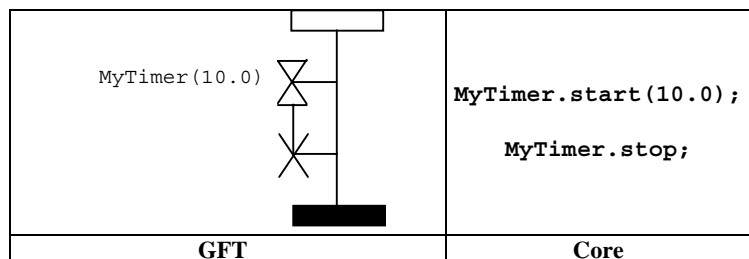


Figure 104/Z.142 – Liaison entre le symbole de déclenchement et le symbole d'arrêt de temporisateur

11.9.3 Opération d'expiration de temporisateur

Pour l'opération d'expiration de temporisateur, le symbole d'expiration de temporisateur doit être rattaché à l'instance de composante. Un nom de temporisateur facultatif peut lui être associé (voir la Figure 105).

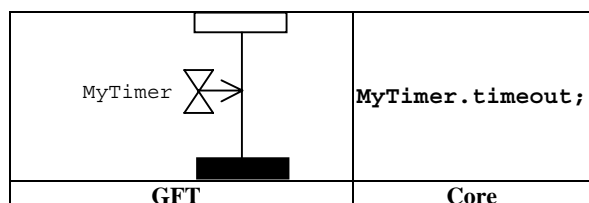


Figure 105/Z.142 – Opération d'expiration de temporisateur

Les symboles de déclenchement de temporisateur et d'expiration de temporisateur peuvent être reliés par une ligne verticale. Dans ce cas, il suffit de spécifier l'identificateur du temporisateur à côté du symbole de déclenchement de temporisateur (voir la Figure 106).

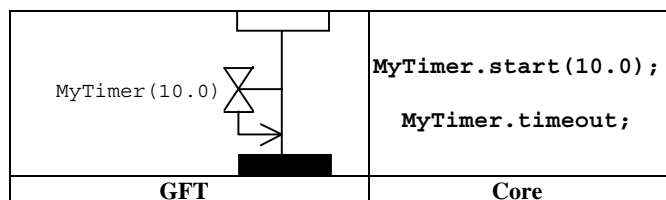


Figure 106/Z.142 – Liaison entre les symboles de déclenchement de temporisateur et d'expiration de temporisateur

11.9.4 Opération de lecture de temporisateur

L'opération de lecture de temporisateur doit être indiquée dans un cadre d'action (voir la Figure 107).

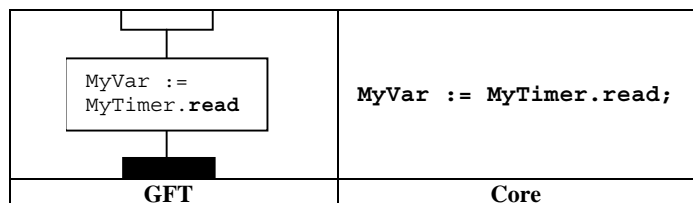


Figure 107/Z.142 – Opération de lecture de temporisateur

11.9.5 Utilisation des mots clés *any* et *all* avec les temporisateurs

L'opération d'arrêt de temporisateur peut être appliquée à tous (**all**) les temporisateurs (voir la Figure 108).

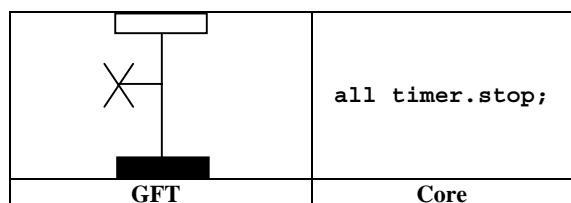


Figure 108/Z.142 – Arrêt de tous les temporisateurs

L'opération d'expiration de temporisateur peut être appliquée à un temporisateur quelconque (**any**) (voir la Figure 109).

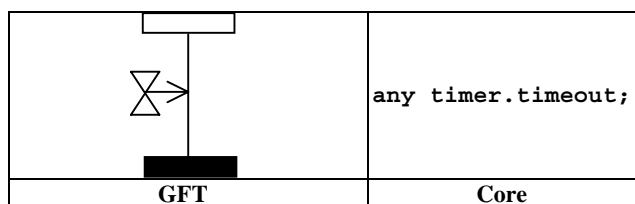


Figure 109/Z.142 – Opération d'expiration de temporisateur à partir d'un temporisateur quelconque

11.10 Opérations de verdict de test

L'opération de définition de verdict **setverdict** est représentée en format GFT par un symbole de condition dans lequel figure la valeur **pass**, **fail**, **inconc** ou **none** (voir la Figure 110).

NOTE – Les règles applicables à la définition d'un nouveau verdict suivent les règles TTCN-3 habituelles d'effacement par recouvrement applicables aux verdicts de test.

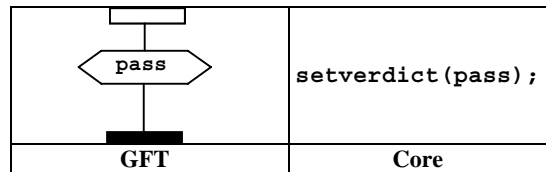


Figure 110/Z.142 – Définition du verdict local

Le format GFT ne fournit pas de représentation graphique pour l'opération `getverdict` (qui est une expression). Cette opération est indiquée par du texte là où elle est utilisée.

11.11 Actions externes

Les actions externes sont représentées dans des symboles de cadre d'action (voir la Figure 111). La syntaxe de l'action externe est indiquée dans ce symbole.

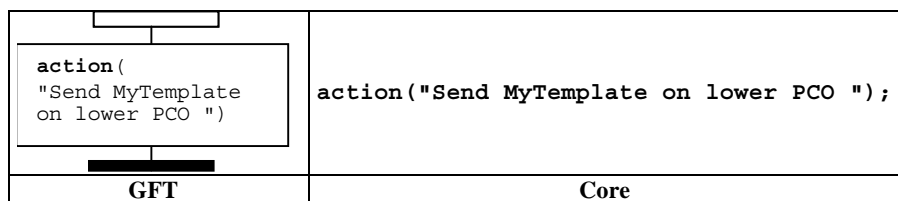


Figure 111/Z.142 – Actions externes

11.12 Spécification des attributs

Les attributs définis dans la partie commande d'un module, les tests élémentaires, les fonctions et les variantes sont représentés dans le symbole de texte. La syntaxe de l'instruction `with` est indiquée dans ce symbole. La Figure 112 donne un exemple de spécification d'attributs.

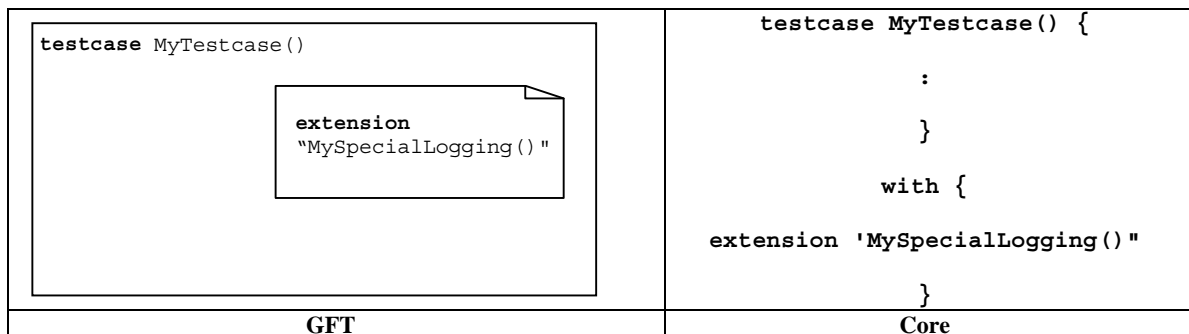


Figure 112/Z.142 – Spécification d'attributs

Annex A

GFT BNF

A.1 Meta-language for GFT

The graphical syntax for GFT is defined on the basis of the graphical syntax of MSC (ITU-T Rec. Z.120 [3]). The graphical syntax definition uses a meta-language, which is explained in 1.4/Z.120 [3]:

"The graphical syntax is not precise enough to describe the graphics such that there are no graphical variations. Small variations on the actual shapes of the graphical terminal symbols are allowed. These include, for instance, shading of the filled symbols, the shape of an arrow head and the relative size of graphical elements. Whenever necessary the graphical syntax will be supplemented with informal explanation of the appearance of the constructions. The meta-language consists of a BNF-like notation with the special meta-constructions: *contains*, *is followed by*, *is associated with*, *is attached to*, *above* and *set*. These constructs behave like normal BNF production rules, but additionally they imply some logical or geometrical relation between the arguments. The *is attached to* construct behaves somewhat differently as explained below. The left-hand side of all constructs except *above* must be a symbol. A symbol is a non-terminal that produces in every production sequence exactly one graphical terminal. We will consider a symbol that *is attached to* other areas or that *is associated with* a text string as a symbol too. The explanation is informal and the meta-language does not precisely describe the geometrical dependencies."

See ITU-T Rec. Z.120 [3] for more details.

A.2 Conventions for the syntax description

Table A.1 defines the meta-notation used to specify the grammar for GFT. It is identical to the meta-notation used by TTCN-3, but different from the meta-notation used by MSC. In order to ease the readability, the correspondence to the MSC meta-notation is given in addition and differences are indicated.

Table A.1/Z.142 – The syntactic meta-notation

Meaning	TTCN-3	GFT	MSC	Differences
is defined to be	::=	::=	::=	
abc followed by xyz	abc xyz	abc xyz	abc xyz	
Alternative				
0 or 1 instances of abc	[abc]	[abc]	[abc]	
0 or more instances of abc	{abc}	{abc}	{abc}*	X
1 or more instances of abc	{abc} +	{abc} +	{abc} +	
Textual grouping	(...)	(...)	{...}	X
the non-terminal symbol abc	abc	abc (for a GFT non-terminal) or <u>abc</u> (for a TTCN non-terminal)	<abc>	X
a terminal symbol abc	abc	abc	abc or <name> or <character string>	X

A.3 The GFT grammar

A.3.1 Diagrams

A.3.1.1 Control diagram

```
ControlDiagram ::=
    Frame contains ( ControlHeading ControlBodyArea )

ControlHeading ::=
    TTCN3ModuleKeyword TTCN3ModuleId
    { LocalDefinition [ SemiColon ] }

ControlBodyArea ::=
    { ControlInstanceArea TextLayer ControlEventLayer } set
```



```

TextLayer ::=
  { TextArea } set

ControlEventLayer ::=
  ControlEventArea | ControlEventArea above ControlEventLayer

ControlEventArea ::=
  (
    InstanceTimerEventArea
  | ControlActionArea
  | InstanceInvocationArea
  | ExecuteTestcaseArea
  | ControlInlineExpressionArea )
  [ is associated with { CommentArea } set ]

```

A.3.1.2 Testcase diagram

```

TestcaseDiagram ::=
  Frame contains ( TestcaseHeading TestcaseBodyArea )

TestcaseHeading ::=
  TestcaseKeyword TestcaseIdentifier
  '(' [ TestcaseFormalParList ] ')'
  ConfigSpec
  { LocalDefinition [ SemiColon ] }

TestcaseBodyArea ::=
  { InstanceLayer TextLayer InstanceEventLayer PortEventLayer ConnectorLayer } set

InstanceLayer ::=
  { InstanceArea } set

InstanceEventLayer ::=
  InstanceEventArea | InstanceEventArea above InstanceEventLayer

InstanceEventArea ::=
  (
    InstanceSendEventArea
  | InstanceReceiveEventArea
  | InstanceCallEventArea
  | InstanceGetcallEventArea
  | InstanceReplyEventArea
  | InstanceGetreplyWithinCallEventArea
  | InstanceGetreplyOutsideCallEventArea
  | InstanceRaiseEventArea
  | InstanceCatchWithinCallEventArea
  | InstanceCatchTimeoutWithinCallEventArea
  | InstanceCatchOutsideCallEventArea
  | InstanceTriggerEventArea
  | InstanceCheckEventArea
  | InstanceFoundEventArea
  | InstanceTimerEventArea
  | InstanceActionArea
  | InstanceLabellingArea
  | InstanceConditionArea
  | InstanceInvocationArea
  | InstanceDefaultHandlingArea
  | InstanceComponentCreateArea
  | InstanceComponentStartArea
  | InstanceComponentStopArea
  | InstanceInlineExpressionArea )
  [ is associated with { CommentArea } set ]

/* STATIC SEMANTICS – A condition area containing a boolean expression shall be used within alt inline expression, i.e. AltArea, and
call inline expression, i.e. CallArea, only */

InstanceCallEventArea ::=
  InstanceBlockingCallEventArea
  | InstanceNonBlockingCallEventArea

PortEventLayer ::=
  PortEventArea | PortEventArea above PortEventLayer

PortEventArea ::=
  PortOutEventArea
  | PortOtherEventArea

```

```

PortOutEventArea ::=
    PortOutMsgEventArea
    | PortGetcallOutEventArea
    | PortGetreplyOutEventArea
    | PortCatchOutEventArea
    | PortTriggerOutEventArea
    | PortCheckOutEventArea

PortOtherEventArea ::=
    PortInMsgEventArea
    | PortCallInEventArea
    | PortReplyInEventArea
    | PortRaiseInEventArea
    | PortConditionArea
    | PortInvocationArea
    | PortInlineExpressionArea

ConnectorLayer ::=
    {
        SendArea
        | ReceiveArea
        | NonBlockingCallArea
        | GetcallArea
        | ReplyArea
        | GetreplyWithinCallArea
        | GetreplyOutsideCallArea
        | RaiseArea
        | CatchWithinCallArea
        | CatchOutsideCallArea
        | TriggerArea
        | CheckArea
        | ConditionArea
        | InvocationArea
        | InlineExpressionArea
    } set

```

A.3.1.3 Function diagram

```

FunctionDiagram ::=
    Frame contains ( FunctionHeading FunctionBodyArea )

FunctionHeading ::=
    FunctionKeyword FunctionIdentifier
    '(' ( [ FunctionFormalParList ] ')' )
    [ RunsOnSpec ] [ ReturnType ]
    { LocalDefinition [ SemiColon ] }

FunctionBodyArea ::=
    TestcaseBodyArea

```

A.3.1.4 Altstep diagram

```

AltstepDiagram ::=
    Frame contains ( AltstepHeading AltstepBodyArea )

AltstepHeading ::=
    AltstepKeyword AltstepIdentifier
    '(' ( [ AltstepFormalParList ] ')' )
    [ RunsOnSpec ]
    { LocalDefinition [ SemiColon ] }

AltstepBodyArea ::=
    TestcaseBodyArea

```

/* STATIC SEMANTICS – A altstep body area shall contain a single alt inline expression only */

A.3.1.5 Comments

```

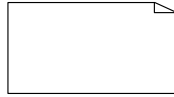
TextArea ::=
    TextSymbol
    contains ( { TTCN3Comments } [ MultiWithAttrib ] { TTCN3Comments } )

```

Note that there is no explicit rule for TTCN-3 comments, they are explained in ITU-T Rec. Z.140 [1], clause A.1.4.

/* STATIC SEMANTICS – Within a diagram there shall be at most one text symbol defining a with statement */

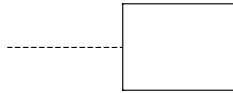
TextSymbol ::=



CommentArea ::=

EventCommentSymbol **contains** TTCN3Comments

EventCommentSymbol ::=



/* STATIC SEMANTICS – A comment symbol can be attached to any graphical symbol in GFT */

A.3.1.6 Diagram

Frame ::=



LocalDefinition ::=

ConstDef
| VarInstance
| TimerInstance

/* STATIC SEMANTICS – Declarations of constants and variables with create, activate, and execute statements as well as with functions that include communication functions must not be made textually within LocalDefinition, but must be made graphically within create, default, execute, and reference symbols, respectively */

A.3.2 Instances

A.3.2.1 Component instances

InstanceArea ::=

ComponentInstanceArea
| PortInstanceArea

ComponentInstanceArea ::=

ComponentHeadArea **is followed by** ComponentBodyArea

ComponentHeadArea ::=

(MTCOp | SelfOp)
is followed by (InstanceHeadSymbol [**contains** ComponentType])

InstanceHeadSymbol ::=



ComponentBodyArea ::=

InstanceAxisSymbol
is attached to { InstanceEventArea } **set**
is followed by ComponentEndArea

InstanceAxisSymbol ::=



ComponentEndArea ::=

InstanceEndSymbol
| StopArea
| ReturnArea

```
| RepeatSymbol
| GotoArea
```

```
/* STATIC SEMANTICS – The return symbol shall be used within function diagrams only */
/* STATIC SEMANTICS – The repeat symbol shall end the component instance of a altstep diagram only */
```

A.3.2.2 Port instances

```
PortInstanceArea ::=
    PortHeadArea is followed by PortBodyArea

PortHeadArea ::=
    Port
    is followed by ( InstanceHeadSymbol [ contains PortType ] )

PortBodyArea ::=
    PortAxisSymbol
    is attached to { PortEventArea } set
    is followed by InstanceEndSymbol

PortAxisSymbol ::=
```

!
!
!
!
!

A.3.2.3 Control instances

```
ControlInstanceArea ::=
    ControlInstanceHeadArea is followed by ControlInstanceBodyArea

ControlInstanceHeadArea ::=
    ControlKeyword
    is followed by InstanceHeadSymbol

ControlInstanceBodyArea ::=
    InstanceAxisSymbol
    is attached to { ControlEventArea } set
    is followed by ControlInstanceEndArea

ControlInstanceEndArea ::=
    InstanceEndSymbol
```

A.3.2.4 Instance end

```
InstanceEndSymbol ::=
```



```
StopArea ::=
    StopSymbol
    is associated with ( Expression )
```

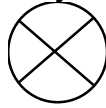
```
/* STATIC SEMANTICS – The expression shall refer to either the mtc or to self */
```

StopSymbol ::=

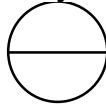


```
ReturnArea ::=
    ReturnSymbol
    [ is associated with Expression ]
```

ReturnSymbol ::=

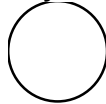


RepeatSymbol ::=



GotoArea ::=
GotoSymbol
contains LabelIdentifier

GotoSymbol ::=



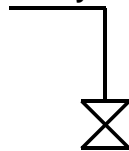
A.3.3 Timer

InstanceTimerEventArea ::=
InstanceTimerStartArea
| InstanceTimerStopArea
| InstanceTimeoutArea

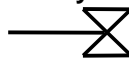
InstanceTimerStartArea ::=
TimerStartSymbol
is associated with (TimerRef ["(" TimerValue ")"])
is attached to InstanceAxisSymbol
[*is attached to* { TimerStopSymbol2 | TimeoutSymbol3 }]

TimerStartSymbol ::=
TimerStartSymbol1 | TimerStartSymbol2

TimerStartSymbol1 ::=



TimerStartSymbol2 ::=

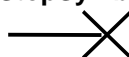


InstanceTimerStopArea ::=
TimerStopArea1 | TimerStopArea2

TimerStopArea1 ::=
TimerStopSymbol1
is associated with TimerRef
is attached to InstanceAxisSymbol

TimerStopArea2 ::=
TimerStopSymbol2
is attached to InstanceAxisSymbol
is attached to TimerStartSymbol

TimerStopSymbol1 ::=



TimerStopSymbol2 ::=



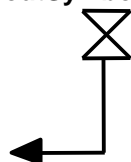
```
InstanceTimeoutArea ::=
  TimeoutArea1 | TimeoutArea2
```

```
TimeoutArea1 ::=
  TimeoutSymbol
  is associated with TimerRef
  is attached to InstanceAxisSymbol
```

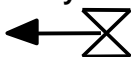
```
TimeoutArea2 ::=
  TimeoutSymbol3
  is attached to InstanceAxisSymbol
  is attached to TimerStartSymbol
```

```
TimeoutSymbol ::=
  TimeoutSymbol1 | TimeoutSymbol2
```

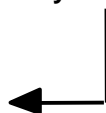
TimeoutSymbol1 ::=



TimeoutSymbol2 ::=



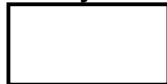
TimeoutSymbol3 ::=



A.3.4 Action

```
InstanceActionArea ::=
  ActionSymbol
  contains { ActionStatement [SemiColon] }+
  is attached to InstanceAxisSymbol
```

ActionSymbol ::=



```
ActionStatement ::=
  SUTStatements
  | ConnectStatement
  | MapStatement
  | DisconnectStatement
  | UnmapStatement
  | ConstDef
  | VarInstance
  | TimerInstance
  | Assignment
  | LogStatement
  | LoopConstruct
  | ConditionalConstruct
```

/ STATIC SEMANTICS – Declarations of constants and variables with create, activate, and execute statements as well as with function invocations of user-defined functions must not be made textually within an action box, but must be made graphically within create, default, execute, and reference symbols, respectively */*

/* STATIC SEMANTICS – Assignments with create, activate, and execute statements as well as with function invocations of user-defined functions must not be made textually within an action box, but must be made graphically within create, default, execute, and reference symbols, respectively */
 /* STATIC SEMANTICS – Only those loop and conditional constructs, which do not involve communication operations, i.e. those with 'data functions' only, may be contained in action boxes */

```
ControlActionArea ::=
  ActionSymbol
  is attached to InstanceAxisSymbol
  contains { ControlActionStatement [SemiColon] }+
```

```
ControlActionStatement ::=
  SUTStatements
  | ConstDef
  | VarInstance
  | TimerInstance
  | Assignment
  | LogStatement
```

