Union internationale des télécommunications

UIT-T

SECTEUR DE LA NORMALISATION
DES TÉLÉCOMMUNICATIONS
DE L'UIT

Z.142

(02/2003)

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

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

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

Recommandation UIT-T Z.142

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

Résumé

La présente Recommandation fournit un format de présentation graphique pour la spécification de test et le langage d'implémentation TTCN-3 (notation de test et de commande de test) appelé format GFT. Ce format est utilisé pour représenter graphiquement un comportement de test à l'aide de diagrammes de séquences. Il fournit 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.

Source

La Recommandation UIT-T Z.142 a été approuvée le 13 février 2003 par la Commission d'études 17 (2001-2004) 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 de renseignements les plus récents, il est vivement recommandé aux responsables de la mise en œuvre de consulter la base de données des brevets du TSB.

TABLE DES MATIÈRES

Introduction

Le format de présentation graphique de la notation TTCN-3 (GFT) est fondé sur la Recommandation UIT-T Z.120 [3] qui définit des diagrammes de séquences de messages (MSC). Le format GFT utilise un sous-ensemble de diagrammes MSC avec des extensions spécifiques 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 Recommandation UIT-T Z.140 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 (Figure 1).

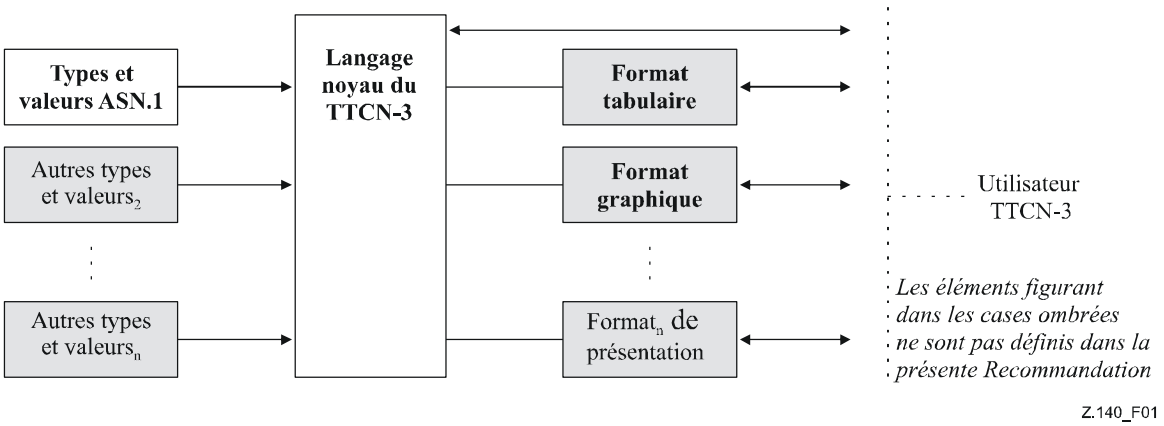


Figure 1/Z.142 – Vue d'utilisateur du langage noyau et des divers formats de présentation

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.

Recommandation UIT-T Z.142

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 Recommandation UIT-T Z.140. 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. 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 (2003), *Notation de test et de commande de test version 3 (TTCN-3): langage noyau.* Cette Recommandation est également disponible sous la forme d'une norme ETSI de cote ES 201 873-1 V2.2.1 (2003-02).

[2] Recommandation UIT-T Z.141 (2003), *Notation de test et de commande de test version 3 (TTCN-3): format de présentation tabulaire*.

[3] Recommandation UIT-T Z.120 (1999), *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)*.

3 Abréviations

La présente Recommandation utilise les abréviations suivantes:

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

CATG génération de tests informatisés (*computer-aided test generation*)

ETSI Institut européen des normes de télécommunication (*European Telecommunication Standards Institute*)

GFT format de présentation graphique de la notation TTCN-3 (*graphical presentation format of TTCN-3*)

HMSC diagramme de séquences de messages de haut niveau (*high-level message sequence chart*)

MSC diagramme de séquences de messages (*message sequence chart*)

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

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

SUT système sous test (*system under test*)

TFT format de présentation tabulaire de la notation TTCN-3 (*tabular presentation format of TTCN-3*)

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

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 composantes de test nécessaires pour valider le comportement du système sous test (il s'agit de la description d'un objectif de test).

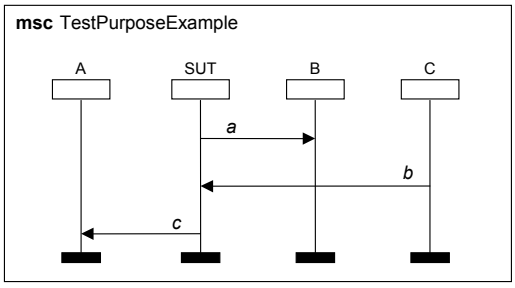


Figure 2/Z.142 – Diagramme MSC décrivant l'interaction entre un système sous test et ses interfaces

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 composantes 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, *main test component*) 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 répartie faisant intervenir plusieurs composantes de test.

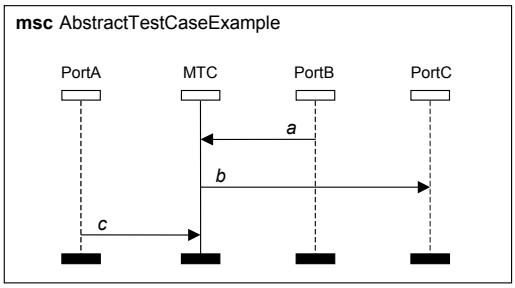


Figure 3/Z.142 – Diagramme MSC décrivant l'interface entre une composante MTC et les interfaces du système sous test

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) *Elaboration 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.*

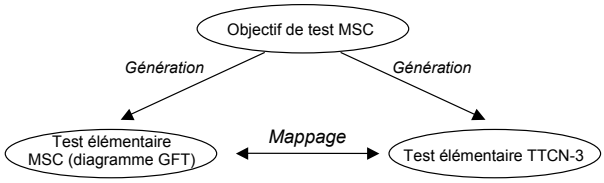


Figure 4/Z.142 – Relations entre la description d'un objectif de test de diagramme MSC, la description d'un test élémentaire de diagramme MSC et la notation TTCN-3

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 relatif 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) [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.

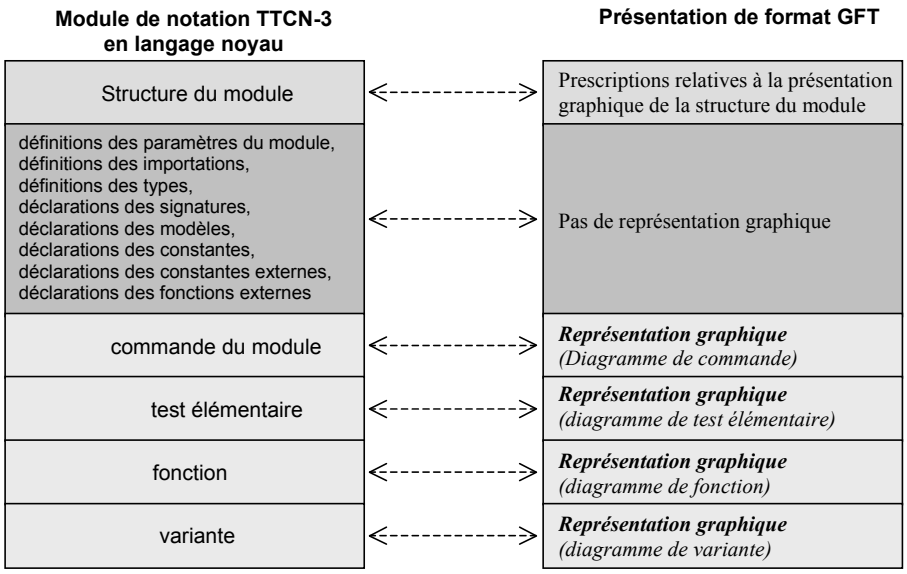


Figure 5/Z.142 – Relation entre le langage noyau TTCN-3 et la description de format GFT correspondante

Le format GFT étant fondé sur un diagramme MSC [3], donc 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.

De nouveaux symboles de format GFT ont été définis pour:

- 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 mappé 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.