/* STATIC SEMANTICS – Declarations of constants and variables with create, activate, and execute statements as well as with function invocations of user-defined functions must not be made textually within an action box, but must be made graphically within create, default, execute, and reference symbols, respectively */

/* STATIC SEMANTICS – Assignments with create, activate, and execute statements as well as with function invocations of user-defined functions must not be made textually within an action box, but must be made graphically within create, default, execute, and reference symbols, respectively */

A.3.5 Invocation

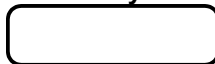
```
InvocationArea ::=
  ReferenceSymbol
  contains Invocation
  is attached to InstanceAxisSymbol
  [ is attached to { PortAxisSymbol } set ]
```

/* STATIC SEMANTICS – All port instances have to be covered by the reference symbol for an invoked function if it has a runs on specification, as well as for an invoked altstep */

/* STATIC SEMANTICS – Only those port instances, which are passed into a function via port parameters, have to be covered by the reference symbol for an invoked function without a runs on specification. Note that the reference symbol may be attached to port instances which are not passed as port parameters into the function. */

```
Invocation ::=
  FunctionInstance
  | AltstepInstance
  | ConstDef
  | VarInstance
  | Assignment
```

ReferenceSymbol ::=



A.3.5.1 Function and altstep invocation on component/Control instances

```
InstanceInvocationArea ::=
  InstanceInvocationBeginSymbol
  is followed by InstanceInvocationEndSymbol
  is attached to InstanceAxisSymbol
  is attached to InvocationArea
```

```
InstanceInvocationBeginSymbol ::=
  VoidSymbol
```

```
InstanceInvocationEndSymbol ::=
  VoidSymbol
```

A.3.5.2 Function and altstep invocation on ports

```
PortInvocationArea ::=
  PortInvocationBeginSymbol
  is followed by PortInvocationEndSymbol
  is attached to PortAxisSymbol
  is attached to InvocationArea
```

/* STATIC SEMANTICS – Only invocations with function instances and test step instances shall be attached to a port instance, in that case all port instances have to be covered by the reference symbol for an invoked function if it has a runs on specification, as well as for an invoked altstep */

```
PortInvocationBeginSymbol ::=
    VoidSymbol
```

```
PortInvocationEndSymbol ::=
    VoidSymbol
```

A.3.5.3 Testcase execution

```
ExecuteTestcaseArea ::=
    ExecuteSymbol
    contains TestCaseExecution
    is attached to InstanceAxisSymbol
```

```
TestCaseExecution ::=
    TestCaseInstance
    | ConstDef
    | VarInstance
    | Assignment
```

/* STATIC SEMANTICS – Declarations of constants and variables as well as assignments shall use as outermost right-hand expression an execute statement */

ExecuteSymbol ::=



A.3.6 Activation/Deactivation of defaults

```
InstanceDefaultHandlingArea ::=
    DefaultSymbol
    contains DefaultHandling
    is attached to InstanceAxisSymbol
```

```
DefaultHandling ::=
    ActivateOp
    | DeactivateStatement
    | ConstDef
    | VarInstance
    | Assignment
```

/* STATIC SEMANTICS – Declarations of constants and variables as well as assignments shall use as outermost right-hand expression an activate statement */

DefaultSymbol ::=



A.3.7 Test components

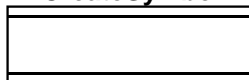
A.3.7.1 Creation of test components

```
InstanceComponentCreateArea ::=
    CreateSymbol
    contains Creation
    is attached to InstanceAxisSymbol
```

```
Creation ::=
    CreateOp
    | ConstDef
    | VarInstance
    | Assignment
```

/* STATIC SEMANTICS – Declarations of constants and variables as well as assignments shall use as outermost right-hand expression a create statement */

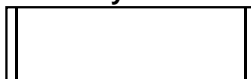
CreateSymbol ::=



A.3.7.2 Starting test components

```
InstanceComponentStartArea ::=
  StartSymbol
  contains StartTCStatement
  is attached to InstanceAxisSymbol
```

StartSymbol ::=



A.3.7.3 Stopping test components

```
InstanceComponentStopArea ::=
  StopSymbol
  is associated with ( Expression | AllKeyword )
  is attached to InstanceAxisSymbol
```

/* STATIC SEMANTICS – The expression shall refer to a component identifier */

/* STATIC SEMANTICS – The instance component stop area shall be used as last event of an operand in an inline expression symbol, if the component stops itself (e.g. self.stop) or stops the test execution (e.g. mtc.stop). */

A.3.8 Inline expressions

```
InlineExpressionArea ::=
  IfArea
  | ForArea
  | WhileArea
  | DoWhileArea
  | AltArea
  | InterleaveArea
  | CallArea
```

```
IfArea ::=
  IfInlineExpressionArea
  is attached to InstanceInlineExpressionBeginSymbol
  [ is attached to InstanceInlineExpressionSeparatorSymbol ]
  is attached to InstanceInlineExpressionEndSymbol
  [ is attached to { PortInlineExpressionBeginSymbol } set
  [ is attached to { PortInlineExpressionSeparatorSymbol } set ]
  is attached to { PortInlineExpressionEndSymbol } set ]
```

/* STATIC SEMANTICS – If a SeparatorSymbol is contained in the inline expression symbol, then InstanceInlineExpressionSeparatorSymbols on component and port instances are used to attach the SeparatorSymbol to the respective instances. */

```
InstanceInlineExpressionBeginSymbol ::=
  VoidSymbol
```

```
InstanceInlineExpressionSeparatorSymbol ::=
  VoidSymbol
```

```
InstanceInlineExpressionEndSymbol ::=
  VoidSymbol
```

```
VoidSymbol ::= .
```

```
IfInlineExpressionArea ::=
  InlineExpressionSymbol
  contains ( IfKeyword '(' BooleanExpression ')'
  is followed by OperandArea
  [ is followed by SeparatorSymbol
  is followed by OperandArea ] )
```

```
OperandArea ::=
  ConnectorLayer
```

/* STATIC SEMANTICS – The event layer within an operand area shall not have a condition with a boolean expression */

```

ForArea ::=
  ForInlineExpressionArea
  is attached to InstanceInlineExpressionBeginSymbol
  is attached to InstanceInlineExpressionEndSymbol
  [ is attached to { PortInlineExpressionBeginSymbol } set
    is attached to { PortInlineExpressionEndSymbol } set ]

ForInlineExpressionArea ::=
  InlineExpressionSymbol
  contains ( ForKeyword '(' Initial [SemiColon] Final [SemiColon] Step ')'
    is followed by OperandArea )

WhileArea ::=
  WhileInlineExpressionArea
  is attached to InstanceInlineExpressionBeginSymbol
  is attached to InstanceInlineExpressionEndSymbol
  [ is attached to { PortInlineExpressionBeginSymbol } set
    is attached to { PortInlineExpressionEndSymbol } set ]

WhileInlineExpressionArea ::=
  InlineExpressionSymbol
  contains ( WhileKeyword '(' BooleanExpression ')'
    is followed by OperandArea )

DoWhileArea ::=
  DoWhileInlineExpressionArea
  is attached to InstanceInlineExpressionBeginSymbol
  is attached to InstanceInlineExpressionEndSymbol
  [ is attached to { PortInlineExpressionBeginSymbol } set
    is attached to { PortInlineExpressionEndSymbol } set ]

DoWhileInlineExpressionArea ::=
  InlineExpressionSymbol
  contains ( DoKeyword WhileKeyword '(' BooleanExpression ')'
    is followed by OperandArea )

AltArea ::=
  AltInlineExpressionArea
  is attached to InstanceInlineExpressionBeginSymbol
  { is attached to InstanceInlineExpressionSeparatorSymbol }
  is attached to InstanceInlineExpressionEndSymbol
  [ is attached to { PortInlineExpressionBeginSymbol } set
    [ is attached to { PortInlineExpressionSeparatorSymbol } set ]
    is attached to { PortInlineExpressionEndSymbol } set ]

```

/* STATIC SEMANTICS – The number of InstanceInlineExpressionSeparatorSymbol per component and port instances has to adhere to the number of SeparatorSymbols contained within the inline expression symbol: the InstanceInlineExpressionSeparatorSymbol on component and port instances are used to attach the SeparatorSymbols to the respective instances. */

```

AltInlineExpressionArea ::=
  InlineExpressionSymbol
  contains ( AltKeyword
    is followed by GuardedOperandArea
    { is followed by SeparatorSymbol
      is followed by GuardedOperandArea }
    [ is followed by SeparatorSymbol
      is followed by ElseOperandArea ] )

```

```

GuardedOperandArea ::=
  GuardOpLayer is followed by
  ConnectorLayer

```

/* STATIC SEMANTICS – For the individual operands of an alt inline expression at first, either a InstanceTimeoutArea shall be given on the component instance, or a GuardOpLayer has to be given */

```

GuardOpLayer ::=
  DoneArea
  | ReceiveArea
  | TriggerArea
  | GetcallArea
  | CatchOutsideCallArea
  | CheckArea
  | GetreplyOutsideCallArea

```

```

ElseOperandArea ::=
  ElseConditionArea
  is followed by ConnectorLayer

```

```

InterleaveArea ::=
    InterleaveInlineExpressionArea
    is attached to InstanceInlineExpressionBeginSymbol
    { is attached to InstanceInlineExpressionSeparatorSymbol }
    is attached to InstanceInlineExpressionEndSymbol
    [ is attached to { PortInlineExpressionBeginSymbol } set
      [ is attached to { PortInlineExpressionSeparatorSymbol } set ]
      is attached to { PortInlineExpressionEndSymbol } set ]

/* STATIC SEMANTICS – The number of InstanceInlineExpressionSeparatorSymbol per component and port instances has to adhere
to the number of SeparatorSymbols contained within the inline expression symbol: the InstanceInlineExpressionSeparatorSymbol on
component and port instances are used to attach the SeparatorSymbols to the respective instances. */

InterleaveInlineExpressionArea ::=
    InlineExpressionSymbol
    contains ( InterleavedKeyword
               is followed by UnguardedOperandArea
               { is followed by SeparatorSymbol
                 is followed by UnguardedOperandArea } )

UnguardedOperandArea ::=
    UnguardedOpLayer is followed by
    ConnectorLayer

/* STATIC SEMANTICS – The connector layer within an interleave inline expression area may not contain loop statements, goto,
activate, deactivate, stop, return or calls to functions */

UnguardedOpLayer ::=
    ReceiveArea
    | TriggerArea
    | GetcallArea
    | CatchOutsideCallArea
    | CheckArea
    | GetreplyOutsideCallArea

CallArea ::=
    CallInlineExpressionArea
    is attached to InstanceInlineExpressionBeginSymbol
    { is attached to InstanceInlineExpressionSeparatorSymbol }
    is attached to InstanceInlineExpressionEndSymbol
    [ is attached to { PortInlineExpressionBeginSymbol } set
      [ is attached to { PortInlineExpressionSeparatorSymbol } set ]
      is attached to { PortInlineExpressionEndSymbol } set ]

/* STATIC SEMANTICS – The number of InstanceInlineExpressionSeparatorSymbol per component and port instances has to adhere
to the number of SeparatorSymbols contained within the inline expression symbol: the InstanceInlineExpressionSeparatorSymbol on
component and port instances are used to attach the SeparatorSymbols to the respective instances. */

CallInlineExpressionArea ::=
    InlineExpressionSymbol
    contains ( CallOpKeyword '(' TemplateInstance ')' [ ToClause ]
               is followed by InstanceCallEventArea
               { is followed by SeparatorSymbol
                 is followed by GuardedCallOperandArea } )

GuardedCallOperandArea ::=
    [ GuardedConditionLayer is followed by ]
    CallBodyOpsLayer
    is attached to SuspensionRegionSymbol
    is followed by ConnectorLayer

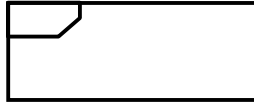
/* STATIC SEMANTICS – For the individual operands in the GuardedCallOperandArea of a call inline expression at first, either a
InstanceCatchTimeoutWithinCallEventArea shall be given on the component instance, or a CallBodyOpsLayer has to be given */

GuardedConditionLayer ::=
    BooleanExpressionConditionArea
    | DoneArea

CallBodyOpsLayer ::=
    GetreplyWithinCallArea
    | CatchWithinCallArea

```

InlineExpressionSymbol ::=



SeparatorSymbol ::=



A.3.8.1 Inline expressions on component instances

```
InstanceInlineExpressionArea ::=
  InstanceIfArea
  | InstanceForArea
  | InstanceWhileArea
  | InstanceDoWhileArea
  | InstanceAltArea
  | InstanceInterleaveArea
  | InstanceCallArea

InstanceIfArea ::=
  ( InstanceInlineExpressionBeginSymbol
    { is followed by InstanceEventArea }
    { is followed by InstanceInlineExpressionSeparatorSymbol
      { is followed by InstanceEventArea } } ]
  is followed by InstanceInlineExpressionEndSymbol )
is attached to InstanceAxisSymbol
is attached to IfInlineExpressionArea

InstanceForArea ::=
  ( InstanceInlineExpressionBeginSymbol
    { is followed by InstanceEventArea }
    is followed by InstanceInlineExpressionEndSymbol )
is attached to InstanceAxisSymbol
is attached to ForInlineExpressionArea

InstanceWhileArea ::=
  ( InstanceInlineExpressionBeginSymbol
    { is followed by InstanceEventArea }
    is followed by InstanceInlineExpressionEndSymbol )
is attached to InstanceAxisSymbol
is attached to WhileInlineExpressionArea

InstanceDoWhileArea ::=
  ( InstanceInlineExpressionBeginSymbol
    { is followed by InstanceEventArea }
    is followed by InstanceInlineExpressionEndSymbol )
is attached to InstanceAxisSymbol
is attached to DoWhileInlineExpressionArea

InstanceAltArea ::=
  ( InstanceInlineExpressionBeginSymbol
    [ is followed by InstanceBooleanExpressionConditionArea ]
    is followed by InstanceGuardArea
    { is followed by InstanceInlineExpressionSeparatorSymbol
      is followed by InstanceGuardArea }
    [ is followed by InstanceInlineExpressionSeparatorSymbol
      is followed by InstanceElseGuardArea ]
    is followed by InstanceInlineExpressionEndSymbol )
is attached to InstanceAxisSymbol
is attached to AltInlineExpressionArea

InstanceGuardArea ::=
  ( InstanceInvocationArea
    | InstanceGuardOpArea )
  { is followed by InstanceEventArea }
  is attached to InstanceAxisSymbol

/* STATIC SEMANTICS – The instance invocation area shall contain a altstep instance only */

InstanceGuardOpArea ::=
  ( InstanceTimeoutArea
    | InstanceReceiveEventArea
    | InstanceTriggerEventArea
    | InstanceGetcallEventArea
    | InstanceGetreplyOutsideCallEventArea
```

```

| InstanceCatchOutsideCallEventArea
| InstanceCheckEventArea
| InstanceDoneArea )
is attached to InstanceAxisSymbol

```

```

InstanceElseGuardArea ::=
ElseConditionArea
{ is followed by InstanceEventArea }
is attached to InstanceAxisSymbol

```

```

InstanceInterleaveArea ::=
( InstanceInlineExpressionBeginSymbol
  is followed by InstanceInterleaveGuardArea
  { is followed by InstanceInlineExpressionSeparatorSymbol
    is followed by InstanceInterleaveGuardArea }
  is followed by InstanceInlineExpressionEndSymbol )
is attached to InstanceAxisSymbol
is attached to InterleaveInlineExpressionArea

```

```

InstanceInterleaveGuardArea ::=
InstanceGuardOpArea
{ is followed by InstanceEventArea }
is attached to InstanceAxisSymbol

```

/ STATIC SEMANTICS – The instance event area may not contain loop statements, goto, activate, deactivate, stop, return or calls to functions */*

```

InstanceCallArea ::=
( InstanceInlineExpressionBeginSymbol
  [ is followed by InstanceBooleanExpressionConditionArea ]
  [ is followed by InstanceCallOpArea ]
  { is followed by InstanceInlineExpressionSeparatorSymbol
    is followed by InstanceCallGuardArea }
  is followed by InstanceInlineExpressionEndSymbol )
is attached to InstanceAxisSymbol
is attached to CallInlineExpressionArea

```

```

InstanceCallOpArea ::=
InstanceCallEventArea
is followed by SuspensionRegionSymbol
[ is attached to InstanceCallTimerStartArea ]
is attached to InstanceAxisSymbol
is attached to CallInlineExpressionArea

```

SuspensionRegionSymbol ::=

□

```

InstanceCallGuardArea ::=
SuspensionRegionSymbol
[ is attached to InstanceGetreplyWithinCallEventArea
  | InstanceCatchWithinCallEventArea
  | InstanceCatchTimeoutWithinCallEventArea ]
{ is followed by InstanceEventArea }
is attached to InstanceAxisSymbol
is attached to CallInlineExpressionArea

```

A.3.8.2 Inline expressions on ports

```

PortInlineExpressionArea ::=
PortIfArea
| PortForArea
| PortWhileArea
| PortDoWhileArea
| PortAltArea
| PortInterleaveArea
| PortCallArea

```

```

PortIfArea ::=
(PortInlineExpressionBeginSymbol
  { is followed by PortEventArea }
  [ is followed by PortInlineExpressionSeparatorSymbol
    { is followed by PortEventArea } ]
  is followed by PortInlineExpressionEndSymbol )
is attached to PortAxisSymbol
is attached to IfInlineExpressionArea

```

```

PortInlineExpressionBeginSymbol ::=
    VoidSymbol

PortInlineExpressionSeparatorSymbol ::=
    VoidSymbol

PortInlineExpressionEndSymbol ::=
    VoidSymbol

PortForArea ::=
    (PortInlineExpressionBeginSymbol
     { is followed by PortEventArea }
     is followed by PortInlineExpressionEndSymbol )
    is attached to PortAxisSymbol
    is attached to ForInlineExpressionArea

PortWhileArea ::=
    (PortInlineExpressionBeginSymbol
     { is followed by PortEventArea }
     is followed by PortInlineExpressionEndSymbol )
    is attached to PortAxisSymbol
    is attached to WhileInlineExpressionArea

PortDoWhileArea ::=
    ( PortInlineExpressionBeginSymbol
     { is followed by PortEventArea }
     is followed by PortInlineExpressionEndSymbol )
    is attached to PortAxisSymbol
    is attached to DoWhileInlineExpressionArea

PortAltArea ::=
    (PortInlineExpressionBeginSymbol
     [ is followed by PortOutEventArea ]
     { is followed by PortEventArea }
     { is followed by PortInlineExpressionSeparatorSymbol
     [ is followed by PortOutEventArea ]
     { is followed by PortEventArea } }
     is followed by PortInlineExpressionEndSymbol )
    is attached to PortAxisSymbol
    is attached to AltInlineExpressionArea

PortInterleaveArea ::=
    ( PortInlineExpressionBeginSymbol
     [ is followed by PortOutEventArea ]
     { is followed by PortEventArea }
     { is followed by PortInlineExpressionSeparatorSymbol
     [ is followed by PortOutEventArea ]
     { is followed by PortEventArea } }
     is followed by PortInlineExpressionEndSymbol )
    is attached to PortAxisSymbol
    is attached to InterleaveInlineExpressionArea

PortCallArea ::=
    (PortInlineExpressionBeginSymbol
     [ is followed by PortCallInEventArea]
     { is followed by PortEventArea }
     { is followed by PortInlineExpressionSeparatorSymbol
     [ is followed by PortOutEventArea ]
     { is followed by PortEventArea } }
     is followed by PortInlineExpressionEndSymbol )
    is attached to InstanceAxisSymbol
    is attached to CallInlineExpressionArea

```

A.3.8.3 Inline expressions on control instances

```

ControlInlineExpressionArea ::=
    ControlIfArea
    | ControlForArea
    | ControlWhileArea
    | ControlDoWhileArea
    | ControlAltArea
    | ControlInterleaveArea

ControlIfArea ::=
    ( InstanceInlineExpressionBeginSymbol
     [ is followed by ControlEventArea ]
     [ is followed by InstanceInlineExpressionSeparatorSymbol
     is followed by ControlEventArea ]

```

```

is followed by InstanceInlineExpressionEndSymbol )
is attached to InstanceAxisSymbol
is attached to IfInlineExpressionArea

```

```

ControlForArea ::=
  ( InstanceInlineExpressionBeginSymbol
    [ is followed by ControlEventArea ]
    is followed by InstanceInlineExpressionEndSymbol )
is attached to InstanceAxisSymbol
is attached to ForInlineExpressionArea

```

```

ControlWhileArea ::=
  ( InstanceInlineExpressionBeginSymbol
    [ is followed by ControlEventArea ]
    is followed by InstanceInlineExpressionEndSymbol )
is attached to InstanceAxisSymbol
is attached to WhileInlineExpressionArea

```

```

ControlDoWhileArea ::=
  ( InstanceInlineExpressionBeginSymbol
    [ is followed by ControlEventArea ]
    is followed by InstanceInlineExpressionEndSymbol )
is attached to InstanceAxisSymbol
is attached to DoWhileInlineExpressionArea

```

```

ControlAltArea ::=
  ( InstanceInlineExpressionBeginSymbol
    [ is followed by ControlGuardArea ]
    { is followed by InstanceInlineExpressionSeparatorSymbol
      is followed by ControlGuardArea }
    [ is followed by InstanceInlineExpressionSeparatorSymbol
      is followed by ControlElseGuardArea ]
    is followed by InstanceInlineExpressionEndSymbol )
is attached to InstanceAxisSymbol
is attached to AltInlineExpressionArea

```

```

ControlGuardArea ::=
  ( InstanceInvocationArea
    | InstanceTimeoutArea)
  { is followed by ControlEventArea }
is attached to InstanceAxisSymbol

```

/* STATIC SEMANTICS – The instance invocation area shall contain a altstep instance only */

```

ControlElseGuardArea ::=
  ElseConditionArea
  { is followed by ControlEventArea }
is attached to InstanceAxisSymbol

```

```

ControlInterleaveArea ::=
  ( InstanceInlineExpressionBeginSymbol
    [ is followed by ControlInterleaveGuardArea ]
    { is followed by InstanceInlineExpressionSeparatorSymbol
      is followed by ControlInterleaveGuardArea }
    is followed by InstanceInlineExpressionEndSymbol )
is attached to InstanceAxisSymbol
is attached to InterleaveInlineExpressionArea

```

```

ControlInterleaveGuardArea ::=
  InstanceTimeoutArea
  { is followed by ControlEventArea }
is attached to InstanceAxisSymbol

```

/* STATIC SEMANTICS – The instance event area may not contain loop statements, goto, activate, deactivate, stop, return or calls to functions */

A.3.9 Condition

```

ConditionArea ::=
  PortOperationArea

```

```

BooleanExpressionConditionArea ::=
  ConditionSymbol
  contains BooleanExpression
is attached to InstanceConditionBeginSymbol
is attached to InstanceConditionEndSymbol

```

/* STATIC SEMANTICS – Boolean expressions within conditions shall be used as guards within alt and call inline expressions only. They shall be attached to a single test component or control instance only.*/

```

InstanceConditionBeginSymbol ::=
    VoidSymbol

InstanceConditionEndSymbol ::=
    VoidSymbol

DoneArea ::=
    ConditionSymbol
    contains DoneStatement
    is attached to InstanceConditionBeginSymbol
    is attached to InstanceConditionEndSymbol

SetVerdictArea ::=
    ConditionSymbol
    contains SetVerdictText
    is attached to InstanceConditionBeginSymbol
    is attached to InstanceConditionEndSymbol

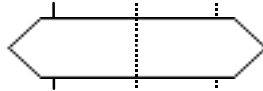
SetVerdictText ::=
    ( SetVerdictKeyword "(" SingleExpression ")" )
    | pass
    | fail
    | inconc
    | none

/* STATIC SEMANTICS - SingleExpression must resolve to a value of type verdict */
/* STATIC SEMANTICS - the SetLocalVerdict shall not be used to assign the value error */
/* STATIC SEMANTICS - if the keywords pass, fail, inconc, and fail are used, the form with the
setverdict keyword shall not be used */
PortOperationArea ::=
    ConditionSymbol
    contains PortOperationText
    is attached to InstanceConditionBeginSymbol
    is attached to InstanceConditionEndSymbol
    is attached to { PortInlineExpressionBeginSymbol }+ set
is attached to { PortInlineExpressionEndSymbol }+ set ]
    is attached to InstancePortOperationArea
    is attached to PortConditionArea

```

/* STATIC SEMANTICS – The condition symbol shall be attached to either all ports or to just one port */

If the condition symbol crosses a port axis symbol of a port which is not involved in this port operation, the port axis symbol is drawn through:



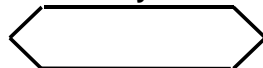
```

PortOperationText ::=
    ClearOpKeyword
    | StartKeyword
    | StopKeyword

ElseConditionArea ::=
    ConditionSymbol
    contains ElseKeyword
    is attached to InstanceAxisSymbol

```

ConditionSymbol ::=



A.3.9.1 Condition on component instances

```

InstanceConditionArea ::=
    InstanceDoneArea
    | InstanceSetVerdictArea
    | InstancePortOperationArea

InstanceBooleanExpressionConditionArea ::=
    InstanceConditionBeginSymbol
    is followed by InstanceConditionEndSymbol
    is attached to InstanceAxisSymbol

```


is attached to BooleanExpressionConditionArea

```
InstanceDoneArea ::=
  InstanceConditionBeginSymbol
  is followed by InstanceConditionEndSymbol
  is attached to InstanceAxisSymbol
  is attached to DoneArea
```

```
InstanceSetVerdictArea ::=
  InstanceConditionBeginSymbol
  is followed by InstanceConditionEndSymbol
  is attached to InstanceAxisSymbol
  is attached to SetVerdictArea
```

```
InstancePortOperationArea ::=
  InstanceConditionBeginSymbol
  is followed by InstanceConditionEndSymbol
  is attached to InstanceAxisSymbol
  is attached to PortOperationArea
```

A.3.9.2 Condition on ports

```
PortConditionArea ::=
  PortConditionBeginSymbol
  is followed by PortConditionEndSymbol
  is attached to PortAxisSymbol
  is attached to PortOperationArea
```

```
PortConditionBeginSymbol ::=
  VoidSymbol
```

```
PortConditionEndSymbol ::=
  VoidSymbol
```

A.3.10 Message-based communication

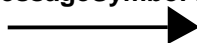
```
SendArea ::=
  MessageSymbol
  [ is associated with Type ]
  is associated with ( [ DerivedDef AssignmentChar ] TemplateBody
    [ ToClause ] )
  is attached to InstanceSendEventArea
  is attached to PortInMsgEventArea
```

*/** STATIC SEMANTICS – A type, if existent, shall be put on top of the message symbol **/*
*/** STATIC SEMANTICS – A derived definition, if existent, shall be put underneath the message symbol **/*
*/** STATIC SEMANTICS – A template shall be put underneath the message symbol **/*
*/** STATIC SEMANTICS – A to clause, if existent, shall be put underneath the message symbol **/*

```
ReceiveArea ::=
  MessageSymbol
  [ is associated with Type ]
  is associated with ( [ [ DerivedDef AssignmentChar ] TemplateBody ]
    [ FromClause ] [ PortRedirect ] )
  is attached to InstanceReceiveEventArea
  is attached to PortOutMsgEventArea
```

*/** STATIC SEMANTICS – A type, if existent, shall be put on top of the message symbol **/*
*/** STATIC SEMANTICS – A derived definition, if existent, shall be put underneath the message symbol **/*
*/** STATIC SEMANTICS – A template, if existent, shall be put underneath the message symbol **/*
*/** STATIC SEMANTICS – A from clause, if existent, shall be put underneath the message symbol **/*
*/** STATIC SEMANTICS – A port redirect, if existent, shall be put underneath the message symbol **/*

MessageSymbol ::=



A.3.10.1 Message-based communication on component instances

```
InstanceSendEventArea ::=
  MessageOutSymbol
  is attached to InstanceAxisSymbol
  is attached to MessageSymbol
```

```
MessageOutSymbol ::=
    VoidSymbol
```

The VoidSymbol is a geometric point without spatial extension.

```
InstanceReceiveEventArea ::=
    MessageInSymbol
    is attached to InstanceAxisSymbol
    is attached to MessageSymbol
```

```
MessageInSymbol ::=
    VoidSymbol
```

A.3.10.2 Message-based communication on port instances

```
PortInMsgEventArea ::=
    MessageInSymbol
    is attached to PortAxisSymbol
    is attached to MessageSymbol
```

```
PortOutMsgEventArea ::=
    MessageOutSymbol
    is attached to PortAxisSymbol
    is attached to MessageSymbol
```

A.3.11 Signature-based communication

```
NonBlockingCallArea ::=
    MessageSymbol
    is associated with CallKeyword [ Signature ]
    is associated with ( [ DerivedDef AssignmentChar ] TemplateBody
                        [ ToClause ] )
    is attached to InstanceCallEventArea
    is attached to PortCallInEventArea
```

/* STATIC SEMANTICS – A signature, if existent, shall be put on top of the message symbol */
 /* STATIC SEMANTICS – A derived definition, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A template shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A to clause, if existent, shall be put underneath the message symbol */

```
GetcallArea ::=
    MessageSymbol
    is associated with GetcallKeyword [ Signature ]
    is associated with ( [ [ DerivedDef AssignmentChar ] TemplateBody ]
                        [ FromClause ] [ PortRedirectWithParam ] )
    is attached to InstanceGetcallEventArea
    is attached to PortGetcallOutEventArea
```

/* STATIC SEMANTICS – A signature, if existent, shall be put on top of the message symbol */
 /* STATIC SEMANTICS – A derived definition, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A template, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A from clause, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A port redirect, if existent, shall be put underneath the message symbol */

```
ReplyArea ::=
    MessageSymbol
    is associated with ReplyKeyword [ Signature ]
    is associated with ( [ DerivedDef AssignmentChar ] TemplateBody
                        [ ReplyValue ] [ ToClause ] )
    is attached to InstanceReplyEventArea
    is attached to PortReplyInEventArea
```

/* STATIC SEMANTICS – A signature, if existent, shall be put on top of the message symbol */
 /* STATIC SEMANTICS – A derived definition, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A template shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A reply value, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A to clause, if existent, shall be put underneath the message symbol */

```
GetreplyWithinCallArea ::=
    MessageSymbol
    is attached to SuspensionRegionSymbol
    is associated with GetreplyKeyword [ Signature ]
    is associated with ( [ [ DerivedDef AssignmentChar ] TemplateBody ]
                        [ ValueMatchSpec ]
                        [ FromClause ] [ PortRedirectWithParam ] )
    is attached to InstanceGetreplyEventArea
    is attached to PortGetreplyOutEventArea
```

/* STATIC SEMANTICS – A signature, if existent, shall be put on top of the message symbol */
 /* STATIC SEMANTICS – A derived definition, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A template, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A value match specification, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A from clause, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A port redirect, if existent, shall be put underneath the message symbol */

```
GetreplyOutsideCallArea ::=
  MessageSymbol
  is associated with GetreplyKeyword [ Signature ]
  is associated with ( [ [ DerivedDef AssignmentChar ] TemplateBody ]
                    [ ValueMatchSpec ]
                    [ FromClause ] [ PortRedirectWithParam ] )
  is attached to InstanceGetreplyEventArea
  is attached to PortGetreplyOutEventArea
```

/* STATIC SEMANTICS – A signature, if existent, shall be put on top of the message symbol */
 /* STATIC SEMANTICS – A derived definition, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A template, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A value match specification, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A from clause, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A port redirect, if existent, shall be put underneath the message symbol */

```
RaiseArea ::=
  MessageSymbol
  is associated with RaiseKeyword Signature [ ',' Type ]
  is associated with ( [ [ DerivedDef AssignmentChar ] TemplateBody
                    [ ToClause ] ] )
  is attached to InstanceRaiseEventArea
  is attached to PortRaiseInEventArea
```

/* STATIC SEMANTICS – A signature shall be put on top of the message symbol */
 /* STATIC SEMANTICS – An exception type, if existent, shall be put on top of the message symbol */
 /* STATIC SEMANTICS – A derived definition, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A template shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A to clause, if existent, shall be put underneath the message symbol */

```
CatchWithinCallArea ::=
  MessageSymbol
  is attached to SuspensionRegionSymbol
  is associated with CatchKeyword Signature [ ',' Type ]
  is associated with ( [ [ DerivedDef AssignmentChar ] TemplateBody ]
                    [ FromClause ] [ PortRedirect ] )
  is attached to InstanceCatchEventArea
  is attached to PortCatchOutEventArea
```

/* STATIC SEMANTICS – A signature shall be put on top of the message symbol */
 /* STATIC SEMANTICS – An exception type, if existent, shall be put on top of the message symbol */
 /* STATIC SEMANTICS – A derived definition, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A template, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A from clause, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A port redirect, if existent, shall be put underneath the message symbol */

```
CatchOutsideCallArea ::=
  MessageSymbol
  is associated with CatchKeyword Signature [ ',' Type ]
  is associated with ( [ [ DerivedDef AssignmentChar ] TemplateBody ]
                    [ FromClause ] [ PortRedirect ] )
  is attached to InstanceCatchEventArea
  is attached to PortCatchOutEventArea
```

/* STATIC SEMANTICS – A signature shall be put on top of the message symbol */
 /* STATIC SEMANTICS – An exception type, if existent, shall be put on top of the message symbol */
 /* STATIC SEMANTICS – A derived definition, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A template, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A from clause, if existent, shall be put underneath the message symbol */
 /* STATIC SEMANTICS – A port redirect, if existent, shall be put underneath the message symbol */

A.3.11.1 Signature-based communication on component instances

```
InstanceBlockingCallEventArea ::=
  InstanceSendEventArea
  [ is attached to InstanceCallTimerStartArea ]
  is attached to SuspensionRegionSymbol
```

```
InstanceCallTimerStartArea ::=
  CallTimerStartSymbol
```

```

is associated with TimerValue
is attached to InstanceAxisSymbol
is attached to SuspensionRegionSymbol
[ is attached to CallTimeoutSymbol3 ]

```

CallTimerStartSymbol ::=



```

InstanceNonBlockingCallEventArea ::=
    InstanceSendEventArea

InstanceGetcallEventArea ::=
    InstanceReceiveEventArea

InstanceReplyEventArea ::=
    InstanceSendEventArea

InstanceGetreplyWithinCallEventArea ::=
    InstanceReceiveEventArea
    is attached to SuspensionRegionSymbol

InstanceGetreplyOutsideCallEventArea ::=
    InstanceReceiveEventArea

InstanceRaiseEventArea ::=
    InstanceSendEventArea

InstanceCatchWithinCallEventArea ::=
    InstanceReceiveEventArea
    is attached to SuspensionRegionSymbol

InstanceCatchTimeoutWithinCallEventArea ::=
    CallTimeoutSymbol
    is attached to SuspensionRegionSymbol
    is attached to InstanceAxisSymbol

```

CallTimeoutSymbol ::=



```

InstanceCatchOutsideCallEventArea ::=
    InstanceReceiveEventArea

```

A.3.11.2 Signature-based communication on ports

```

PortGetcallOutEventArea ::=
    PortOutMsgEventArea

PortGetreplyOutEventArea ::=
    PortOutMsgEventArea

PortCatchOutEventArea ::=
    PortOutMsgEventArea

PortCallInEventArea ::=
    PortInMsgEventArea

PortReplyInEventArea ::=
    PortInMsgEventArea

PortRaiseInEventArea ::=
    PortInMsgEventArea

```

A.3.12 Trigger and check

A.3.12.1 Trigger and check on component instances

```

TriggerArea ::=
    MessageSymbol
    is associated with ( TriggerOpKeyword [ Type ] )
    is associated with ( [ [ DerivedDef AssignmentChar ] TemplateBody ]
        [ FromClause ] [ PortRedirect ] )
    is attached to ReceiveEventArea
    is attached to PortOutMsgEventArea

```

```

/* STATIC SEMANTICS – The trigger keyword shall be put on top of the message symbol */
/* STATIC SEMANTICS – A type, if existent, shall be put on top of the message symbol */
/* STATIC SEMANTICS – A derived definition, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – A template, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – A from clause, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – A port redirect, if existent, shall be put underneath the message symbol */

```

```

CheckArea ::=
    MessageSymbol
    is associated with ( CheckOpKeyword [ CheckOpInformation ] )
    is associated with CheckData
    is attached to ReceiveEventArea
    is attached to PortOutMsgEventArea

```

```

/* STATIC SEMANTICS – The check keyword shall be put on top of the message symbol */
/* STATIC SEMANTICS – The check op information, if existent, shall be put on top of the message symbol */
/* STATIC SEMANTICS – The check data, if existent, shall be put underneath the message symbol */

```

```

CheckOpInformation ::=
    Type
    | ( GetCallOpKeyword [ Signature ] )
    | ( GetReplyOpKeyword [ Signature ] )
    | ( CatchOpKeyword Signature [ Type ] )

```

```

CheckData ::=
    ( [ [ DerivedDef AssignmentChar ] TemplateBody [ ValueMatchSpec ] ]
      [ FromClause ] [ PortRedirect | PortRedirectWithParam ] )
    | ( [ [ FromClause ] [ PortRedirectSymbol SenderSpec ] ] )

```

```

/* STATIC SEMANTICS – A value matching specification shall be used in combination with getreply only */
/* STATIC SEMANTICS – A port redirect with parameters shall be used in combination with getcall and getreply only */

```

```

InstanceTriggerEventArea ::=
    InstanceReceiveEventArea

```

```

InstanceCheckEventArea ::=
    InstanceReceiveEventArea

```

A.3.12.2 Trigger and check on port instances

```

PortTriggerOutEventArea ::=
    PortOutMsgEventArea

```

```

PortCheckOutEventArea ::=
    PortOutMsgEventArea

```

A.3.13 Handling of communication from any port

```

InstanceFoundEventArea ::=
    FoundSymbol
    contains FoundEvent
    is attached to InstanceAxisSymbol

```

```

/* STATIC SEMANTICS – The label identifier shall be placed inside the circle of the labelling symbol */

```

```

FoundEvent ::=
    FoundMessage
    | FoundTrigger
    | FoundGetCall
    | FoundGetReply
    | FoundCatch
    | FoundCheck

```

```

FoundMessage ::=
    FoundSymbol
    [ is associated with Type ]
    is associated with ( [ [ DerivedDef AssignmentChar ] TemplateBody ]
                        [ FromClause ] [ PortRedirect ] )
    is attached to InstanceAxisSymbol

```

```

/* STATIC SEMANTICS – A type, if existent, shall be put on top of the message symbol */
/* STATIC SEMANTICS – A derived definition, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – A template, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – A from clause, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – A port redirect, if existent, shall be put underneath the message symbol */

```

```

FoundTrigger ::=
    FoundSymbol
    is associated with ( TriggerOpKeyword [ Type ] )
    is associated with ( [ [ DerivedDef AssignmentChar ] TemplateBody ]
                        [ FromClause ] [ PortRedirect ] )
    is attached to InstanceAxisSymbol

/* STATIC SEMANTICS – The trigger keyword shall be put on top of the message symbol */
/* STATIC SEMANTICS – A type, if existent, shall be put on top of the message symbol */
/* STATIC SEMANTICS – A derived definition, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – A template, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – A from clause, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – A port redirect, if existent, shall be put underneath the message symbol */

FoundGetCall ::=
    FoundSymbol
    is associated with GetcallKeyword [ Signature ]
    is associated with ( [ [ DerivedDef AssignmentChar ] TemplateBody ]
                        [ FromClause ] [ PortRedirectWithParam ] )
    is attached to InstanceAxisSymbol

/* STATIC SEMANTICS – A signature, if existent, shall be put on top of the message symbol */
/* STATIC SEMANTICS – A derived definition, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – A template, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – A from clause, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – A port redirect, if existent, shall be put underneath the message symbol */

FoundGetReply ::=
    FoundSymbol
    is associated with GetreplyKeyword [ Signature ]
    is associated with ( [ [ DerivedDef AssignmentChar ] TemplateBody ]
                        [ ValueMatchSpec ]
                        [ FromClause ] [ PortRedirectWithParam ] )
    is attached to InstanceAxisSymbol

/* STATIC SEMANTICS – A signature, if existent, shall be put on top of the message symbol */
/* STATIC SEMANTICS – A derived definition, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – A template, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – A value match specification, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – A from clause, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – A port redirect, if existent, shall be put underneath the message symbol */

FoundCatch ::=
    FoundSymbol
    is associated with CatchKeyword Signature [ ', ' Type ]
    is associated with ( [ [ DerivedDef AssignmentChar ] TemplateBody ]
                        [ FromClause ] [ PortRedirect ] )
    is attached to InstanceAxisSymbol

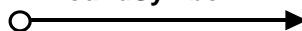
/* STATIC SEMANTICS – A signature shall be put on top of the message symbol */
/* STATIC SEMANTICS – An exception type, if existent, shall be put on top of the message symbol */
/* STATIC SEMANTICS – A derived definition, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – A template, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – A from clause, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – A port redirect, if existent, shall be put underneath the message symbol */

FoundCheck ::=
    FoundSymbol
    is associated with ( CheckOpKeyword [ CheckOpInformation ] )
    is associated with CheckData
    is attached to ReceiveEventArea
    is attached to InstanceAxisSymbol

/* STATIC SEMANTICS – The check keyword shall be put on top of the message symbol */
/* STATIC SEMANTICS – The check op information, if existent, shall be put on top of the message symbol */
/* STATIC SEMANTICS – The check data, if existent, shall be put underneath the message symbol */

```

FoundSymbol ::=

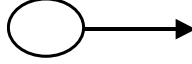


A.3.14 Labelling

```
InstanceLabellingArea ::=
  LabellingSymbol
  contains LabelIdentifier
  is attached to InstanceAxisSymbol
```

/* STATIC SEMANTICS – The label identifier shall be placed inside the circle of the labelling symbol */

LabellingSymbol ::=

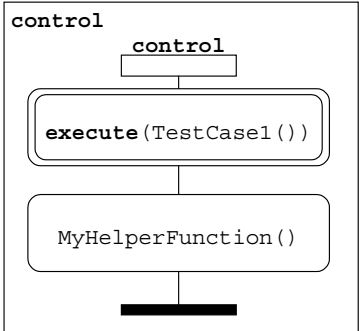
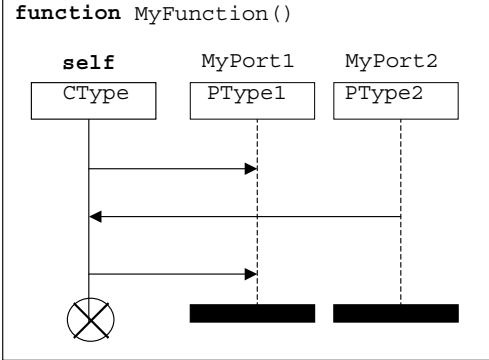
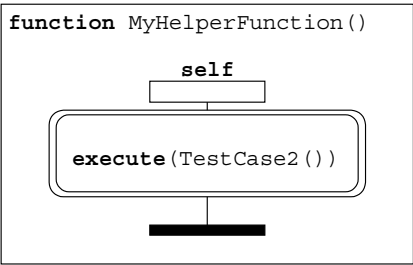
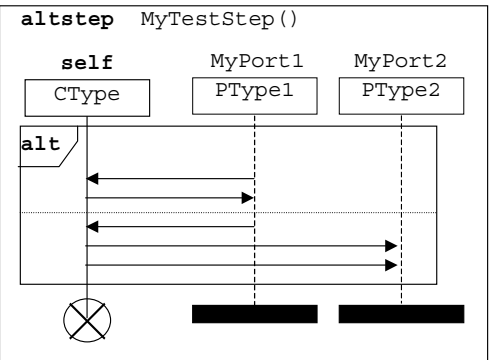


Annex B

Reference guide for GFT

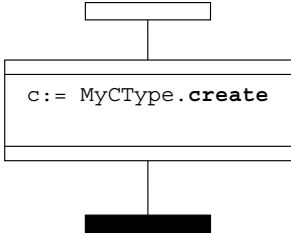
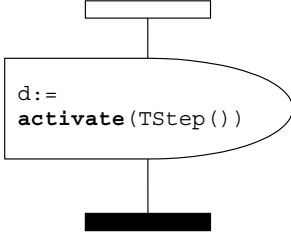
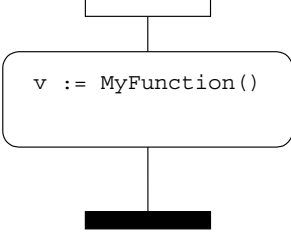
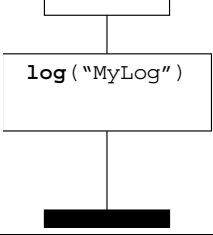
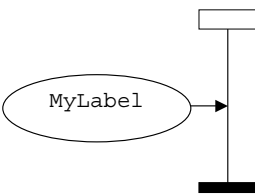
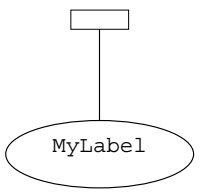
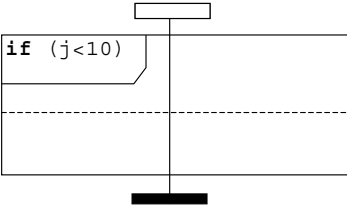
This annex lists the main TTCN-3 language elements and their representation in GFT. For a complete description of the GFT symbols and their use, please refer to the main text.