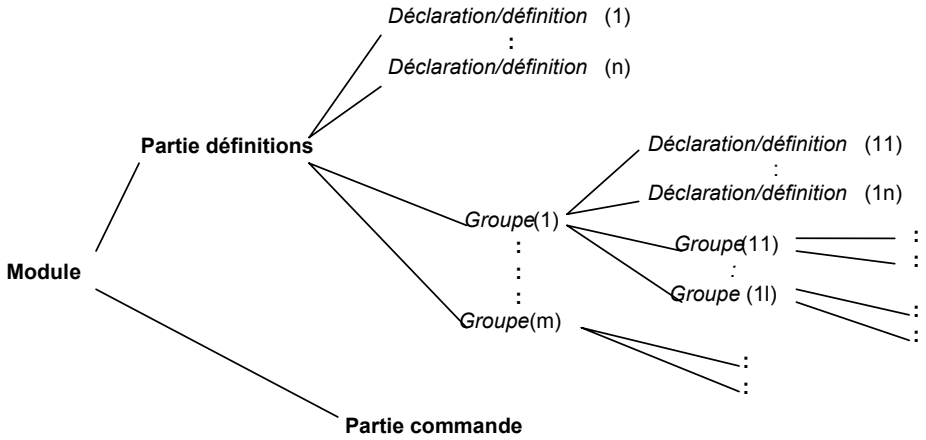


Figure 6/Z.142 – Structure d'un module de notation TTCN-3

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).

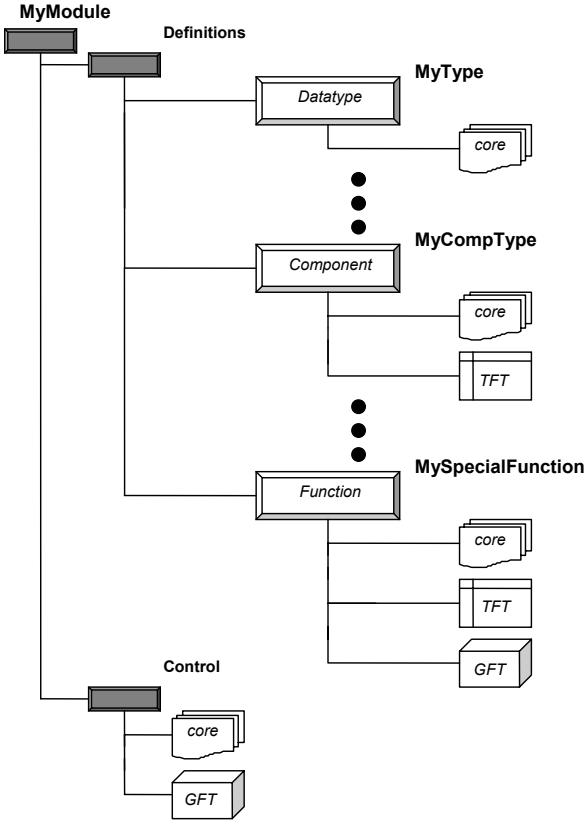


Figure 7/Z.142 – Divers formats de présentation d'une structure de module de notation TTCN-3 au sein d'une vue organisationnelle

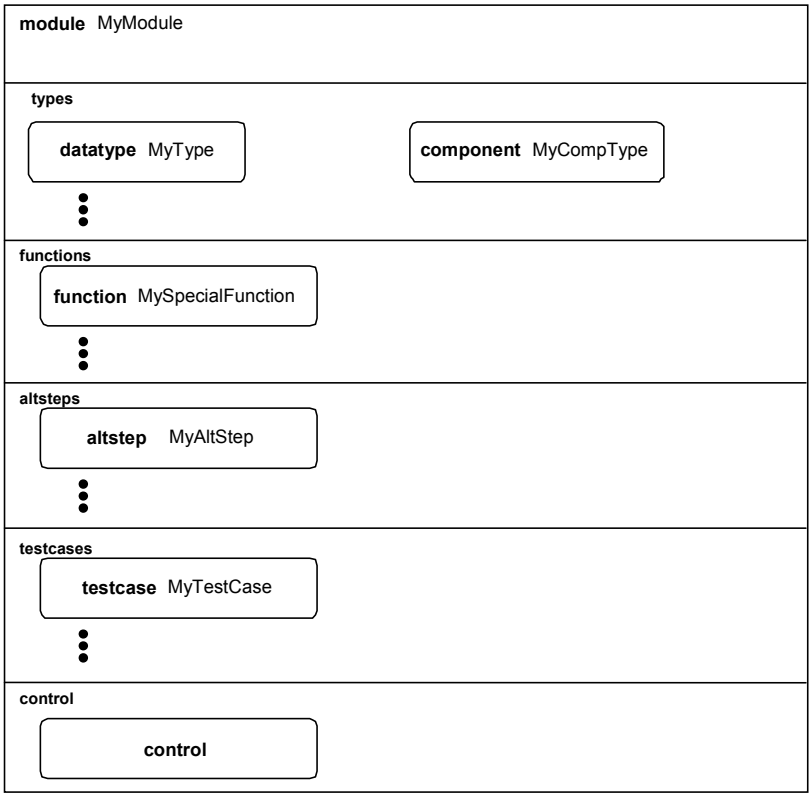
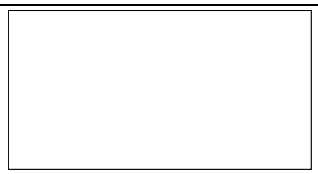
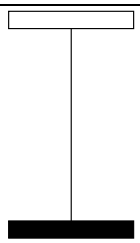
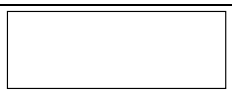
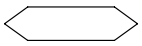
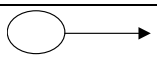
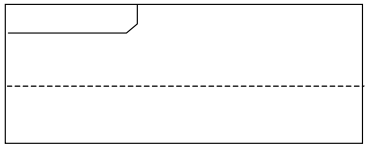
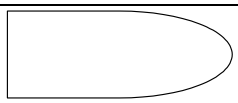
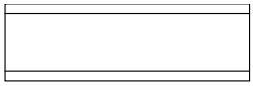


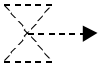
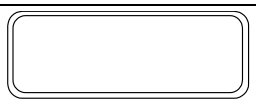
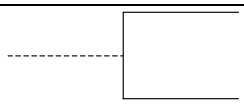
Figure 8/Z.142 – Présentation graphique de type document MSC d'une structure de module de notation

8 Symboles GFT

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

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

Elément GFT	Symbole	Description
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.
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.

Elément GFT	Symbole	Description
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 d'armement de temporisateur		utilisé pour l'opération d'armement de temporisateur TTCN-3, à rattacher à un symbole de composante.
Symbole de fin de temporisation		utilisé pour l'opération de fin de temporisation TTCN-3, à associer à un symbole de composante.
Symbole de désarmement de temporisateur		utilisé pour l'opération de désarmement de temporisateur TTCN-3, à rattacher à un symbole de composante.
Symbole d'armement implicite de temporisateur		utilisé pour l'armement implicite de temporisateur TTCN-3 en appel de blocage, à placer dans une expression en ligne d'appel et à rattacher à un symbole de composante.
Symbole de fin implicite de temporisation		utilisé pour l'exception de temporisation 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

Le format GFT fournit les types de diagramme suivant:

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

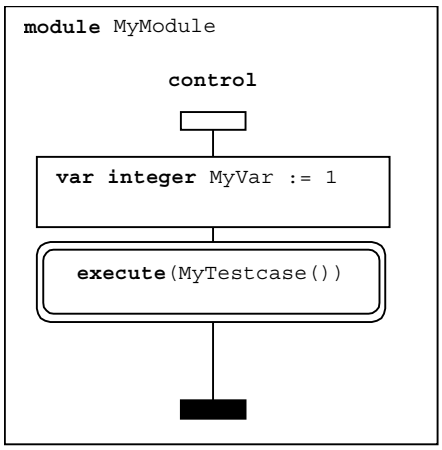
Les diagrammes GFT peuvent être structurés en pages, un grand diagramme GFT pouvant ainsi être réparti sur plusieurs pages. Chaque page d'un diagramme multipage doit avoir un numéro inscrit au coin supérieur droit qui l'identifie de manière univoque. Cette numérotation est facultative si le diagramme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



GFT

```
module MyModule {
     :
     :
     :
     control {
          var integer MyVar := 1;
          execute(MyTestcase());
          :
          :
          :
     } // end control
} // end module
```

Core

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.

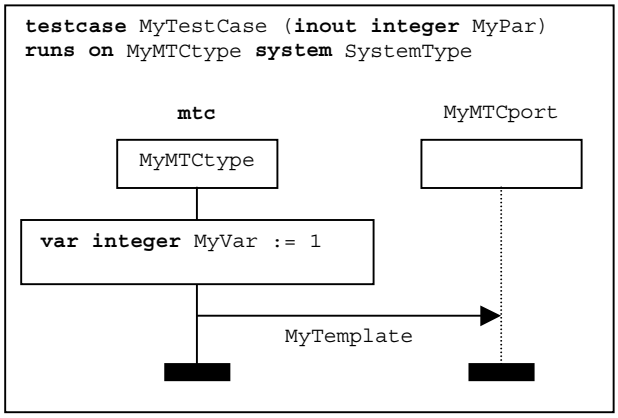
On se référera au § 11.4 pour la représentation en format GFT des expressions, des affectations, des instructions **log**, **label**, **goto**, **if-else**, boucle **for**, boucle **while**, boucle **do while** et **stop**, au § 11.9 pour les opérations de temporisation et aux § 9.4 et 11.2.2 pour les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



GFT

```
testcase MyTestCase (inout integer MyPar)
runs on MyMTCtype system SystemType {

    var integer MyVar := 1;

    MyMTCPort.send(MyTemplate);

        :
        :
}
```

Core

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.

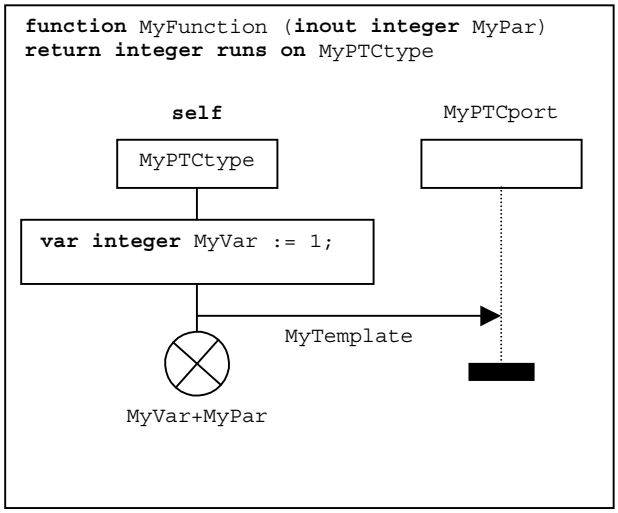
NOTE – Les noms et les types des ports utilisables par une fonction sont transmis sous forme de paramètres ou correspondent aux noms et types de port définis dans la définition de type de composante référencée dans la clause **runs on**.

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.

self et les noms 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.




```
function MyFunction (inout integer MyPar)
 return integer runs on MyPTCtype {

        var integer MyVar := 1;

        MyPTCport.send(MyTemplate);

        :
        :

        return MyVar+MyPar
}
```

GFT **Core**

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.

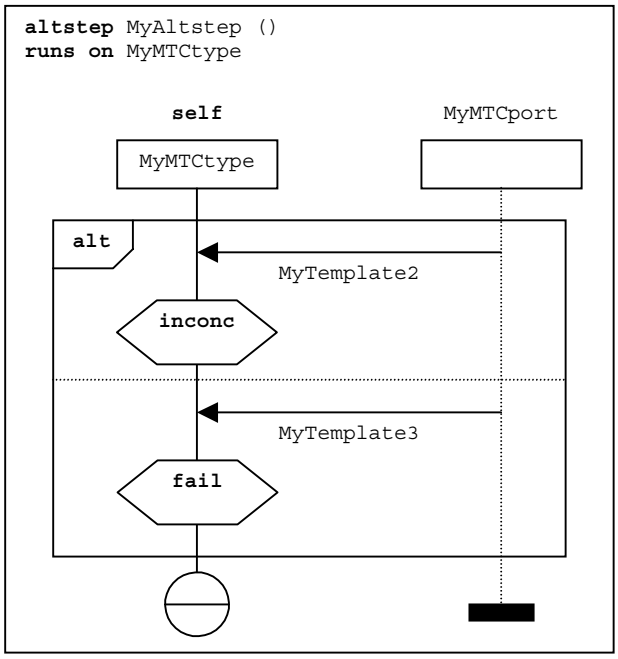
NOTE – Les noms et les types des ports utilisables par une variante sont transmis sous forme de paramètres ou correspondent aux noms et types de port définis dans la définition de type de composante référencée dans la clause **runs on**.

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.

self et les noms 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.




```
variante MyVariante () runs on MyMTCtype {

    alt {

        []  MyMTCport.receive(MyTemplate2) {
                setverdict(inconc)
            }
        []  MyMTCport.receive(MyTemplate3) {
                setverdict(fail)
            }
    }
        :
        :
    repeat
}
```

GFT Core

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:

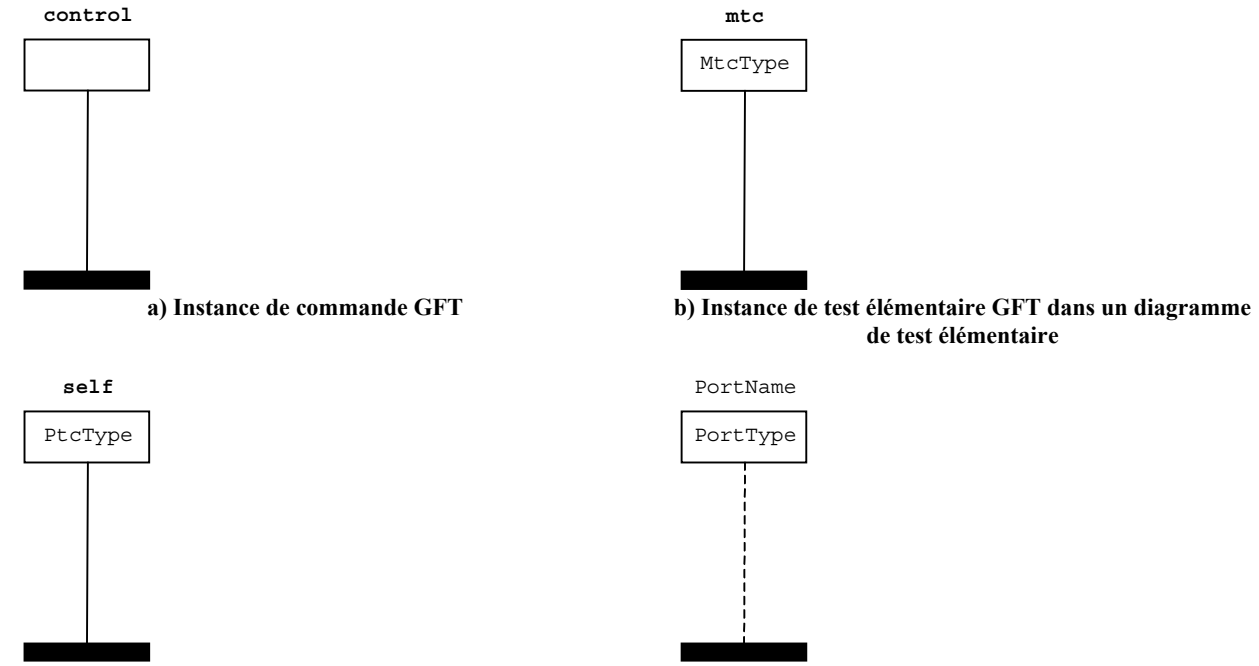
- le nom obligatoire **mtc** est placé au-dessus du symbole de début d'instance dans le cas d'un diagramme de test élémentaire;
- le nom obligatoire **self** est placé au-dessus du symbole de début d'instance dans le cas d'un diagramme de fonction ou de variante.

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).



a) Instance de commande GFT

b) Instance de test élémentaire GFT dans un diagramme de test élémentaire

c) Instance de composante de test GFT dans un diagramme de fonction ou de variante

d) Instance de port GFT

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

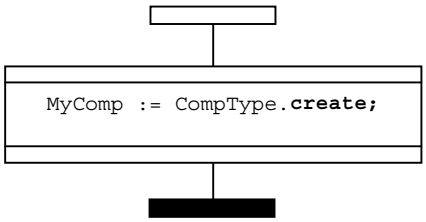
Les déclarations, instructions et opérations TTCN-3 suivantes sont spécifiées dans des symboles d'action: les déclarations (moyennant les restrictions définies au § 11.3) et les assignations **log**, **connect**, **disconnect**, **map**, **unmap** et **action**.

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.

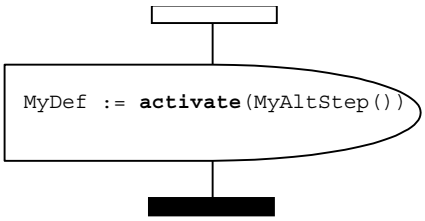
11.1.3 Commentaires

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

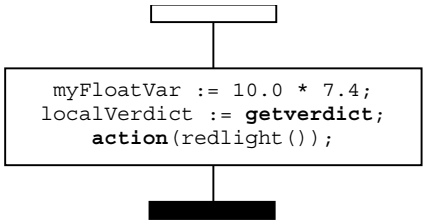
- les commentaires peuvent être placés dans des symboles GFT après l'inscription de symbole, en utilisant la syntaxe de commentaires du langage noyau TTCN-3 (Figure 14 d));
- les commentaires conformes à la syntaxe du langage noyau TTCN-3 peuvent être inscrits dans des symboles de texte librement placés 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)).