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
Module definitions			
TTCN-3 module definition	module		No special GFT symbol, i.e., the core language or another presentation format may be used.
Import of definitions from other module	import		No special GFT symbol, i.e., the core language or another presentation format may be used.
Grouping of definitions	group		No special GFT symbol, i.e., the core language or another presentation format may be used.
Data type definitions	type		No special GFT symbol, i.e., the core language or another presentation format may be used.
Communication port definitions	port		No special GFT symbol, i.e., the core language or another presentation format may be used.
Test component definitions	component		No special GFT symbol, i.e., the core language or another presentation format may be used.
Signature definitions	signature		No special GFT symbol, i.e., the core language or another presentation format may be used.
External function/constant definitions	external		No special GFT symbol, i.e., the core language or another presentation format may be used.
Constant definitions	const	<code>const integer MyConst := 5;</code>	Textual constant declaration in the header of a control, test case, test step or function diagram.
			Local constant declaration in an action box.
Data/signature template definitions	template		No special GFT symbol, i.e., the core language or another presentation format may be used.

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
Control definitions	control		GFT control diagram represents the control part of a TTCN-3 module.
Function definitions	function		GFT function diagrams are used to represent functions.
			GFT function diagrams may be defined to structure the behaviour of the control part of a TTCN-3 module.
Altstep definitions	altstep		GFT altstep diagrams are used to represent altsteps.

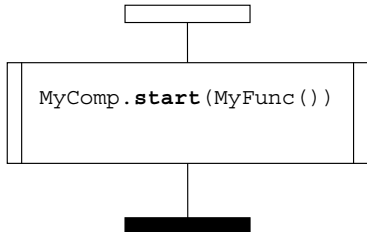
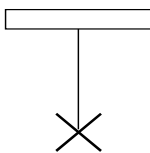
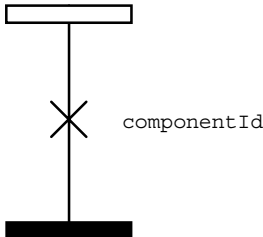
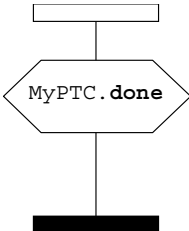
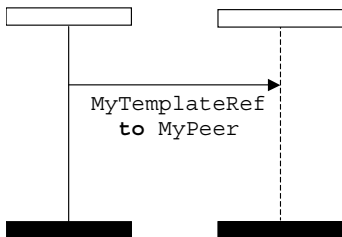
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
Test case definitions	testcase		GFT test case diagrams are used to represent test cases.
Usage of component instances and ports			
Port instance			A Port in a test case, test step and function diagram is represented by an instance with a dashed instance line. The port name is specified above and the (optional) port type is described within the instance header.
Test component instance			<p>An mtc instance represents the main test component in a test case diagram.</p> <p>A self instance represents a test component in a test step or function diagram.</p> <p>A control instance represents the instance that executes the module control part in a control diagram.</p>
Declarations			
Variable declarations	var	<pre>var integer MyVar := 5</pre>	Textual variable declaration in the header of a control, test case, test step or function diagram.
			Variable declaration in an action box.
			Variable declaration within a test case execution symbol.

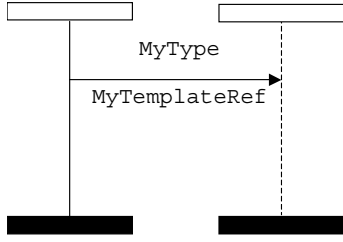
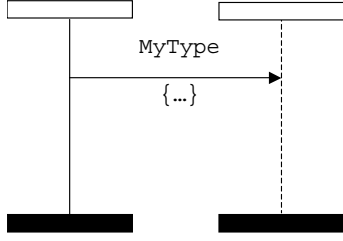
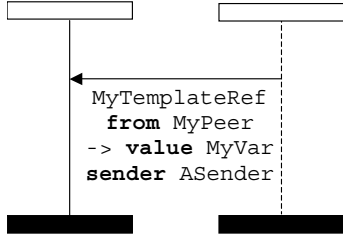
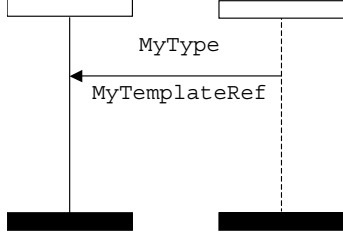
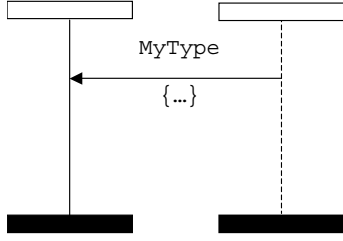
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
			Variable declaration within a test component creation symbol.
			Variable declaration within a default activation symbol.
			Variable declaration within a reference symbol.
Timer declarations	timer	timer MyTimer	Textual timer declaration in the header of a control, test case, test step or function diagram.
			Timer declaration in an action box.
Basic program statements			
Expressions	(...)		No special GFT symbol, i.e., the core language or another presentation format may be used.
Assignments	:=		Assignment in an action box.
			Assignment within a test case execution symbol.

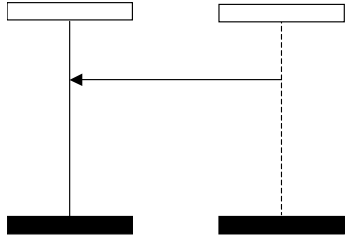
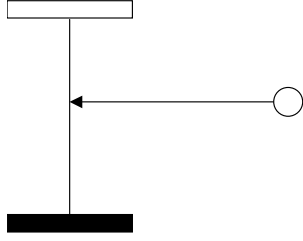
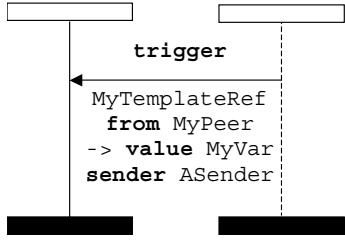
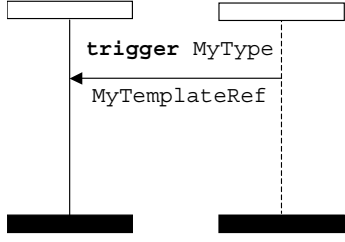
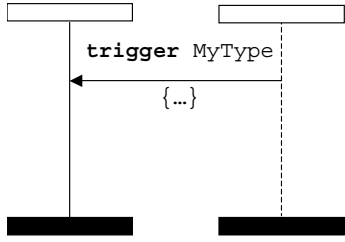
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
			Assignment within a test component creation symbol.
			Assignment within a default activation symbol.
			Assignment within a reference symbol.
Logging	log		The log statement is put into an action box.
Label and Goto	label		Definition of a label.
	goto		Go to label.
If-else	if (...) {...} else {...}		

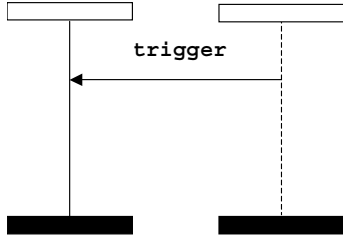
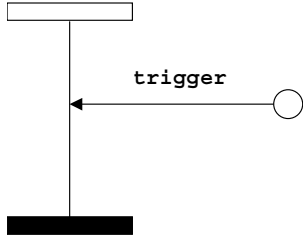
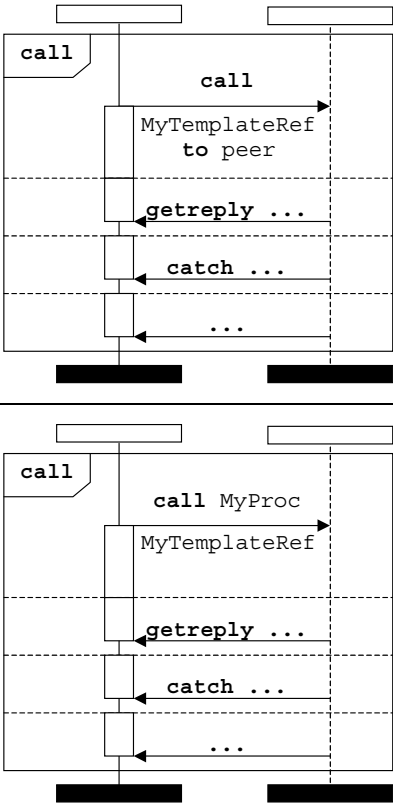
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
For loop	<code>for (...) {...}</code>		
While loop	<code>while (...)</code> <code>{...}</code>		
Do while loop	<code>do {...}</code> <code>while (...)</code>		
Behavioural program statements			
Alternative behaviour	<code>alt {...}</code>		
Repeat	<code>repeat</code>		To be used within alternative behaviour and test steps.
Interleaved behaviour	<code>interleave</code> <code>{...}</code>		
Activate a default	<code>activate</code>		The activate statement is put into a default symbol.

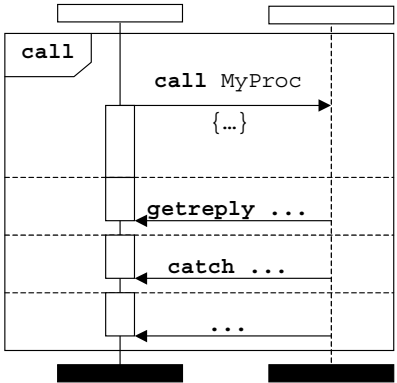
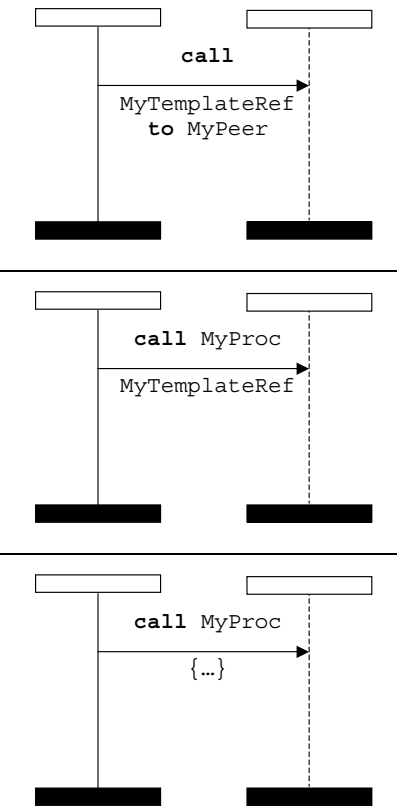
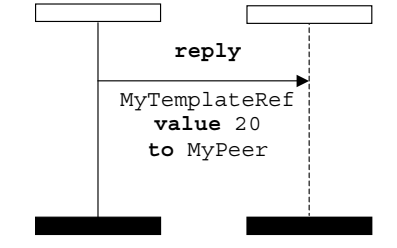
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
Deactivate a default	deactivate		The deactivate statement is put into a default symbol.
Returning control	return	<p style="text-align: center;">MyValue</p>	The optional return value is attached to the return symbol.
Configuration operations			
Create parallel test component	create		The create statement is put into a test component creation symbol.
Connect component to component	connect		The connect statement is put into an action box.
Disconnect two components	disconnect		The disconnect statement is put into an action box.
Map port to test system interface	map		The map statement is put into an action box.
Unmap port from test system interface	unmap		The unmap statement is put into an action box.

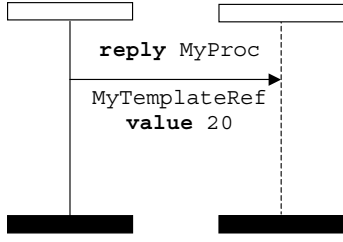
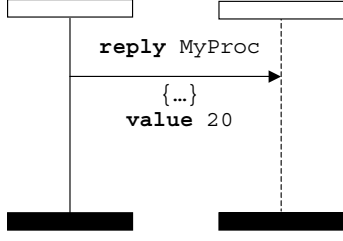
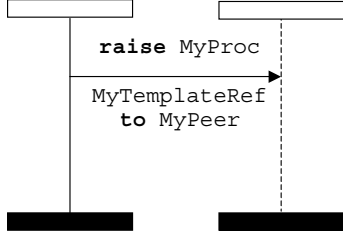
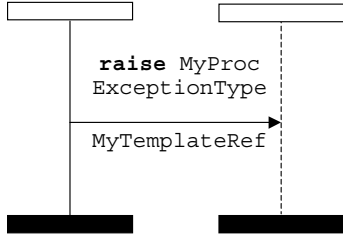
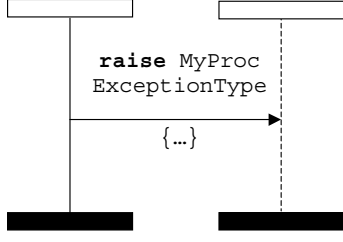
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
Get MTC address	mtc		No special GFT symbol, used within statements, expressions or as test component identifier.
Get test system interface address	system		No special GFT symbol, used within statements or expressions.
Get own address	self		No special GFT symbol, used within statements, expressions or as test component identifier.
Start execution of test component	start		The start statement is put into a start symbol.
Stop execution of a test component by itself	stop		The termination of mtc terminates also all the other test components. Port instances cannot be stopped.
Stop execution of another test component			The component identifier is put near to the stop symbol.
Check termination of a PTC	running		No special GFT symbol, used within expressions.
Wait for termination of a PTC	done		The done statement is put into a condition symbol.
Communication operations			
Send message	send		Send a message defined by a template reference but without type information. The receiver is identified uniquely by the (optional) to -directive.

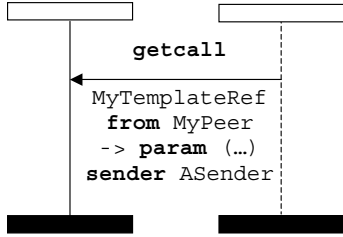
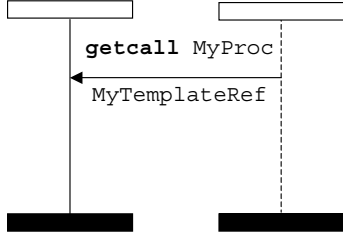
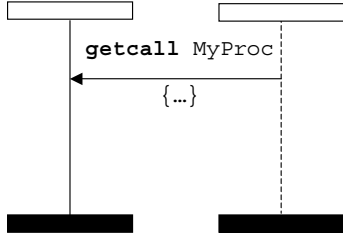
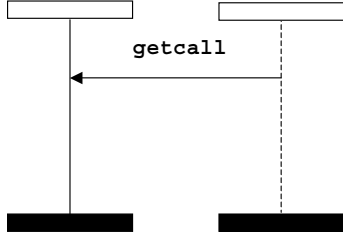
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
			<p>Send a message defined by a template reference and with type information.</p> <p>An (optional) to-directive may be present to identify the peer entity uniquely.</p>
			<p>Send a message defined by an inline template definition.</p> <p>An (optional) to-directive may be present to identify the peer entity uniquely.</p>
Receive message	receive		<p>Receive a message with a value defined by a template reference but without type information.</p> <p>The (optional) from-directive denotes that the sender of the message shall be identified by variable <i>MyPeer</i>.</p> <p>The (optional) value-directive assigns received message to variable <i>MyVar</i>.</p> <p>The (optional) sender-directive retrieves the identifier of the sender and stores it in variable <i>ASender</i>.</p>
			<p>Receive a message with a value defined by a template reference and with type information.</p> <p>Optional from-, value- and sender-directives may be present to identify the sender of the message, to assign the message to a variable or to retrieve the identifier of the peer entity.</p>
			<p>Receive a message with a value defined by an inline template definition.</p> <p>Optional from-, value- and sender-directives may be present to identify the sender of the message, to assign the message to a variable or to retrieve the identifier of the peer entity.</p>

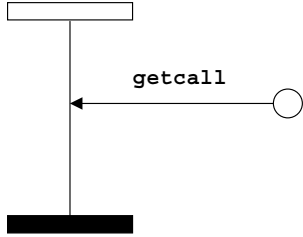
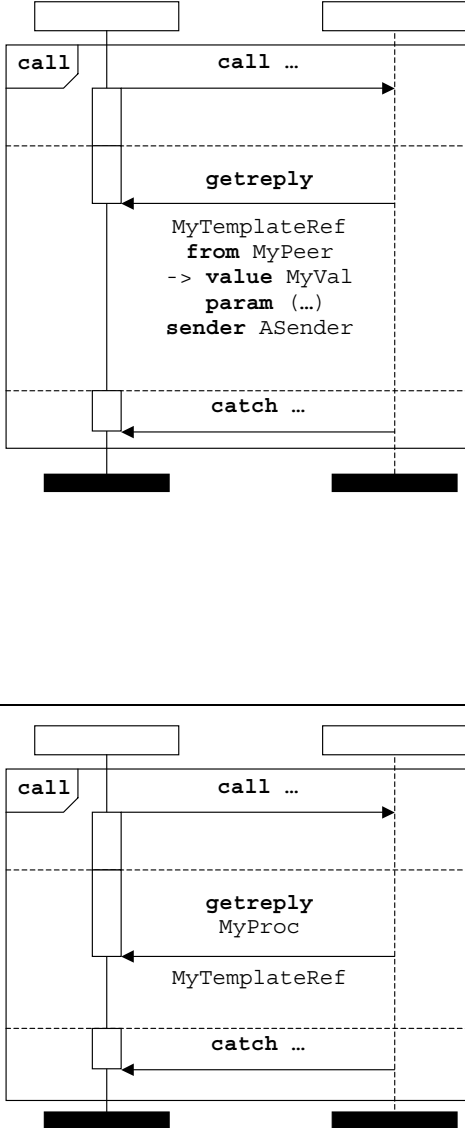
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
			<p>Receive any message (no value and no type is specified).</p> <p>Optional from-, value- and sender- directives may be present to identify the sender of the message, to assign the message to a variable or to retrieve the identifier of the peer entity.</p>
			<p>Receive any message (no value and no type is specified) from any port.</p> <p>The message value to be received from any port may be restricted by means referring to templates or by using inline templates.</p> <p>Optional from-, value- and sender- directives may be present to identify the sender of the message, to assign the message to a variable or to retrieve the identifier of the peer entity.</p>
Trigger message	trigger		<p>Trigger on a message with a value defined by a template reference but without type information.</p> <p>The (optional) from- directive denotes that the sender of the message shall be identified by variable MyPeer.</p> <p>The (optional) value- directive assigns received message to variable MyVar.</p> <p>The (optional) sender- directive retrieves the identifier of the sender and stores it in variable ASender.</p>
			<p>Trigger on a message with a value defined by a template reference and with type information.</p> <p>Optional from-, value- and sender- directives may be present to identify the sender of the message, to assign the message to a variable or to retrieve the identifier of the peer entity.</p>
			<p>Trigger on a message with a value defined by an inline template definition.</p> <p>Optional from-, value- and sender- directives may be present to identify the sender of the message, to assign the message to a variable or to retrieve the identifier of the peer entity.</p>