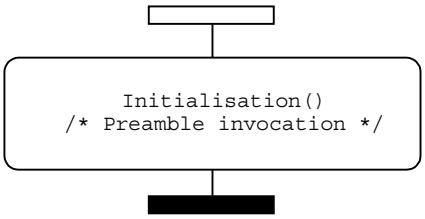
a) Création de composante avec un point-virgule de terminaison facultatif



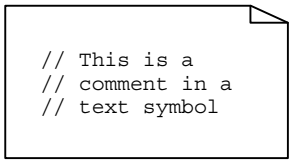
b) Activation par défaut sans point-virgule de terminaison



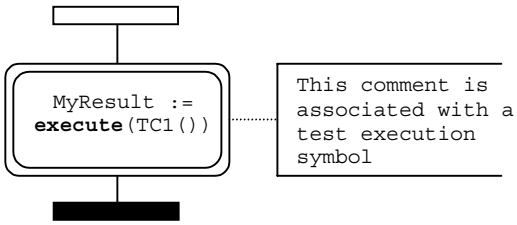
c) Séquence d'instructions dans un symbole d'action



d) Commentaires dans un symbole de référence GFT



e) Commentaires dans un symbole de texte



f) Commentaires dans un symbole de commentaires associé à un symbole d'exécution

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**.

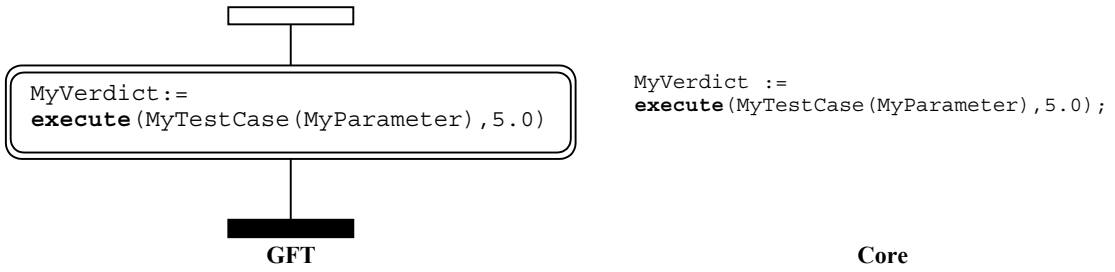


Figure 15/Z.142 – Exécution d'un test élémentaire

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).

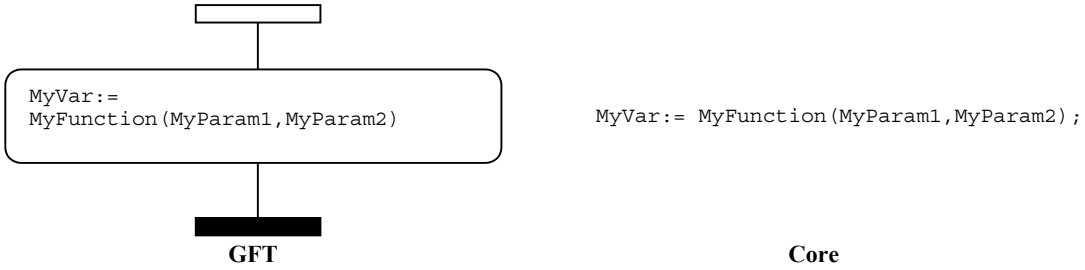


Figure 16/Z.142 – Invocation d'une fonction définie par l'utilisateur

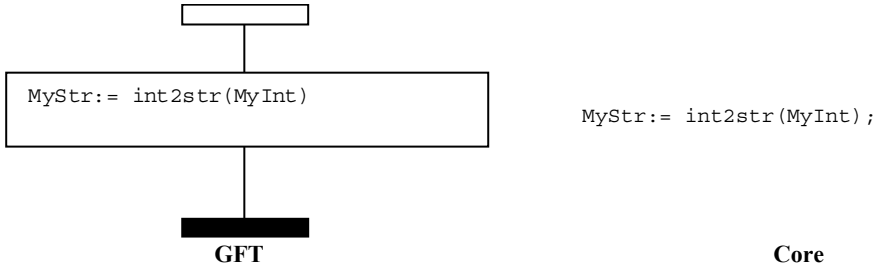


Figure 17/Z.142 – Invocation d'une fonction prédéfinie/externe

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

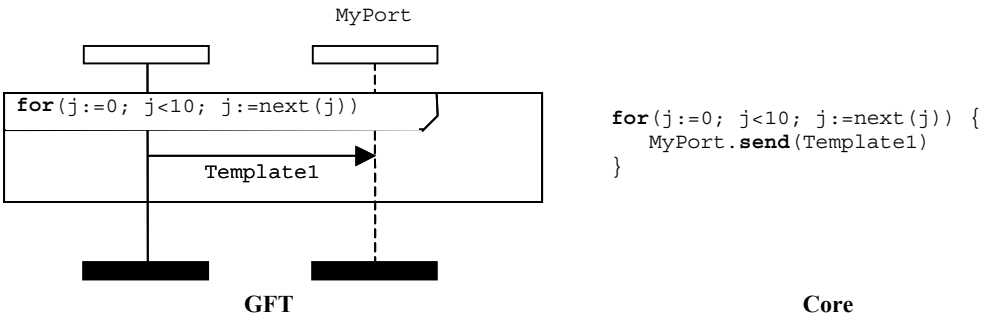


Figure 18/Z.142 – Invocation d'une fonction définie par l'utilisateur dans un symbole GFT

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).

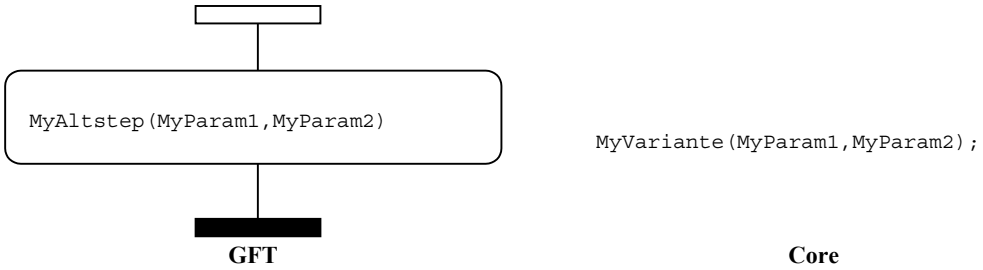


Figure 19/Z.142 – Invocation d'une variante

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).

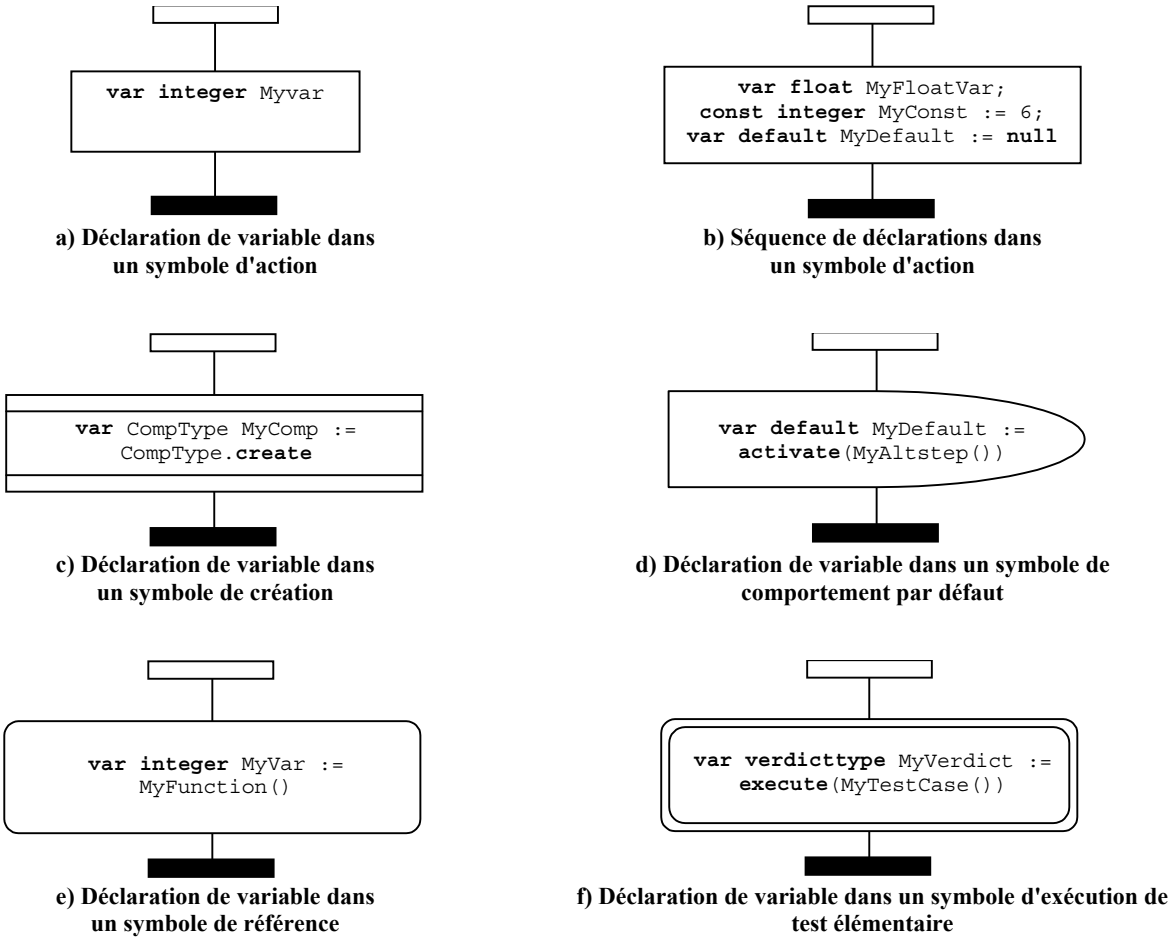


Figure 20/Z.142 – Exemples de déclarations en format GFT

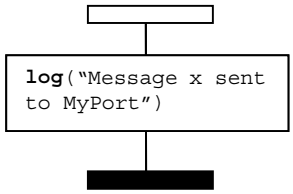
11.4 Instructions de programme de base

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).



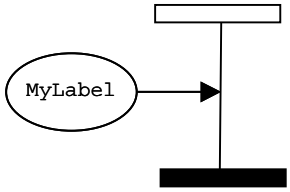

```
log("Message x sent to MyPort");
```

GFT **Core**

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`.



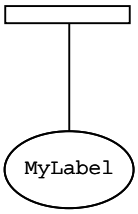

```
label MyLabel;
```

GFT **Core**

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**.




```
goto MyLabel;
```

GFT **Core**

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.

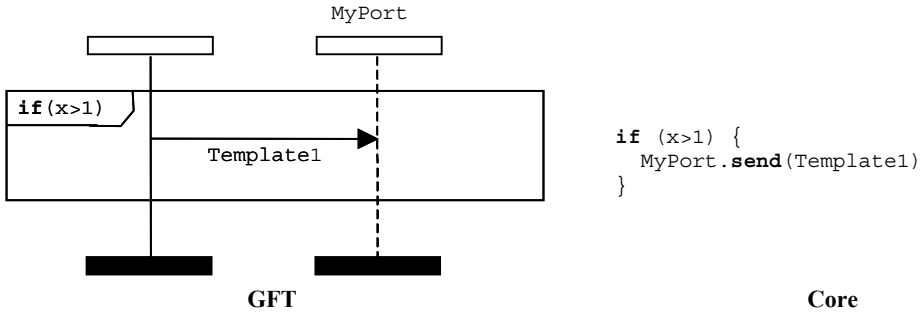


Figure 24/Z.142 – Instruction if

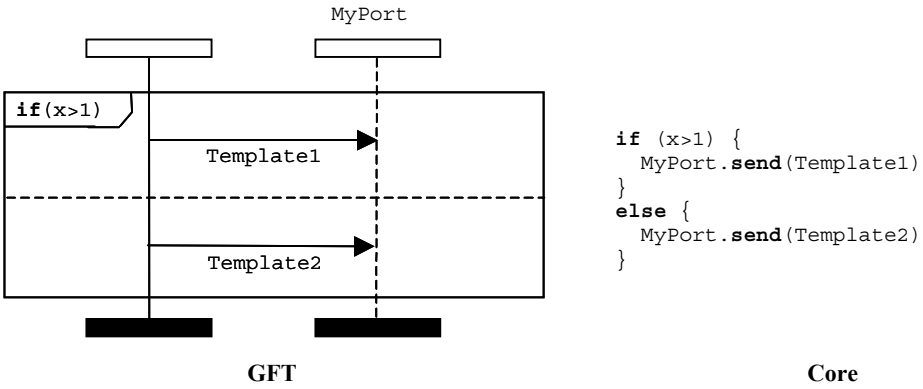


Figure 25/Z.142 – Instruction if-else

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`**.

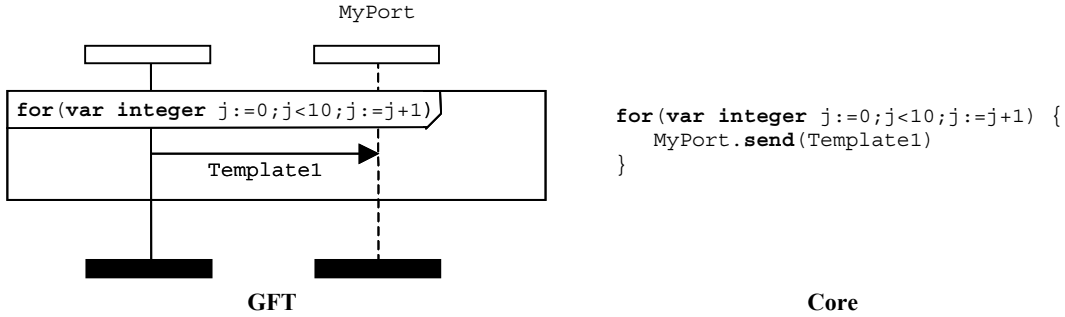


Figure 26/Z.142 – Instruction For

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`**.

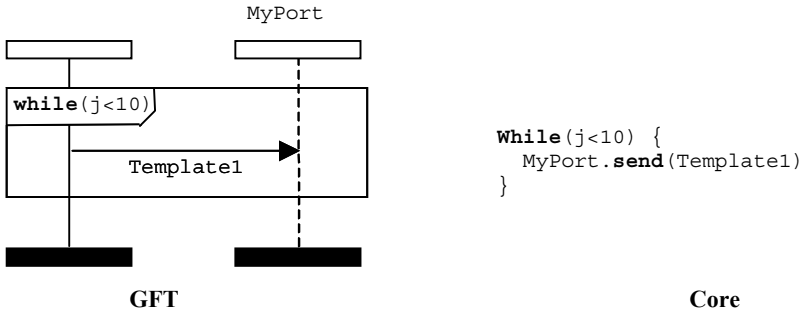


Figure 27/Z.142 – Instruction while

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**.

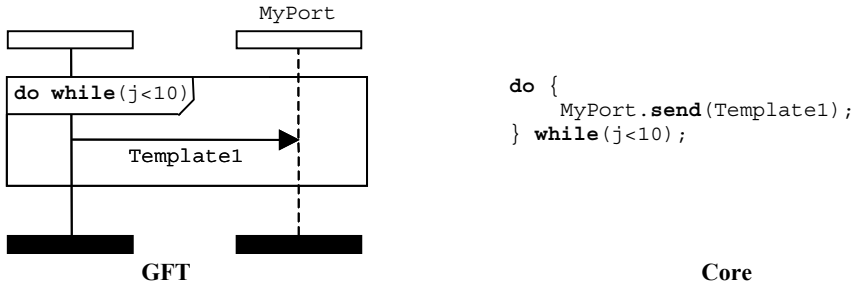


Figure 28/Z.142 – Instruction do-while

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`.

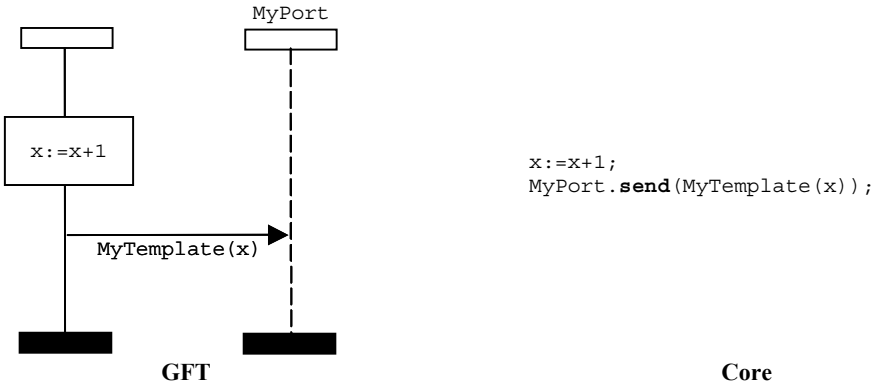


Figure 29/Z.142 – Comportement séquentiel

Le séquencement 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.