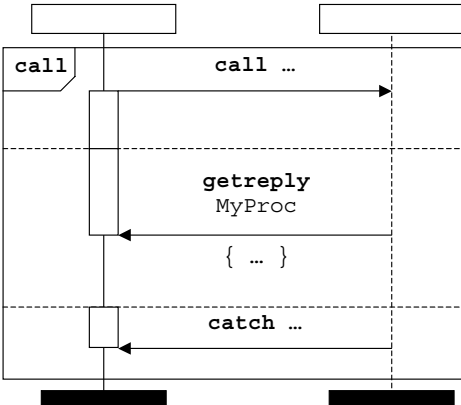
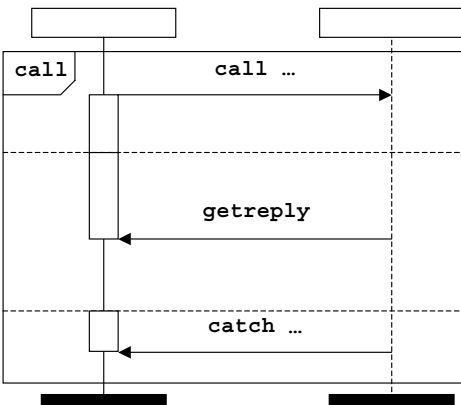
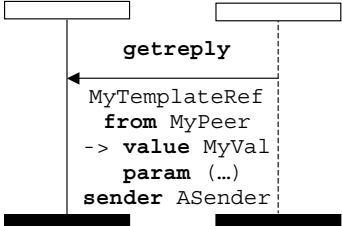
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
			<p>Trigger on any message (no value and no type is specified).</p> <p>Optional from-, value- and sender- directives may be present to identify the sender of the message, to assign the message to a variable (of type anytype) and to retrieve the identifier of the peer entity.</p>
			<p>Trigger on any message (no value and no type is specified) from any port.</p> <p>The value of the message that shall cause the trigger from any port may be restricted by means referring to templates or by using inline templates.</p> <p>Optional from-, value- and sender- directives may be present to identify the sender of the message, to assign the message to a variable (of type anytype) and to retrieve the identifier of the peer entity.</p>
Invoke blocking procedure call	call		<p>Invoking a blocking procedure by using a signature template.</p> <p>The receiver is identified uniquely by the (optional) to-directive.</p> <p>The call body, i.e., possible getreply and catch operations, is shown schematically only.</p> <p>Invoking a blocking procedure by using a signature template and signature information.</p> <p>An (optional) to-directive may be present to identify the peer entity uniquely.</p> <p>The call body, i.e., possible getreply and catch operations, is shown schematically only.</p>

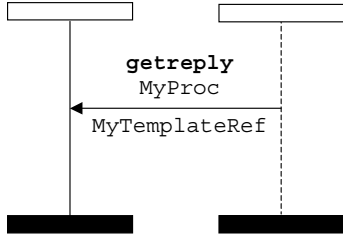
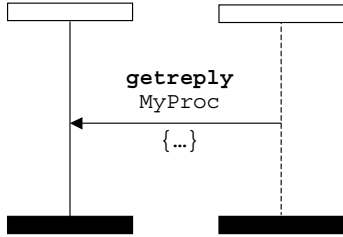
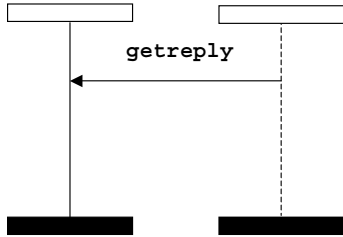
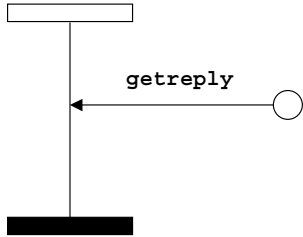
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
			<p>Invoking a blocking procedure by using an inline template.</p> <p>An (optional) to-directive may be present to identify the peer entity uniquely.</p> <p>The call body, i.e., possible getreply and catch operations, is shown schematically only.</p>
Invoke non-blocking procedure call	call		<p>Call a remote procedure, the call is defined by a template reference but without signature information.</p> <p>The receiver is identified uniquely by the (optional) to-directive.</p> <p>Call the remote procedure MyProc. The call is defined by a template reference.</p> <p>An (optional) to-directive may be present to identify the peer entity uniquely.</p> <p>Call the remote procedure MyProc. The call is defined by an inline template.</p> <p>An (optional) to-directive may be present to identify the peer entity uniquely.</p>
Reply to procedure call from remote entity	reply		<p>Reply to a remote procedure call. The reply is defined by a template reference and the possible return value (value-directive).</p> <p>NOTE – The signature information is part of the template definition.</p> <p>The receiver is identified uniquely by the (optional) to-directive.</p>

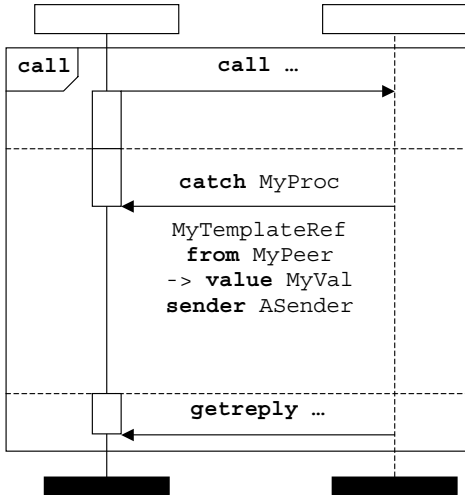
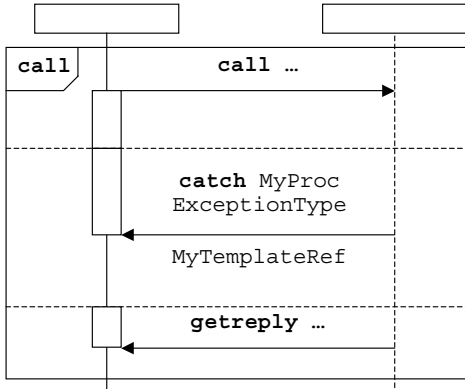
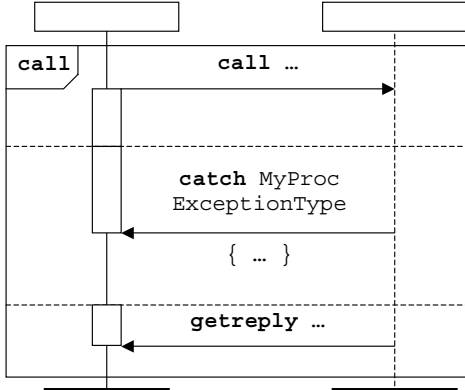
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
		 <p>A GFT diagram showing a call from a peer entity (left) to a peer entity (right). The call is labeled <code>reply MyProc</code>. The message contains <code>MyTemplateRef</code> and <code>value 20</code>. The receiver entity is shown with a dashed line, indicating it is a peer entity.</p>	<p>Reply to a remote procedure call of <code>MyProc</code>. The reply is defined by a template reference and the possible return value (value-directive). An (optional) to-directive may be present to identify the peer entity uniquely.</p>
		 <p>A GFT diagram showing a call from a peer entity (left) to a peer entity (right). The call is labeled <code>reply MyProc</code>. The message contains an inline template <code>{...}</code> and <code>value 20</code>. The receiver entity is shown with a dashed line, indicating it is a peer entity.</p>	<p>Reply to a remote procedure call of <code>MyProc</code>. The reply is defined by an inline template and the possible return value (value-directive). An (optional) to-directive may be present to identify the peer entity uniquely.</p>
Raise exception (to an accepted call)	raise	 <p>A GFT diagram showing a call from a peer entity (left) to a peer entity (right). The call is labeled <code>raise MyProc</code>. The message contains <code>MyTemplateRef</code> and <code>to MyPeer</code>. The receiver entity is shown with a dashed line, indicating it is a peer entity.</p>	<p>Raise an exception to an accepted call of <code>MyProc</code>. The exception is defined by a template reference. NOTE – The type of the exception is defined within the template definition. The receiver is identified uniquely by the (optional) to-directive.</p>
		 <p>A GFT diagram showing a call from a peer entity (left) to a peer entity (right). The call is labeled <code>raise MyProc</code>. The message contains <code>ExceptionType</code> and <code>MyTemplateRef</code>. The receiver entity is shown with a dashed line, indicating it is a peer entity.</p>	<p>Raise an exception to an accepted call of <code>MyProc</code>. The exception is defined by its (optional) type and a template reference. An (optional) to-directive may be present to identify the peer entity uniquely.</p>
		 <p>A GFT diagram showing a call from a peer entity (left) to a peer entity (right). The call is labeled <code>raise MyProc</code>. The message contains <code>ExceptionType</code> and an inline template <code>{...}</code>. The receiver entity is shown with a dashed line, indicating it is a peer entity.</p>	<p>Raise an exception to an accepted call of <code>MyProc</code>. The exception is defined by its type and an inline template. An (optional) to-directive may be present to identify the peer entity uniquely.</p>

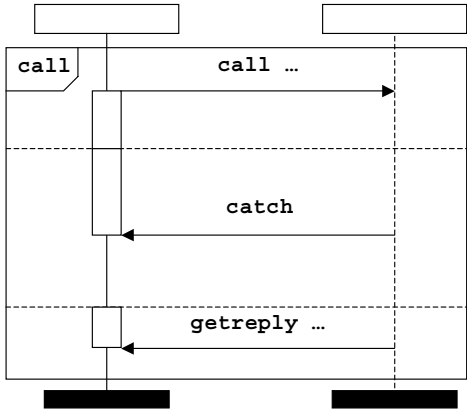
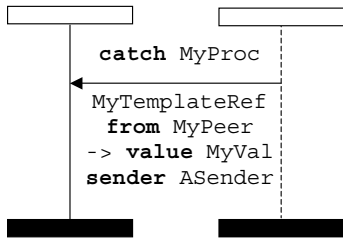
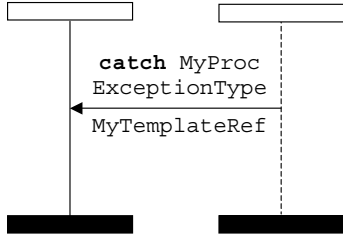
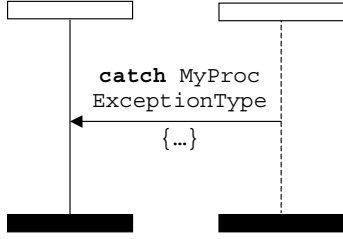
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
Accept procedure call from remote entity	getcall	 <pre> sequenceDiagram participant Remote as [Remote Entity] participant Local as [Local Entity] Remote->>Local: getcall Note over Remote,Local: MyTemplateRef Note over Remote,Local: from MyPeer Note over Remote,Local: -> param (...) Note over Remote,Local: sender ASender </pre>	<p>Accept a procedure call from a remote entity. The call signature has to match the conditions defined by the template reference.</p> <p>NOTE – The signature information is part of the template definition.</p> <p>The (optional) from-directive denotes that the sender of the call shall be identified by variable MyPeer.</p> <p>The (optional) param-directive assigns in-parameter values to Variables.</p> <p>The (optional) sender-directive retrieves the identifier of the sender and stores it in variable ASender.</p>
		 <pre> sequenceDiagram participant Remote as [Remote Entity] participant Local as [Local Entity] Remote->>Local: getcall MyProc Note over Remote,Local: MyTemplateRef </pre>	<p>Accept a procedure call from a remote entity. The call signature has to match the conditions defined by signature reference and the template reference.</p> <p>Optional from-, param- and sender-directives may be present to identify the sender of the call, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</p>
		 <pre> sequenceDiagram participant Remote as [Remote Entity] participant Local as [Local Entity] Remote->>Local: getcall MyProc Note over Remote,Local: {...} </pre>	<p>Accept a procedure call from a remote entity. The call signature has to match the conditions defined by signature reference and the inline template definition.</p> <p>Optional from-, param- and sender-directives may be present to identify the sender of the call, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</p>
		 <pre> sequenceDiagram participant Remote as [Remote Entity] participant Local as [Local Entity] Remote->>Local: getcall </pre>	<p>Accept any procedure call from any remote entity.</p> <p>Optional from- and sender-directives may be present to identify the sender of the call or to retrieve the identifier of the peer entity.</p>

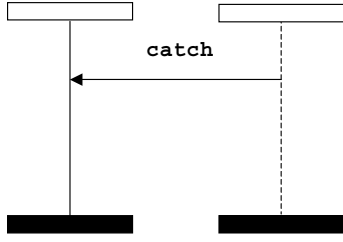
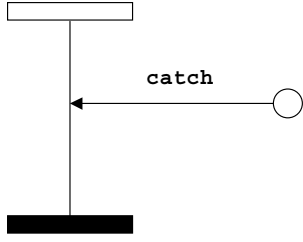
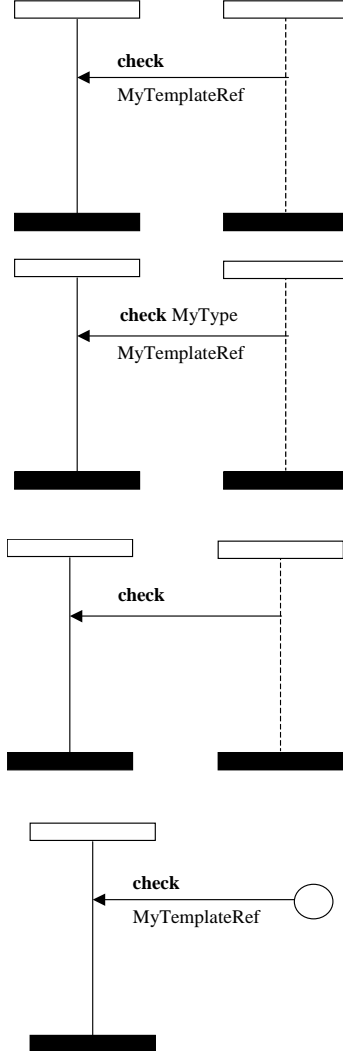
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
			<p>Accept any procedure call from any remote entity at any port.</p> <p>The call to be received from any port may be restricted by means referring to templates or by using inline templates.</p> <p>Optional from-, param- and sender-directives may be present to identify the sender of the call, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</p>
Handle response from a previous blocking call	getreply		<p>Receive a response from a blocking call. The reply has to match the conditions defined by the template reference.</p> <p>NOTE – The signature information is part of the template definition.</p> <p>The (optional) from-directive denotes that the sender of the call shall be identified by variable <code>MyPeer</code>.</p> <p>The (optional) value-directive assigns the possible return value of the procedure to variable <code>MyVal</code>.</p> <p>The (optional) param-directive assigns out-parameter values to Variables.</p> <p>The (optional) sender-directive retrieves the identifier of the sender and stores it in variable <code>ASender</code>.</p> <p>Receive a response from a blocking call. The reply has to match the conditions defined by signature reference and the template reference.</p> <p>Optional from-, value-, param- and sender-directives may be present to identify the sender of the reply, to retrieve the return value of the procedure, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</p>

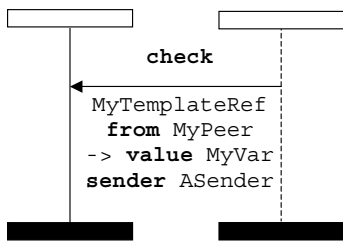
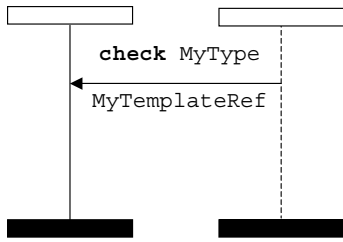
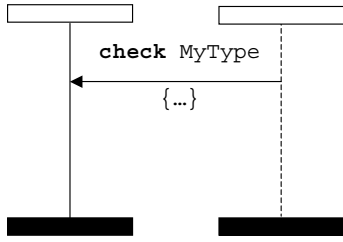
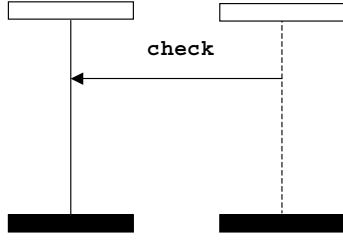
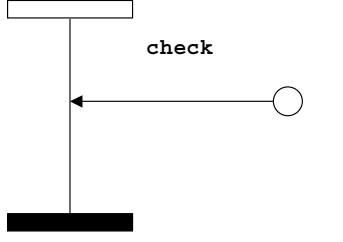
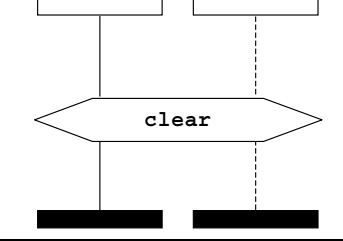
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
			<p>Receive a response from a blocking call. The reply has to match the conditions defined by signature reference and the inline template definition.</p> <p>Optional from-, value-, param- and sender-directives may be present to identify the sender of the reply, to retrieve the return value of the procedure, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</p>
			<p>Accept any response from a blocking call.</p>
<p>Handle response from a previous non-blocking call or independent from a call</p>	<p>getreply</p>		<p>Receive a response from a previous call. The reply has to match the conditions defined by the template reference.</p> <p>NOTE – The signature information is part of the template definition.</p> <p>The (optional) from-directive denotes that the sender of the call shall be identified by variable MyPeer.</p> <p>The (optional) value-directive assigns the possible return value of the procedure to variable MyVal.</p> <p>The (optional) param-directive assigns out-parameter values to Variables.</p> <p>The (optional) sender-directive retrieves the identifier of the sender and stores it in variable ASender.</p>

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
		 <p>The diagram shows two lifelines. The left lifeline is a solid line with a thick black bar at the bottom. The right lifeline is a dashed line with a thick black bar at the bottom. A message arrow labeled 'getreply' points from the right lifeline to the left. Below the arrow, the text 'MyProc' and 'MyTemplateRef' are written.</p>	<p>Receive a response from a previous call. The reply has to match the conditions defined by signature reference and the template reference.</p> <p>Optional from-, value-, param- and sender-directives may be present to identify the sender of the reply, to retrieve the return value of the procedure, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</p>
		 <p>The diagram shows two lifelines. The left lifeline is a solid line with a thick black bar at the bottom. The right lifeline is a dashed line with a thick black bar at the bottom. A message arrow labeled 'getreply' points from the right lifeline to the left. Below the arrow, the text 'MyProc' and '{...}' are written.</p>	<p>Receive a response from a previous call. The reply has to match the conditions defined by signature reference and the inline template definition.</p> <p>Optional from-, value-, param- and sender-directives may be present to identify the sender of the reply, to retrieve the return value of the procedure, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</p>
		 <p>The diagram shows two lifelines. The left lifeline is a solid line with a thick black bar at the bottom. The right lifeline is a dashed line with a thick black bar at the bottom. A message arrow labeled 'getreply' points from the right lifeline to the left.</p>	<p>Accept any response from any previous call.</p> <p>Optional from- and sender-directives may be present to identify the sender of the reply or to retrieve the identifier of the peer entity.</p>
		 <p>The diagram shows a single lifeline on the left, which is a solid line with a thick black bar at the bottom. A message arrow labeled 'getreply' points from a small circle on the right to the lifeline.</p>	<p>Accept any response from any previous call at any port.</p> <p>The reply to be received from any port may be restricted by means referring to templates or by using inline templates.</p> <p>Optional from-, value-, param- and sender-directives may be present to identify the sender of the reply, to retrieve the return value of the procedure, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</p>

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
Catch exception from a previous blocking call	<code>catch</code>	 <pre> sequenceDiagram participant A participant B A->>B: call ... activate B B-->>A: catch MyProc B-->>A: MyTemplateRef B-->>A: from MyPeer B-->>A: -> value MyVal B-->>A: sender ASender B-->>A: getreply ... deactivate B </pre>	<p>Catch an exception from a previous call. The exception has to match the conditions defined by the template reference.</p> <p>NOTE – The type information is part of the template definition.</p> <p>The (optional) from-directive denotes that the sender of the exception shall be identified by variable <code>MyPeer</code>.</p> <p>The (optional) value-directive assigns the value of the exception to variable <code>MyVal</code>.</p> <p>The (optional) sender-directive retrieves the identifier of the sender and stores it in variable <code>ASender</code>.</p>
		 <pre> sequenceDiagram participant A participant B A->>B: call ... activate B B-->>A: catch MyProc B-->>A: ExceptionType B-->>A: MyTemplateRef B-->>A: getreply ... deactivate B </pre>	<p>Catch an exception from a previous call. The exception has to match the conditions defined by the exception type and the template reference.</p> <p>Optional from-, value-, and sender-directives may be present to identify the sender of the exception, to retrieve the exception value or to retrieve the identifier of the peer entity.</p>
		 <pre> sequenceDiagram participant A participant B A->>B: call ... activate B B-->>A: catch MyProc B-->>A: ExceptionType B-->>A: { ... } B-->>A: getreply ... deactivate B </pre>	<p>Catch an exception from a previous call. The exception has to match the conditions defined by the exception type and the inline template definition.</p> <p>Optional from-, value-, and sender-directives may be present to identify the sender of the exception, to retrieve the exception value or to retrieve the identifier of the peer entity.</p>

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
		 <p>The diagram shows two lifelines. The left lifeline has a solid activation bar. The right lifeline has a dashed activation bar. A message arrow labeled 'call' goes from the left lifeline to the right. A return arrow labeled 'call ...' goes from the right lifeline to the left. A message arrow labeled 'catch' goes from the right lifeline to the left. A message arrow labeled 'getreply ...' goes from the left lifeline to the right. The lifelines end with thick black bars.</p>	<p>Accept any exception from a blocking call.</p> <p>Optional from-, value- and sender-directives may be present to identify the sender of the exception, to retrieve the exception value (and assign it to a variable of type anytype) or to retrieve the identifier of the peer entity.</p>
<p>Catch exception from a previous non-blocking call or independent from a call</p>	<p>catch</p>	 <p>The diagram shows two lifelines. A message arrow labeled 'catch MyProc' goes from the right lifeline to the left. The message contains the text: 'MyTemplateRef from MyPeer -> value MyVal sender ASender'. The lifelines end with thick black bars.</p>	<p>Catch an exception from a previous call. The exception has to match the conditions defined by the template reference.</p> <p>NOTE – The type information is part of the template definition.</p> <p>The (optional) from-directive denotes that the sender of the exception shall be identified by variable MyPeer.</p> <p>The (optional) value-directive assigns the value of the exception to variable MyVal.</p> <p>The (optional) sender-directive retrieves the identifier of the sender and stores it in variable ASender.</p>
		 <p>The diagram shows two lifelines. A message arrow labeled 'catch MyProc ExceptionType' goes from the right lifeline to the left. The message contains the text: 'MyTemplateRef'. The lifelines end with thick black bars.</p>	<p>Catch an exception from a previous call. The exception has to match the conditions defined by the exception type and the template reference.</p> <p>Optional from-, value-, and sender-directives may be present to identify the sender of the exception, to retrieve the exception value or to retrieve the identifier of the peer entity.</p>
		 <p>The diagram shows two lifelines. A message arrow labeled 'catch MyProc ExceptionType {...}' goes from the right lifeline to the left. The message contains the text: '{...}'. The lifelines end with thick black bars.</p>	<p>Catch an exception from a previous call. The exception has to match the conditions defined by the exception type and the inline template definition.</p> <p>Optional from-, value-, and sender-directives may be present to identify the sender of the exception, to retrieve the exception value or to retrieve the identifier of the peer entity.</p>