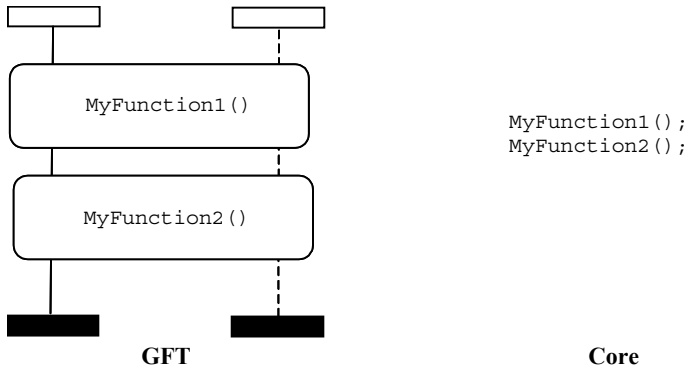


Figure 30/Z.142 – Séquencement utilisant des références

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.

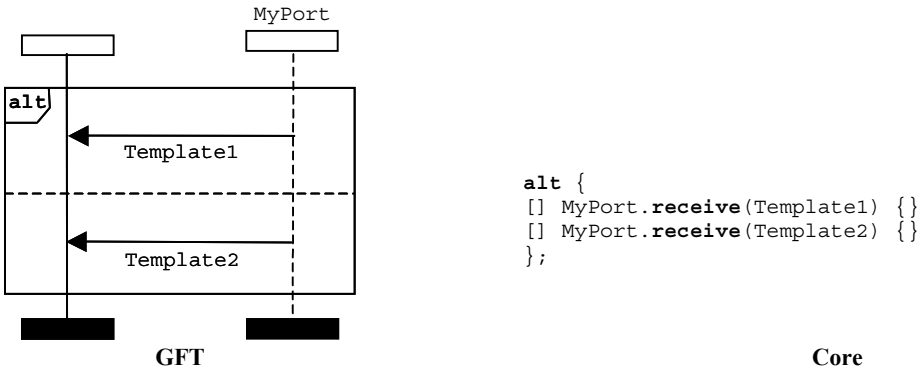


Figure 31/Z.142 – Instruction de comportement à variantes

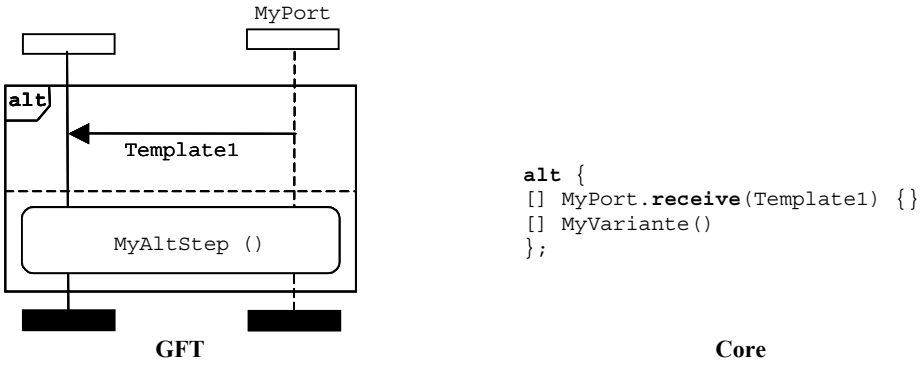


Figure 32/Z.142 – Comportement à variantes avec invocation d'une variante

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).

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<=1`.

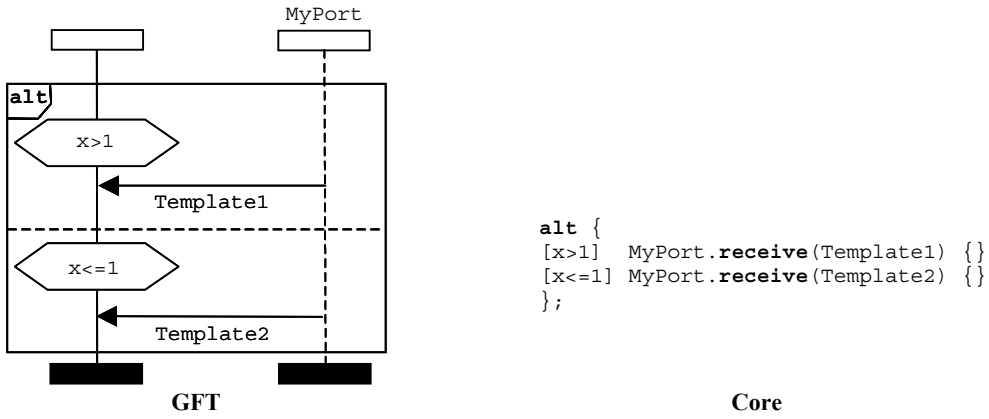
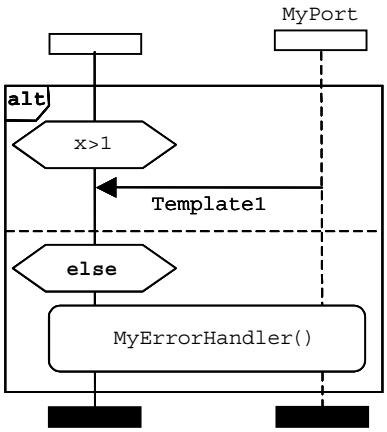


Figure 33/Z.142 – Activation/désactivation d'une alternative

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`**.




```
alt {
[x>1] MyPort.receive(Template1) {}
[else]MyErrorHandler()
};
```

GFT **Core**

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

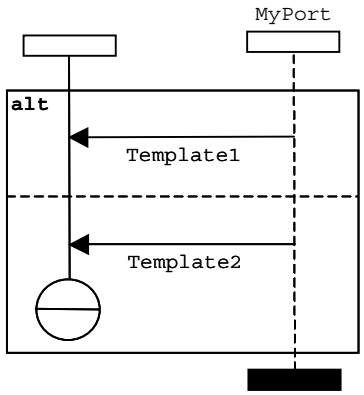
Il convient de noter que le symbole de référence dans une branche else devrait toujours prendre en compte toutes les instances de port, si des opérations de communication interviennent.

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.




```
alt {
[] MyPort.receive(Template1) {}
[] MyPort.receive(Template2) { repeat; }
};
```

GFT **Core**

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.

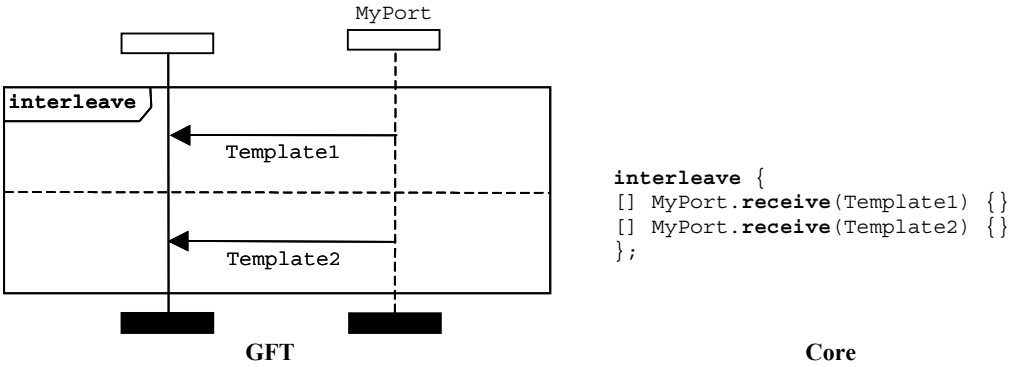


Figure 36/Z.142 – Instruction d'entrelacement

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.

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

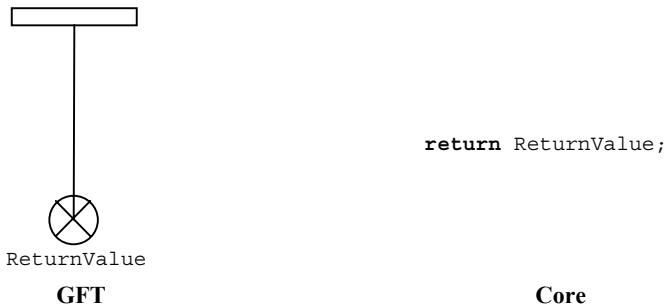


Figure 38/Z.142 – Symbole de retour avec utilisation d'une valeur de retour

11.6 Gestion des comportements par défaut

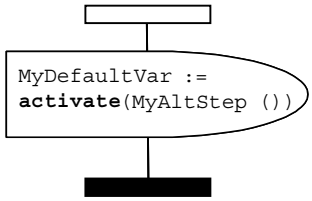
Le format GFT fournit une représentation graphique pour l'activation et la désactivation des comportements par défaut (voir le § 21/Z.140 [1]).