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
		 <p>A diagram showing two lifelines. The left lifeline is a solid line with a thick black bar at the bottom. The right lifeline is a dashed line with a thick black bar at the bottom. An arrow labeled 'catch' points from the right lifeline to the left lifeline.</p>	<p>Catch any exception from any previous call.</p> <p>Optional from-, value- and sender-directives may be present to identify the sender of the exception, to retrieve the exception value (and assign it to a variable of type anytype) or to retrieve the identifier of the peer entity.</p>
		 <p>A diagram showing one lifeline: a solid line with a thick black bar at the bottom. An arrow labeled 'catch' points from a small circle (representing a port) to the lifeline.</p>	<p>Catch any exception from any previous call at any port.</p> <p>The exception to be received from any port may be restricted by means referring to templates or by using inline templates.</p> <p>Optional from-, value-, and sender-directives may be present to identify the sender of the exception, to retrieve the exception value or to retrieve the identifier of the peer entity.</p>
<p>Check (current) message/call received</p>	<p>check</p>	 <p>Four diagrams illustrating the 'check' message:</p> <ul style="list-style-type: none"> Diagram 1: Two lifelines (solid left, dashed right, both with thick black bars). Arrow labeled 'check MyTemplateRef' from right to left. Diagram 2: Two lifelines (solid left, dashed right, both with thick black bars). Arrow labeled 'check MyType MyTemplateRef' from right to left. Diagram 3: Two lifelines (solid left, dashed right, both with thick black bars). Arrow labeled 'check' from right to left. Diagram 4: One lifeline (solid line with thick black bar). Arrow labeled 'check MyTemplateRef' from a small circle to the lifeline. <p>Can be used also in combination with <code>getcall</code>, <code>getreply</code>, and <code>catch</code></p>	<p>with template, without type</p> <p>with template, with type</p> <p>without template, without type (any message from that port)</p> <p>with template, without type, without port (this message from that port)</p>

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
Check current message, call, reply or exception	check		<p>Check if a message with a value defined by a template reference has been received.</p> <p>The syntax follows the syntax for the reception of messages.</p> <p>NOTE – Check may also be used in combination with getcall, getreply and catch.</p>
			<p>Check if a message with a value defined by a template reference has been received.</p> <p>The syntax follows the syntax for the reception of messages.</p> <p>NOTE – Check may also be used in combination with getcall, getreply and catch.</p>
			<p>Check if a message with a value defined by an inline template definition has been received.</p> <p>The syntax follows the syntax for the reception of messages.</p> <p>NOTE – Check may also be used in combination with getcall, getreply and catch.</p>
			<p>Check if any message (no value and no type is specified) has been received.</p> <p>The syntax follows the syntax for the reception of messages.</p> <p>NOTE – Check may also be used in combination with getcall, getreply and catch.</p>
			<p>Check if any message (no value and no type is specified) has been received at any port.</p> <p>The syntax follows the syntax for the reception of messages.</p> <p>NOTE – Check may also be used in combination with getcall, getreply and catch.</p>
Clear port	clear		<p>The clear port statement is put into a condition symbol. The condition shall cover the instance of the port to be cleared only.</p>

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
Clear and give access to port	start		The start port statement is put into a condition symbol. The condition shall cover the instance of the port to be started only.
Stop access (receiving & sending) at port	stop		The stop statement is put into a condition symbol. The condition shall cover the instance of the port to be stopped only.
Timer operations			
Start timer	start		
Stop timer	stop		
Read elapsed time	read		No special GFT symbol, used within statements or expressions.
Check if timer running	running		No special GFT symbol, used within statements or expressions.
Timeout operation	timeout		
Set local verdict	verdict.set		The verdict is put into a condition symbol.
Get local verdict	verdict.get		No special GFT symbol, used within statements or expressions.

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
SUT operations			
Remote action to be done by the SUT	sut.action		The action statement is put into an action box.
Execution of test cases			
Execute test case	execute		The execute statement is put into a testcase execution symbol.
Attributes			
Definition of attributes for control, testcases, teststeps and functions	with		The with statement is put into a text symbol.
Comments			
Comments within text		<pre>/* My several lines comment */ // My single line comment</pre>	Can be used wherever text can be placed.
Comments for instance events			Shall be attached to events on a control, test component or port instance
Comments control, test case, function or test step diagrams			Shall be attached to events on a control, test component or port instance

Annex C

Examples

C.1 The Restaurant example

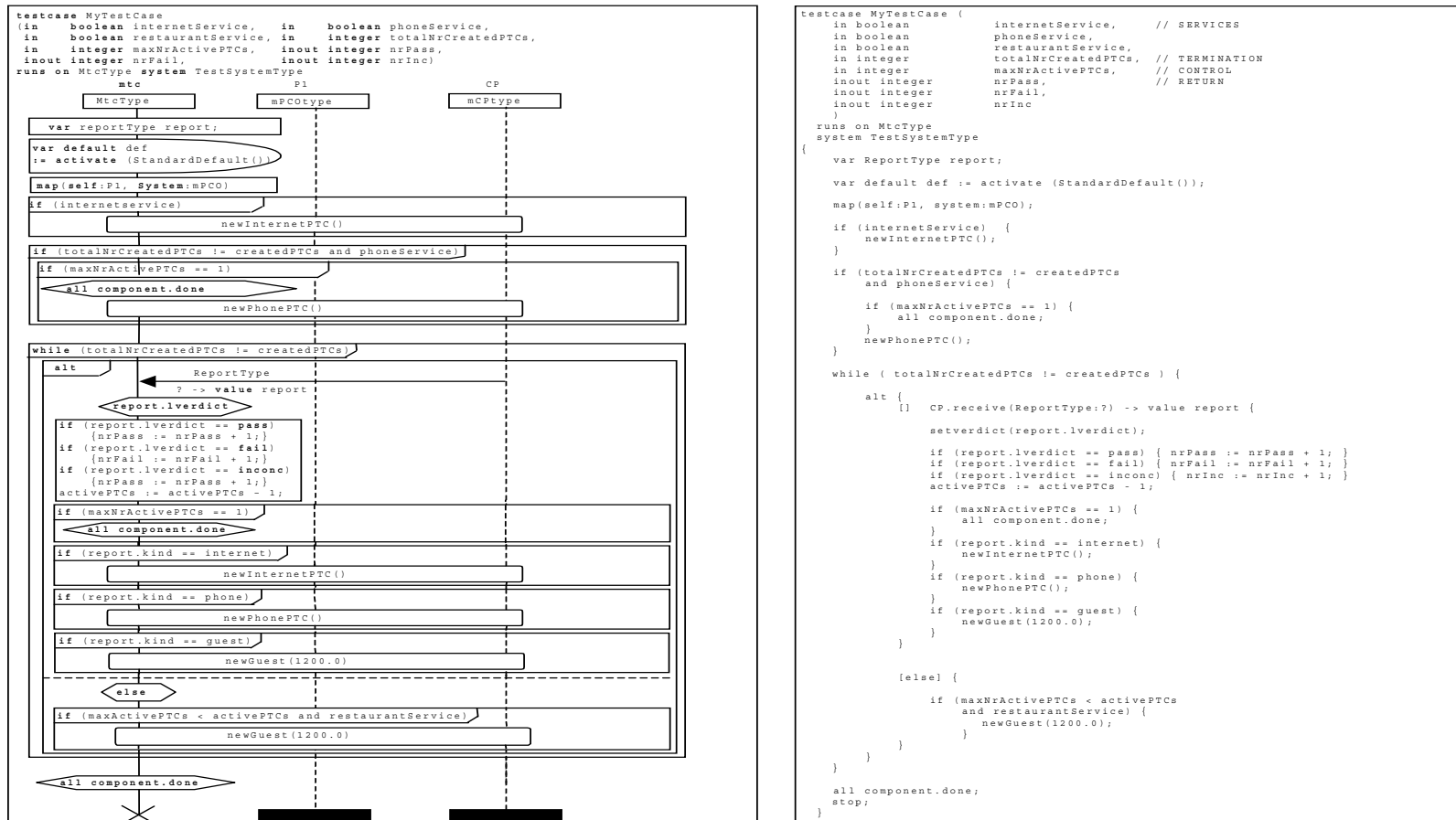
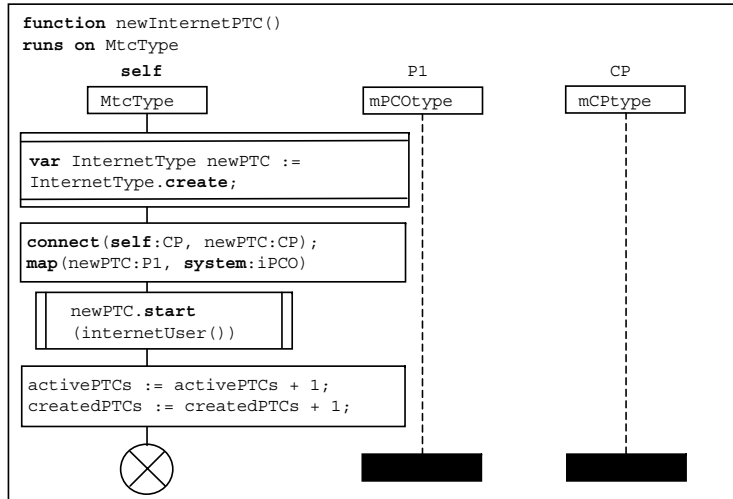


Figure C.1/Z.142 – Restaurant example – MyTestCase test case



```

function newInternetPTC ()
runs on MtcType {

var InternetType newPTC := InternetType.create;

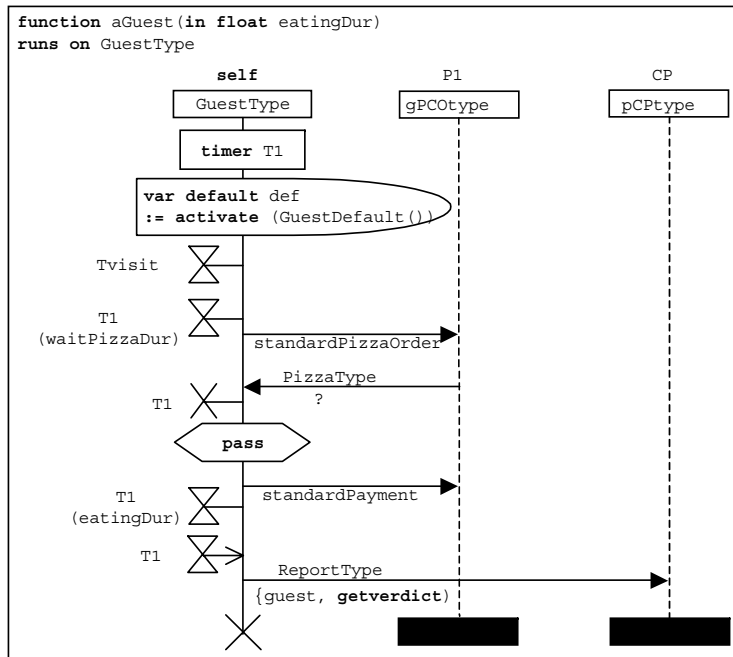
connect(self:CP, newPTC:CP);
map(newPTC:P1, system:iPCO);

newPTC.start (internetUser());

activePTCs := activePTCs + 1;
createdPTCs := createdPTCs + 1;

return;
}

```



```

function aGuest (in float eatingDur) runs on GuestType {

timer T1;

var default def := activate(GuestDefault());
Tvisit.start; // component timer
T1.start (waitPizzaDur);
P1.send(standardPizzaOrder);
P1.receive (PizzaType : ?);
T1.stop;
setverdict (pass);
P1.send(standardPayment);
T1.start (eatingDur); // eating
T1.timeout;
CP.send(ReportType : {guest, getverdict});
stop;
} // end function aGuest

```

Figure C.2/Z.142 – Restaurant example – newInternetPTC and aGuest functions

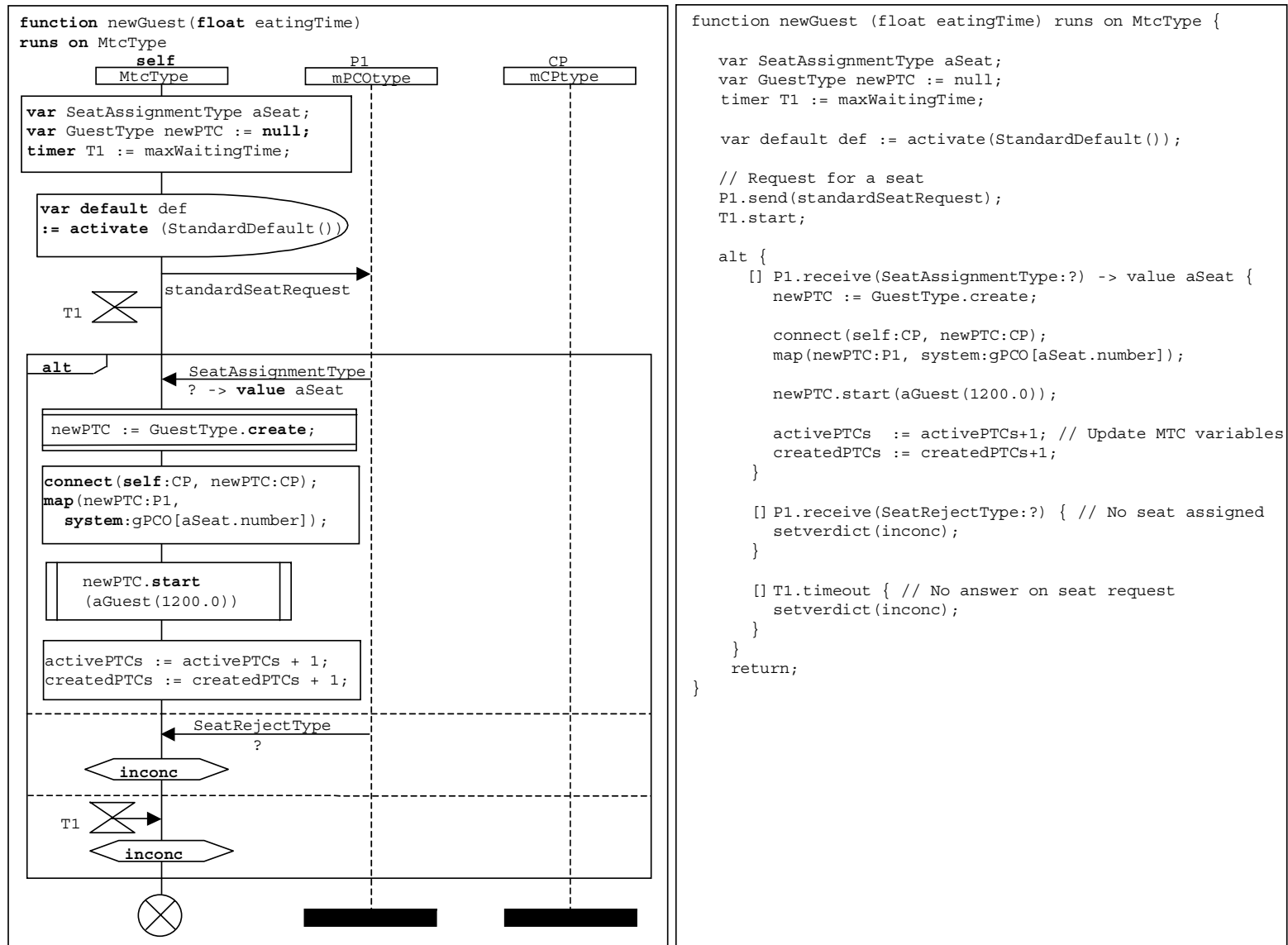


Figure C.3/Z.142 – Restaurant example – newGuest function

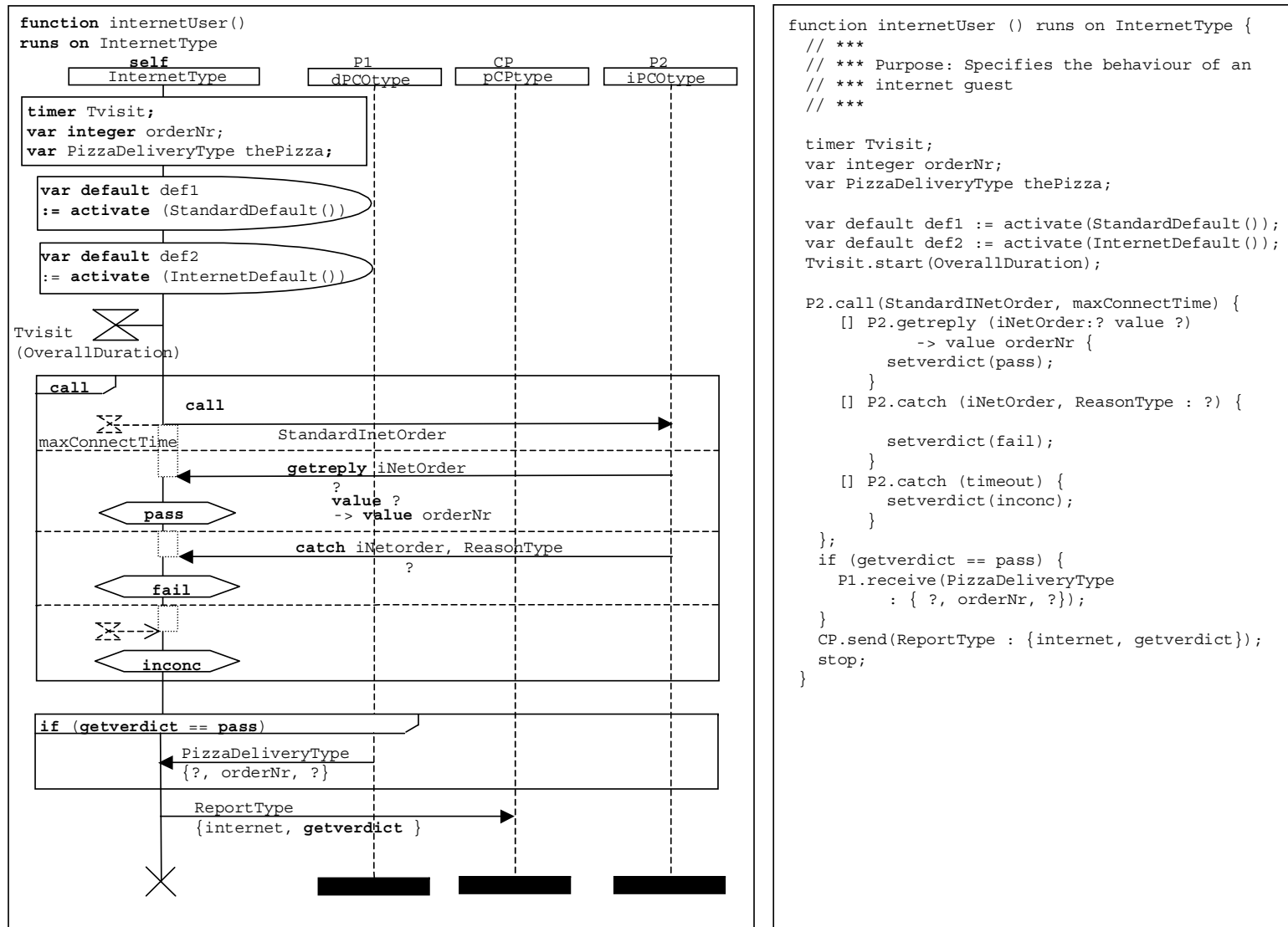
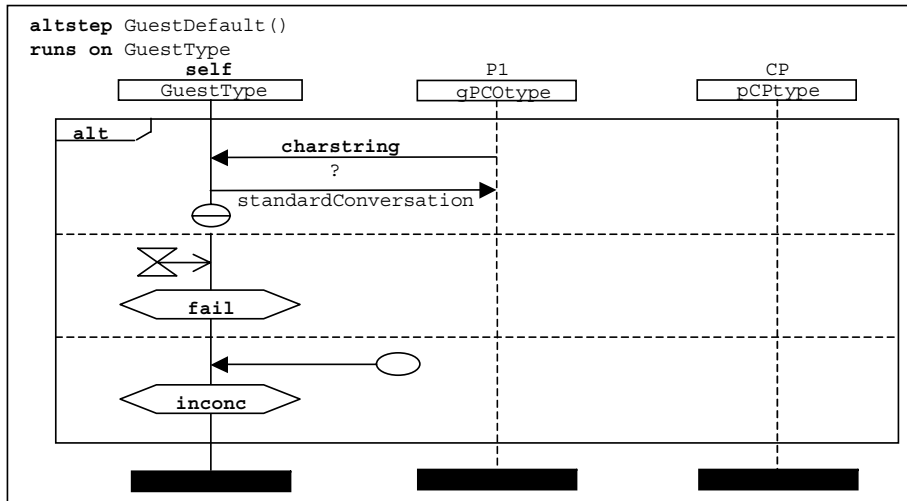