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 activate. Les paragraphes 11.3.1 et 11.3.3 illustrent la façon dont une variable nommée `MyDefaultType` est déclarée en format GFT.

11.6.2 Opération activate

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).



GFT

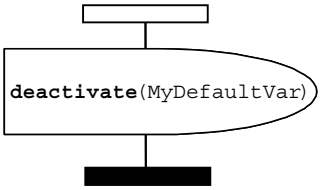
```
MyDefaultVar:=activate(MyAltStep());
```

Core

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

11.6.3 Opération deactivate

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.



GFT

```
deactivate(MyDefaultVar);
```

Core

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

11.7 Opérations de configuration

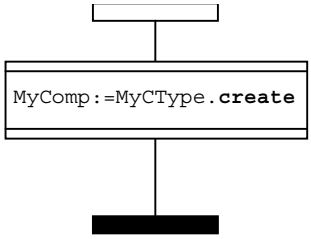
Les opérations de configuration sont utilisées pour mettre en place et commander des composantes de test. Elles ne doivent être utilisées que pour les diagrammes GFT de test élémentaire, de fonction ou de variante.

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.



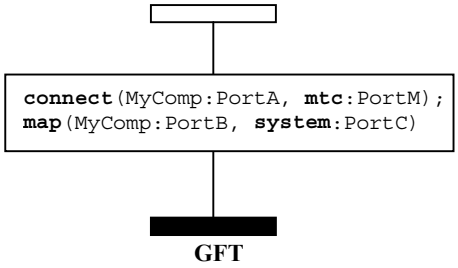

```
MyComp:=MyCType.create;
```

Core

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**.



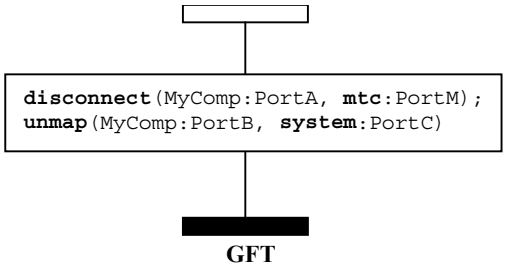

```
connect(MyComp:PortA, mtc:PortM);
map(MyComp:PortB, system:PortC);
```

Core

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**.



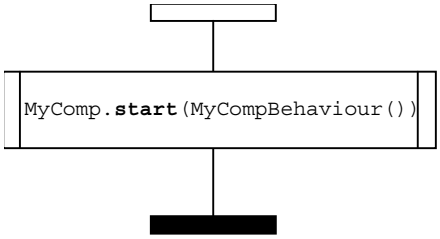

```
disconnect(MyComp:PortA, mtc:PortM);
unmap(MyComp:PortB, system:PortC);
```

Core

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, rattaché à l'instance de composante de test qui réalise l'opération (voir la Figure 44). Le symbole de lancement contient l'instruction **start**.




```
MyComp.start(MyCompBehaviour());
```

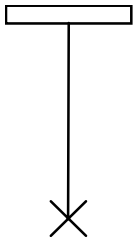
Core

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 (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.




```
stop;
```

Core

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. 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 composantes arrête l'instance qui exécute l'opération d'arrêt ou si elle arrête l'exécution du test.

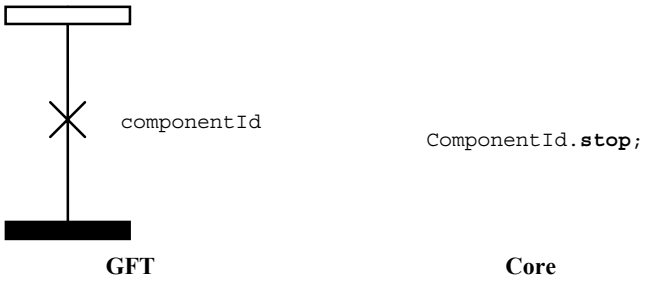


Figure 46/Z.142 – Opération d'arrêt de composantes de test

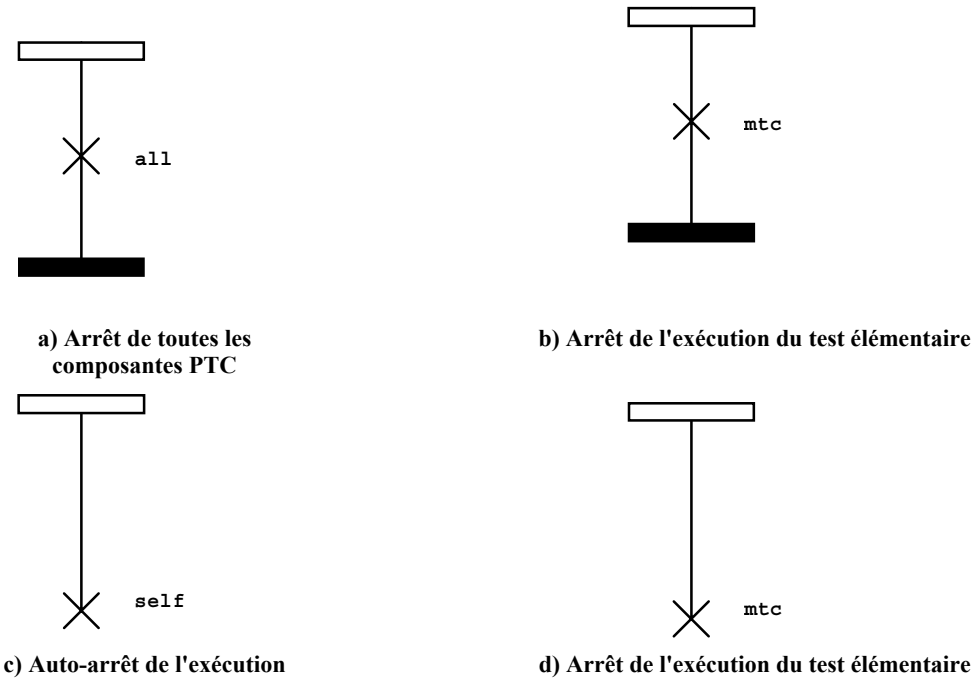


Figure 47/Z.142 – Utilisations particulières de l'opération d'arrêt de composantes de test

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 (voir la Figure 48). Le symbole de condition contient l'instruction **done**.

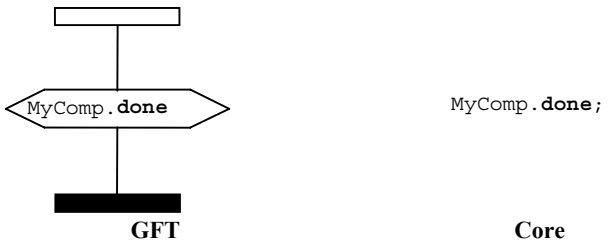


Figure 48/Z.142 – Opération de fin d'exécution

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:

a) 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`**);

b) les *opérations de réception*: une composante reçoit un message (operation **`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`**).

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).

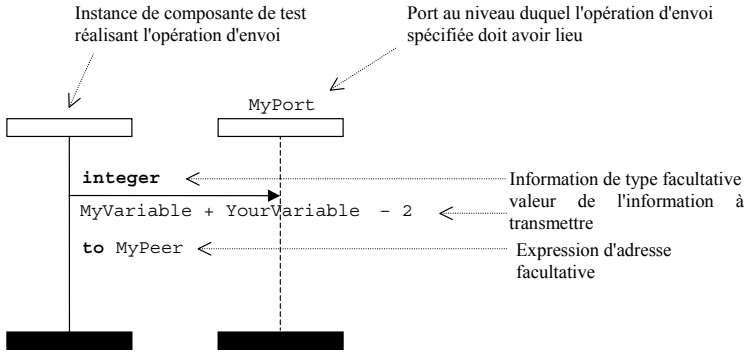


Figure 49/Z.142 – Format général des opérations d'envoi

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).

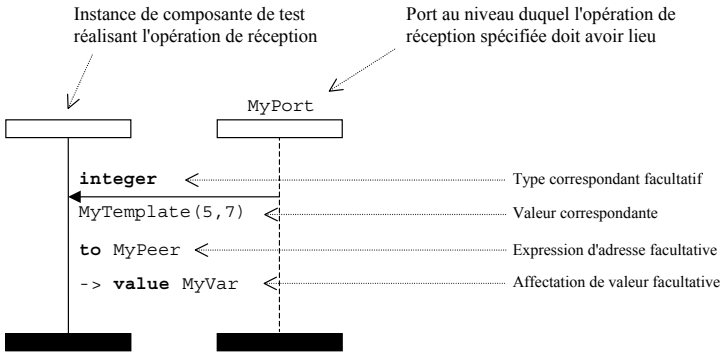


Figure 50/Z.142 – Format général des opérations de réception avec affectation d'adresse et de valeur

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

La partie *réception*:

a) spécifie le port au niveau duquel l'opération doit avoir lieu;

b) définit une partie correspondante comprenant une information de type facultative et la valeur correspondante qui spécifie l'entrée acceptable satisfaisant à l'instruction;

c) 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).