Figure C.4/Z.142 – Restaurant example – internetUser function



```

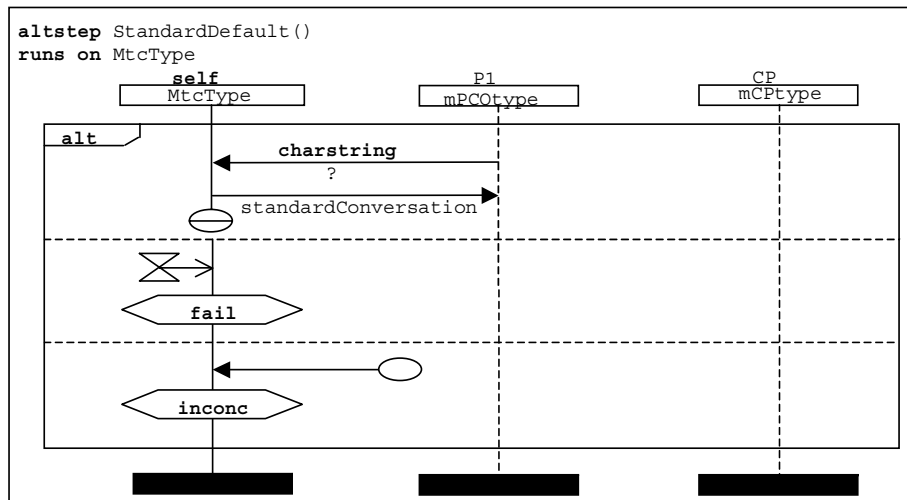
altstep GuestDefault() runs on GuestType {
// ***
// *** Purpose: Default behaviour for
// *** message based ports
// ***

[] P1.receive(charstring : ?) {
P1.send(standardConversation);
repeat;
}

[] any timer.timeout {
setverdict(fail);
}

[] any port.receive {
setverdict(inconc);
}
}

```



```

altstep StandardDefault() runs on MtcType {
// ***
// *** Purpose: Default behaviour for
// *** message based ports
// ***

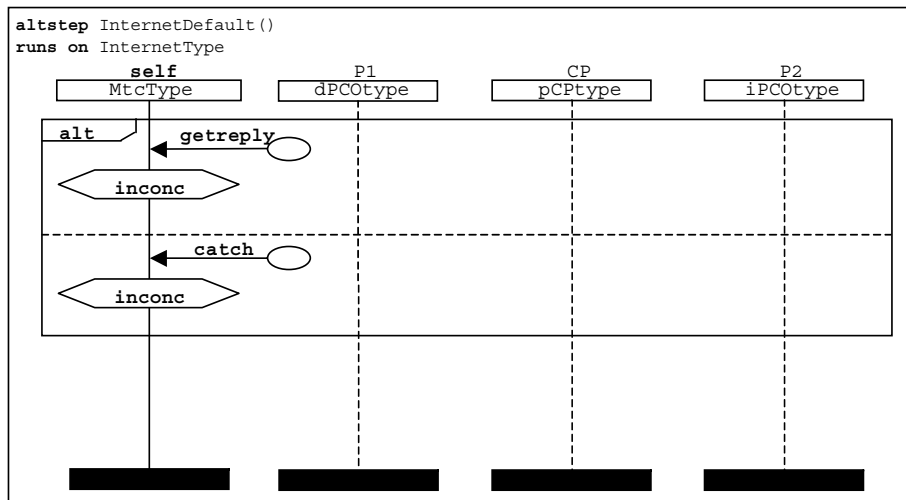
[] P1.receive(charstring : ?) {
P1.send(standardConversation);
repeat;
}

[] any timer.timeout {
setverdict(fail);
}

[] any port.receive {
setverdict(inconc);
}
}

```

Figure C.5/Z.142 – Restaurant example – GuestDefault and StandardDefault functions



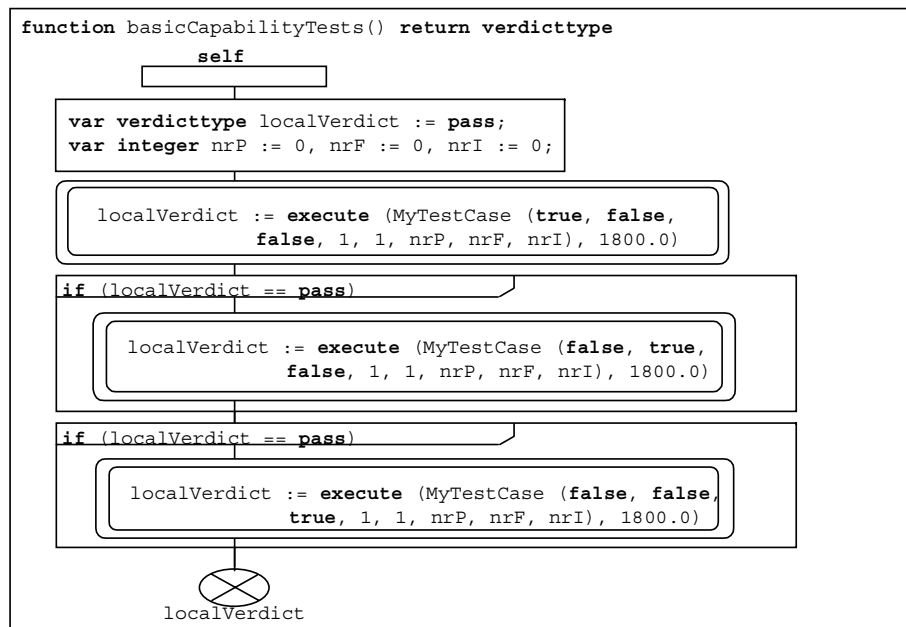
```

altstep InternetDefault()
runs on InternetType {
// ***
// *** Purpose: Default behaviour for
// **** the procedure based port
// ***

[] any port.getreply {
setverdict (inconc);
}

[] any port.catch {
setverdict (inconc);
}
}

```



```

function basicCapabilityTests ()
return verdicttype {
var verdicttype localVerdict := pass;
var integer nrP := 0, nrF := 0, nrI := 0;

// *** INTERNET ORDER ***
localVerdict := execute(MyTestCase (true,false,
false,1,1,nrP,nrF,nrI) ,1800.0);

// *** PHONE ORDER
if (localVerdict == pass) {
localVerdict := execute(MyTestCase
(false,true,false,1,1,nrP,nrF,nrI) ,1800.0);
}

// *** RESTAURANT ORDER ***
if (localVerdict == pass) {
localVerdict := execute(MyTestCase
(false,false,true,1,1,nrP,nrF,nrI) ,1800.0);
}
return (localVerdict);
}

```

Figure C.6/Z.142 – Restaurant example – internetDefault altstep and basicCapabilityTests function

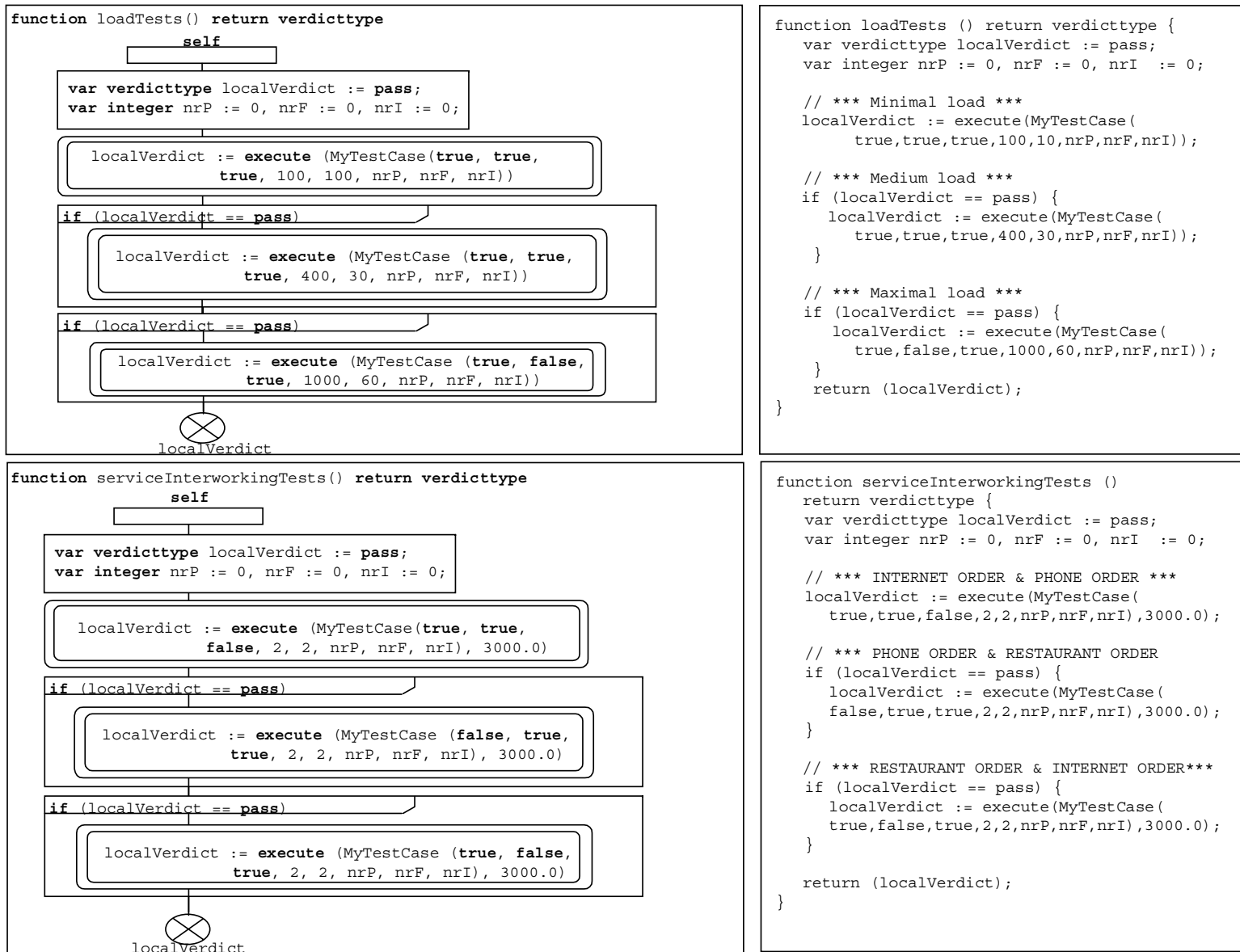


Figure C.7/Z.142 – Restaurant example – loadTests and serviceInterworkingTests functions

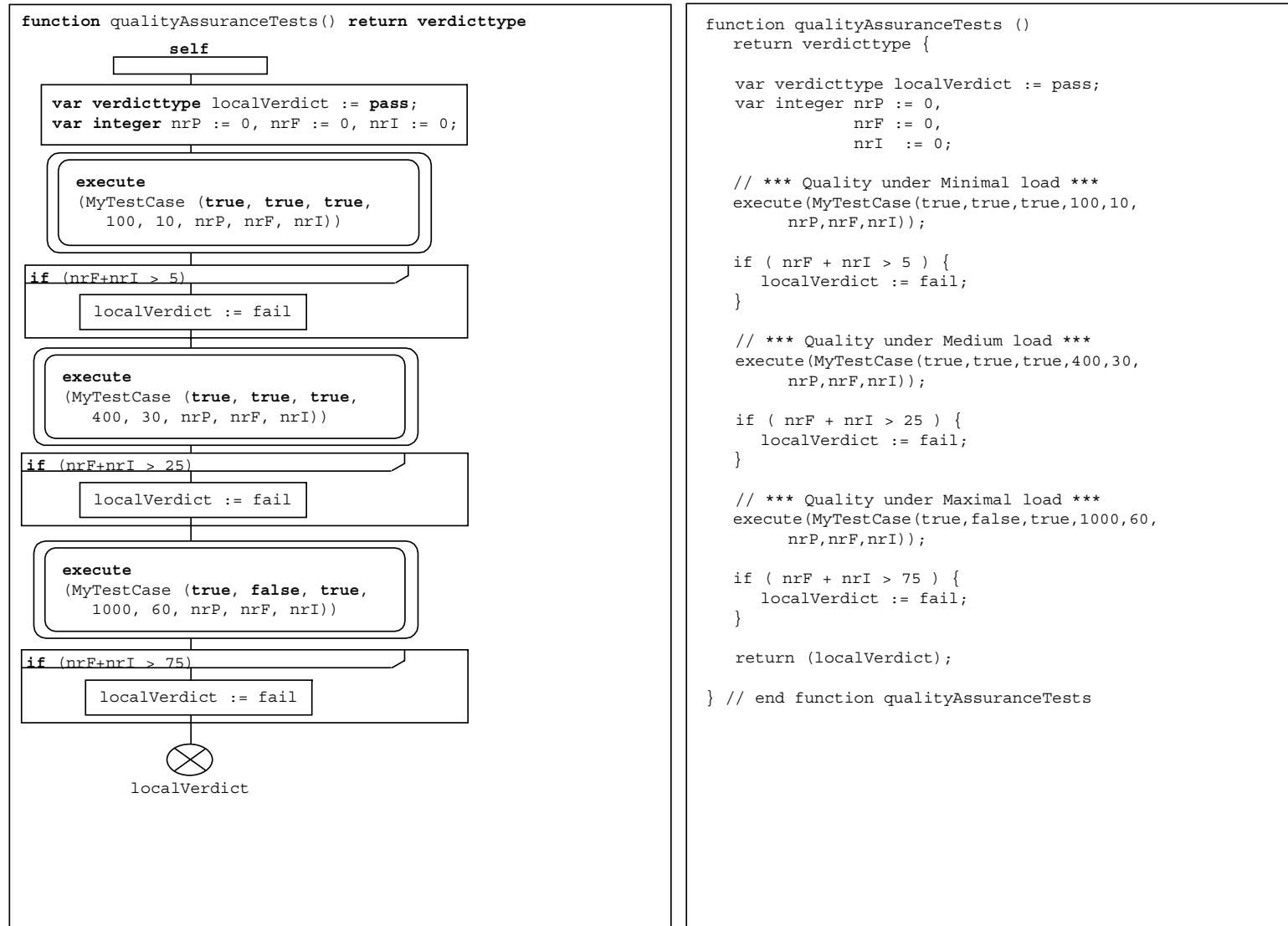
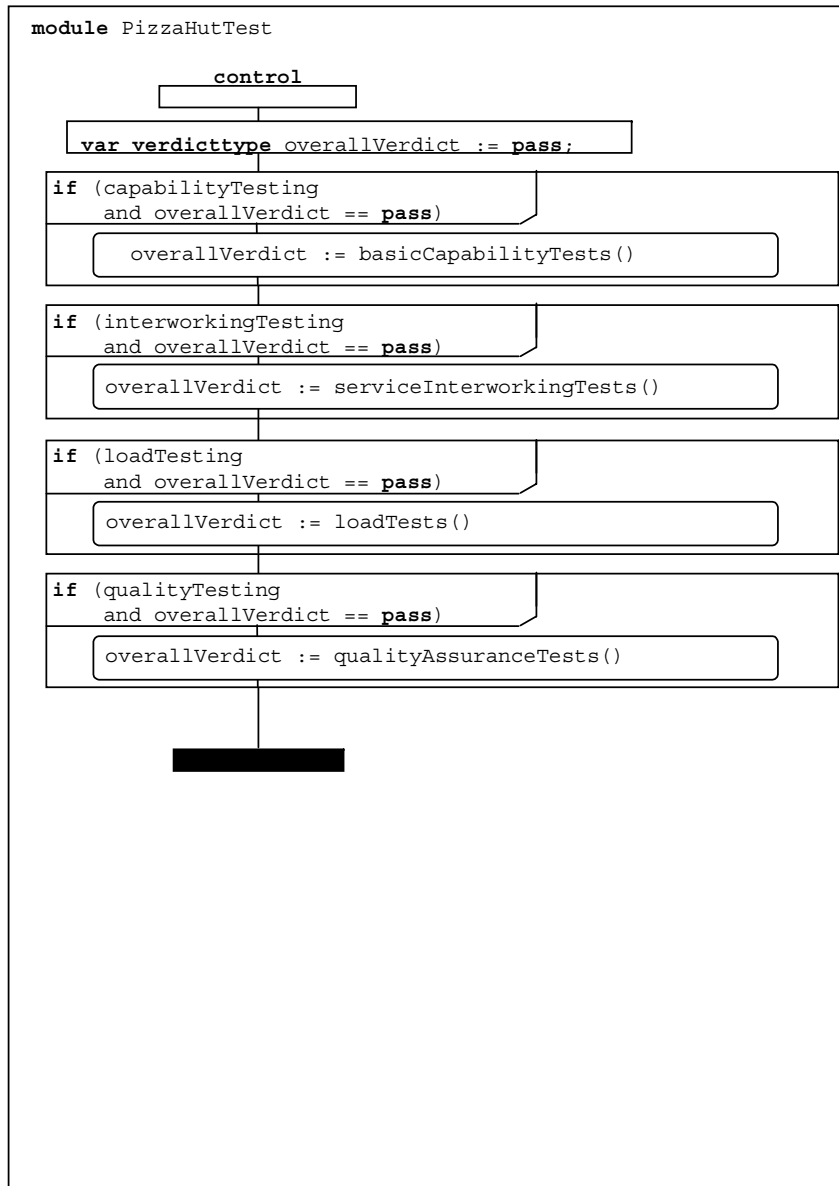


Figure C.8/Z.142 – Restaurant example – qualityAssuranceTests



```

module PizzaHutTest (
  boolean capabilityTesting,
  boolean interworkingTesting,
  boolean loadTesting,
  boolean qualityTesting ) {
  control {
    var verdicttype overallVerdict := pass;

    // Basic Capability Tests
    if (capabilityTesting and overallVerdict == pass) {
      overallVerdict := basicCapabilityTests();
    }

    // Interworking Tests
    if (interworkingTesting and overallVerdict == pass) {
      overallVerdict := serviceInterworkingTests();
    }

    // Load Tests
    if (loadTesting and overallVerdict == pass) {
      overallVerdict := loadTests();
    }

    // Quality Assurance Tests
    if (qualityTesting and overallVerdict == pass) {
      overallVerdict := qualityAssuranceTests();
    }
  }
}

```

Figure C.9/Z.142 – Restaurant example – PizzaHutTest module

C.2 The INRES example

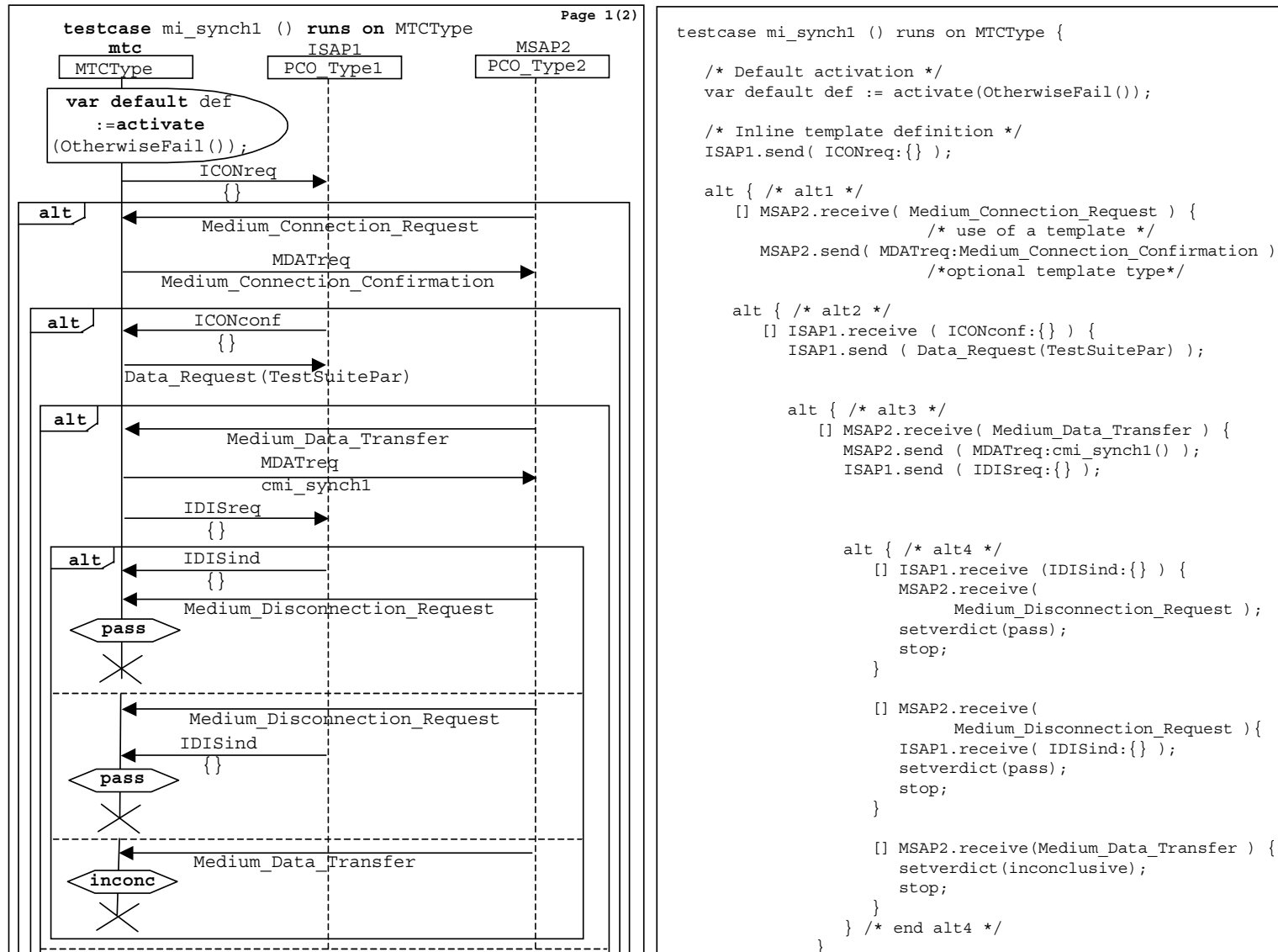


Figure C.10/Z.142 – INRES example – mi_synch1 1(2) test case

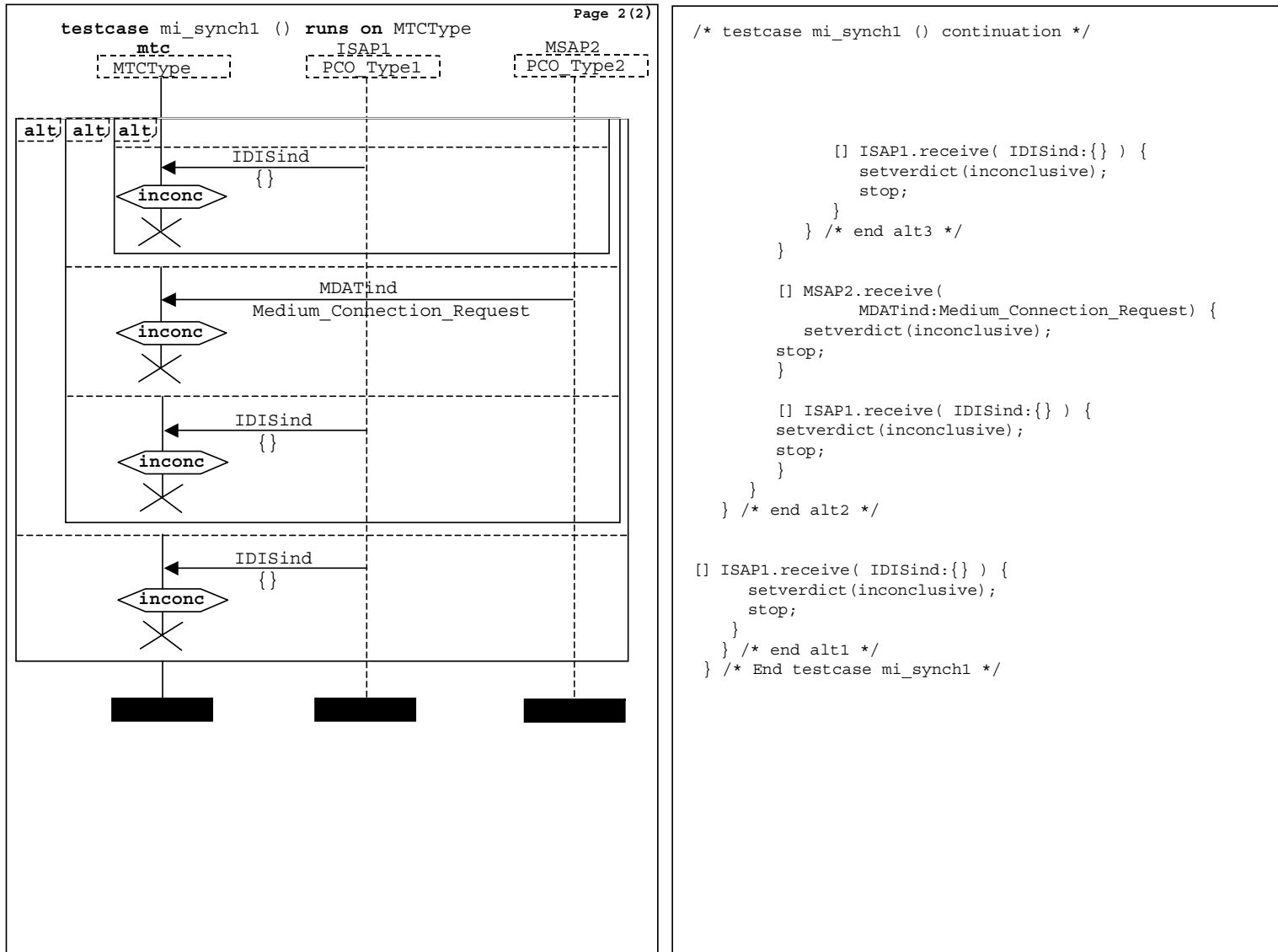
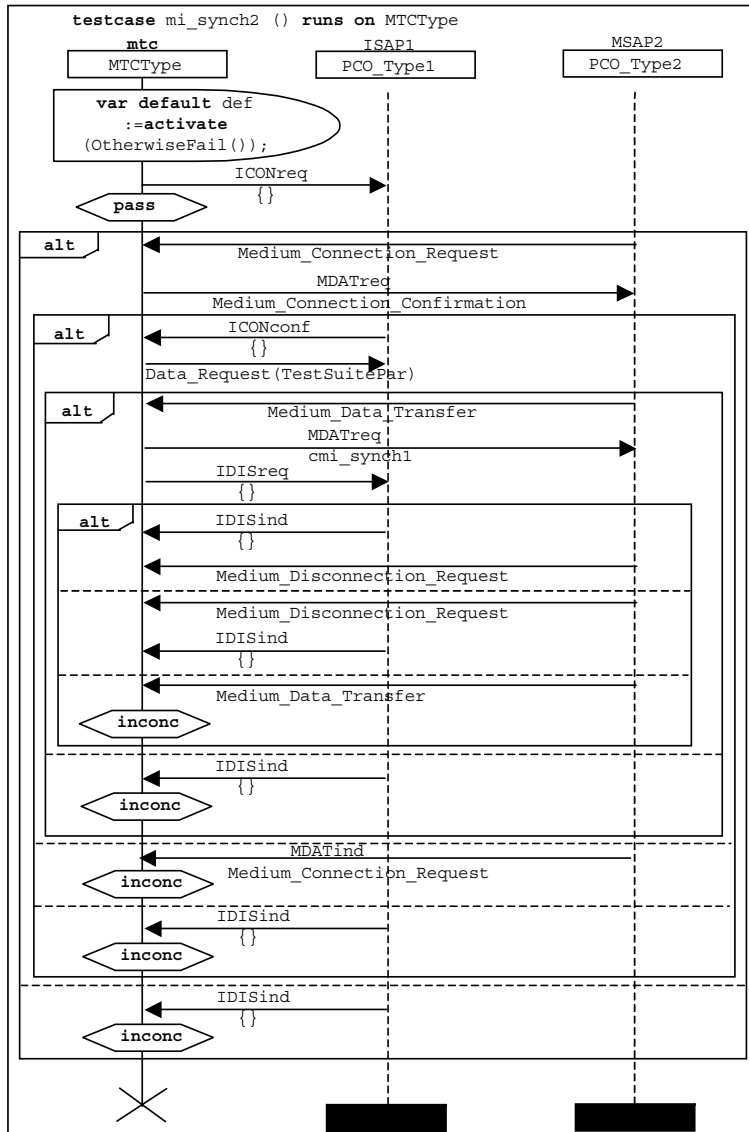


Figure C.11/Z.142 – INRES example – mi_synch1 2(2) test case



```

testcase mi_synch2 () runs on MTCType {

var default def := activate(OtherwiseFail());
/* Default activation */

ISAP1.send( ICONreq: {} );
setverdict( pass );

alt {
[] MSAP2.receive( Medium_Connection_Request ) {
MSAP2.send ( MDATreq:Medium_Connection_Confirmation );
alt {
[] ISAP1.receive ( ICONconf: {} ) {
ISAP1.send ( Data_Request(TestSuitePar) );
alt {
[] MSAP2.receive ( Medium_Data_Transfer ) {
MSAP2.send ( MDATreq:cmi_synch1 );
ISAP1.send ( IDISreq: {} );
alt {
[] ISAP1.receive ( IDISind: {} ) { /* PASS */
MSAP2.receive(
Medium_Disconnection_Request );
}
[] MSAP2.receive(
Medium_Disconnection_Request ){
ISAP1.receive( IDISind: {} ); /* PASS */
}
}
[] MSAP2.receive ( Medium_Data_Transfer ) {
setverdict( inconclusive);
}
}
}
}
[] ISAP1.receive( IDISind: {} ) {
setverdict( inconclusive);
}
}
}
}
[] MSAP2.receive( MDATind:Medium_Connection_Request ) {
setverdict( inconclusive);
}
}
[] ISAP1.receive( IDISind: {} ) {
setverdict( inconclusive);
}
}
}
}
stop; } /* End testcase mi_synch2 */

```

Figure C.12/Z.142 – INRES example – mi_synch2 test case

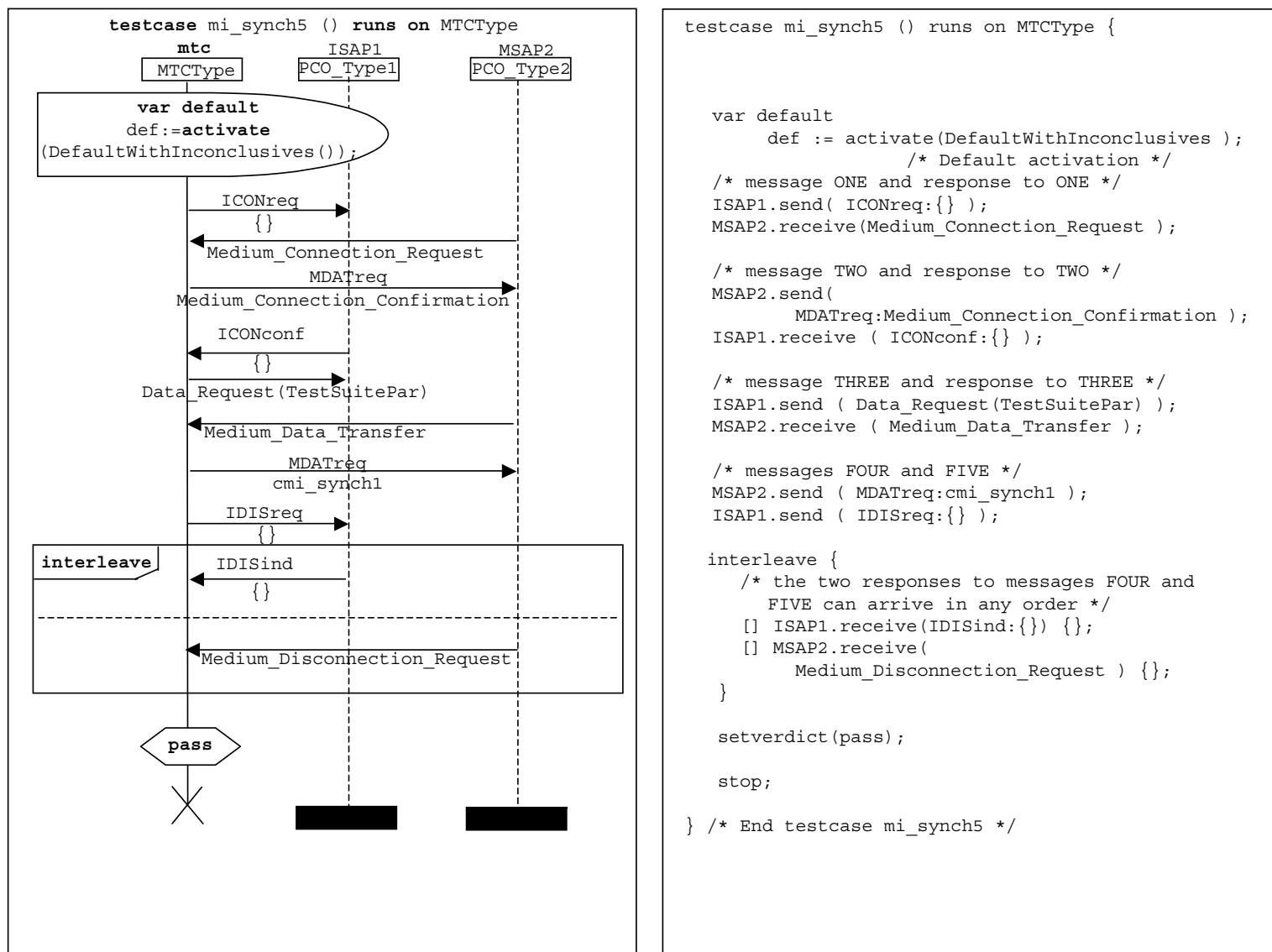


Figure C.13/Z.142 – INRES example – mi_synch5 test case

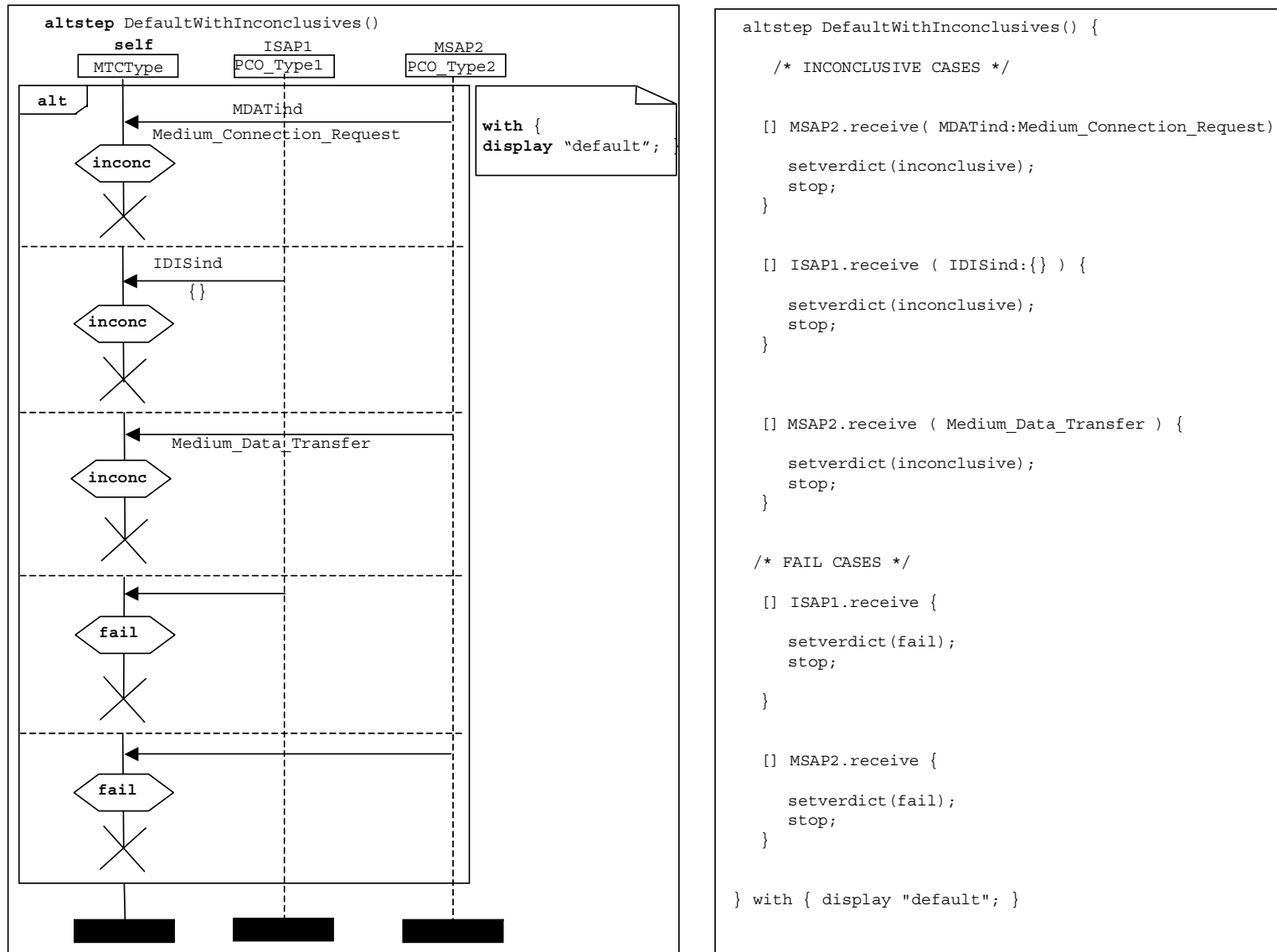


Figure C.14/Z.142 – INRES example – DefaultWithInconclusives altstep

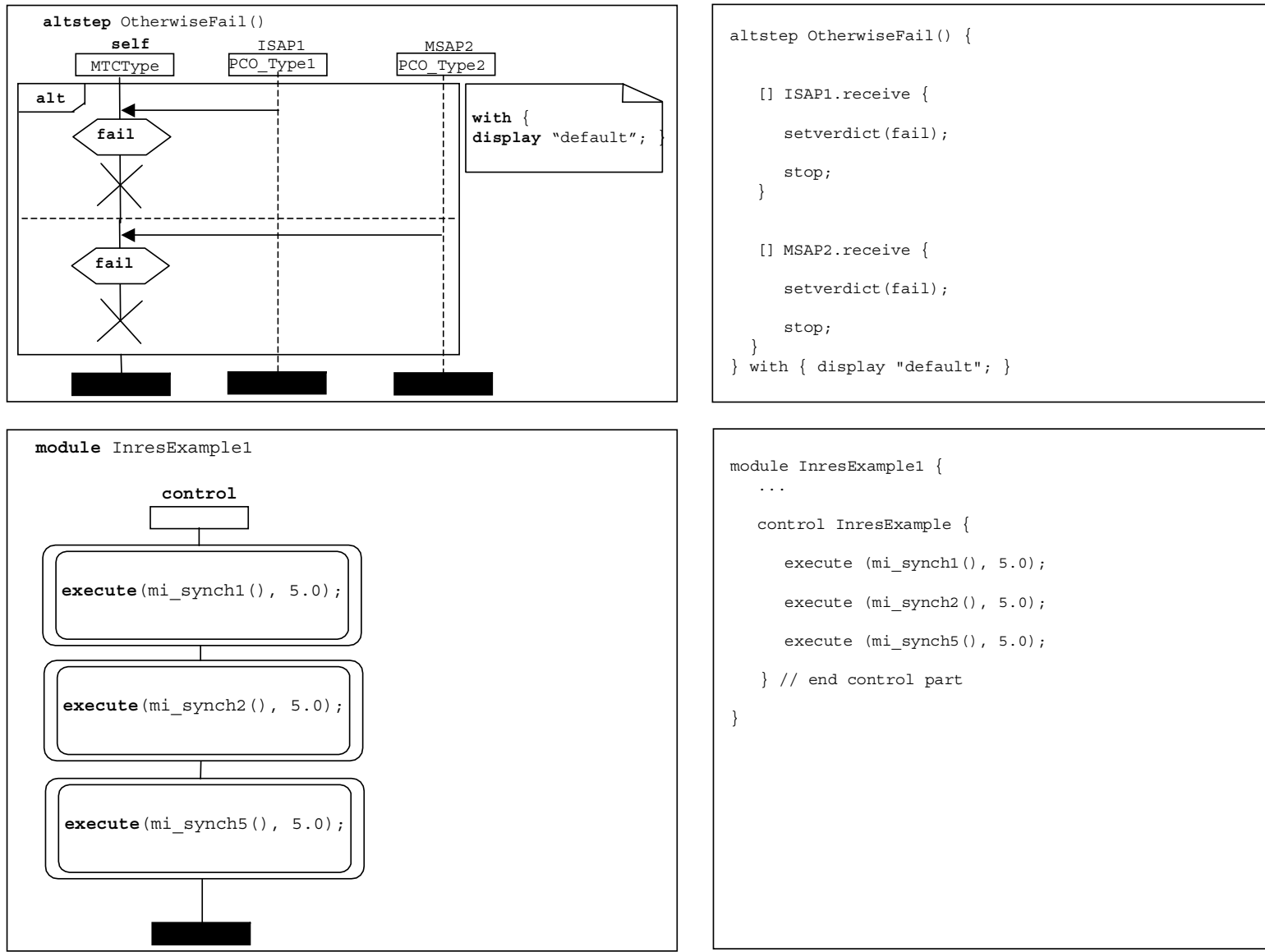


Figure C.15/Z.142 – INRES example – OtherwiseFail altstep and InresExample1 module definitions

SÉRIES DES RECOMMANDATIONS UIT-T

Série A	Organisation du travail de l'UIT-T
Série D	Principes généraux de tarification
Série E	Exploitation générale du réseau, service téléphonique, exploitation des services et facteurs humains
Série F	Services de télécommunication non téléphoniques
Série G	Systèmes et supports de transmission, systèmes et réseaux numériques
Série H	Systèmes audiovisuels et multimédias
Série I	Réseau numérique à intégration de services
Série J	Réseaux câblés et transmission des signaux radiophoniques, télévisuels et autres signaux multimédias
Série K	Protection contre les perturbations
Série L	Construction, installation et protection des câbles et autres éléments des installations extérieures
Série M	Gestion des télécommunications y compris le RGT et maintenance des réseaux
Série N	Maintenance: circuits internationaux de transmission radiophonique et télévisuelle
Série O	Spécifications des appareils de mesure
Série P	Qualité de transmission téléphonique, installations téléphoniques et réseaux locaux
Série Q	Commutation et signalisation
Série R	Transmission télégraphique
Série S	Equipements terminaux de télégraphie
Série T	Terminaux des services télématiques
Série U	Commutation télégraphique
Série V	Communications de données sur le réseau téléphonique
Série X	Réseaux de données, communication entre systèmes ouverts et sécurité
Série Y	Infrastructure mondiale de l'information, protocole Internet et réseaux de prochaine génération
Série Z	Langages et aspects généraux logiciels des systèmes de télécommunication