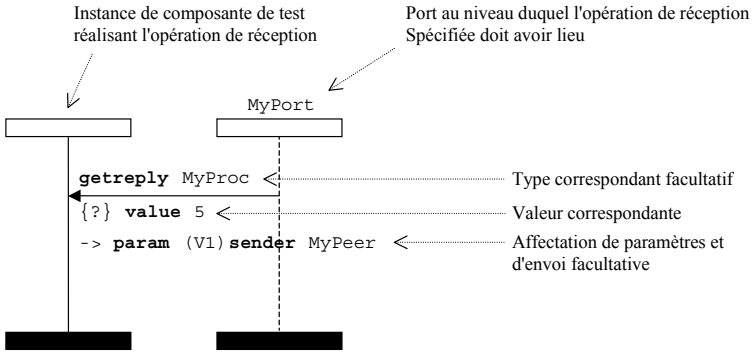
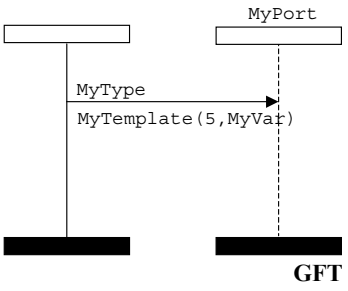


Figure 51/Z.142 – Format général des opérations de réception avec affectation de paramètres et d'envoi facultative

11.8.3 Communication en mode message

11.8.3.1 Opération d'envoi

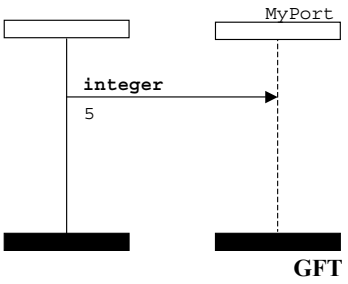
L'opération **send** doit être représentée par un symbole de message sortant allant de la composante de test à l'instance de port. L'information de type facultative doit être placée au-dessus de la flèche de message. Le modèle (en ligne) doit être indiqué sous la flèche (voir les Figures 52 et 53).




```
MyPort.send(MyType: MyTemplate(5,MyVar));
```

Core

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



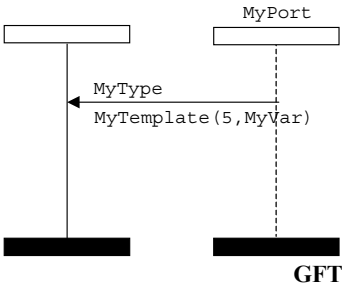

```
MyPort.send(integer:5);
```

Core

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

11.8.3.2 Opération de réception

L'opération **receive** 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).




```
MyPort.receive(MyType: MyTemplate(5, MyVar));
```

Core

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 **receive any** 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).

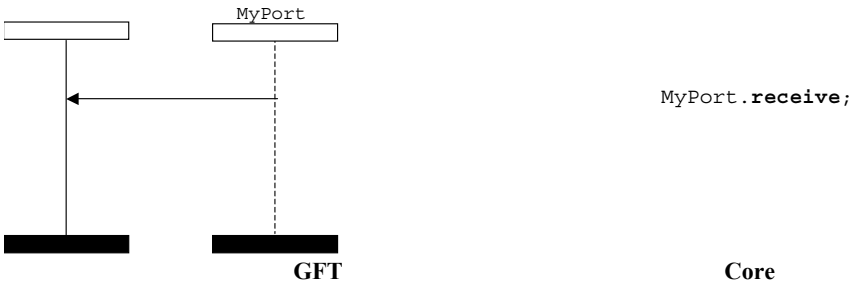


Figure 56/Z.142 – Réception de tout message

11.8.3.2.2 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).

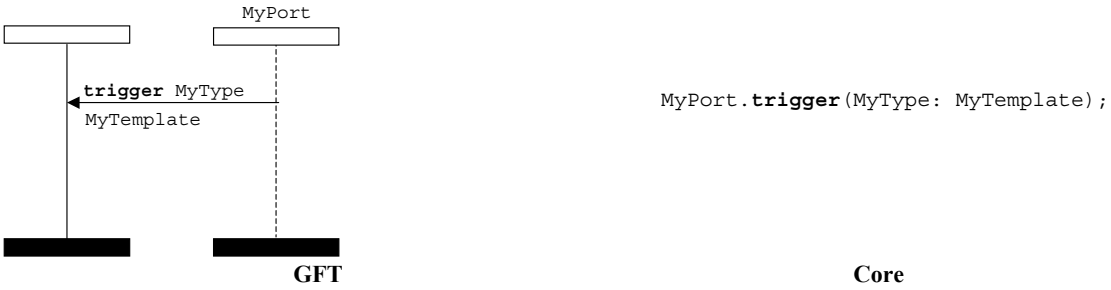


Figure 58/Z.142 – Opération de déclenchement avec modèle de référence

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).

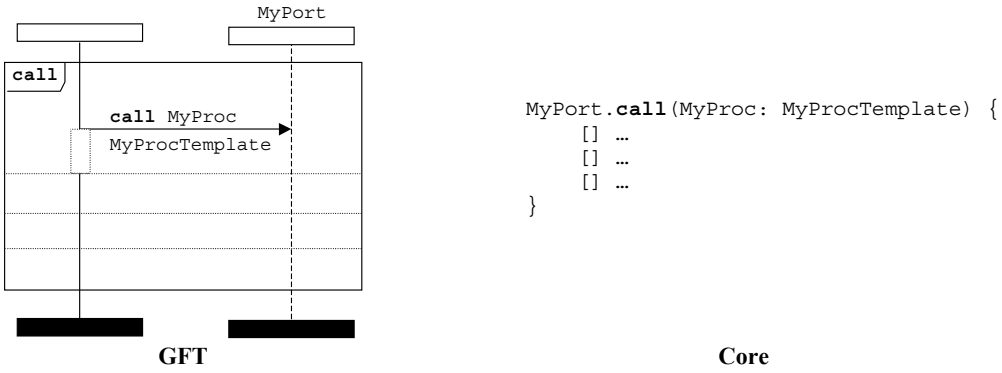


Figure 62/Z.142 – Opération d'appel bloquant avec référence de modèle

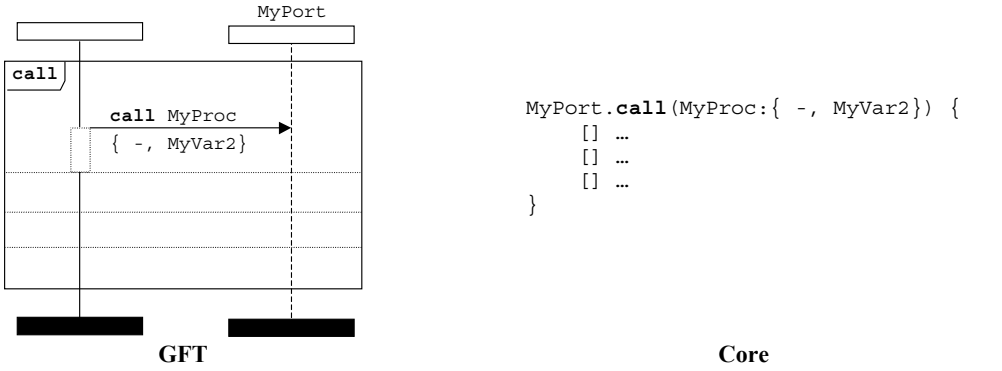
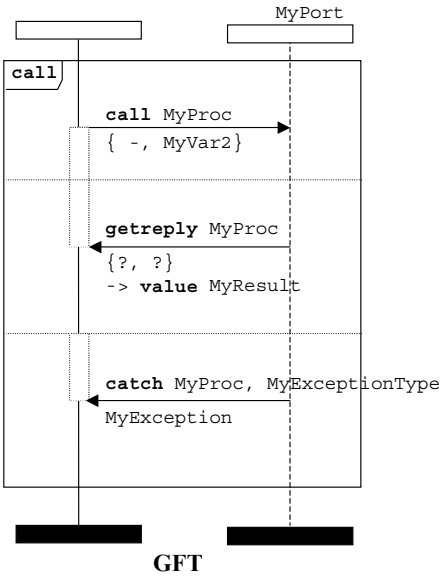


Figure 63/Z.142 – Opération d'appel bloquant avec modèle en ligne

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).

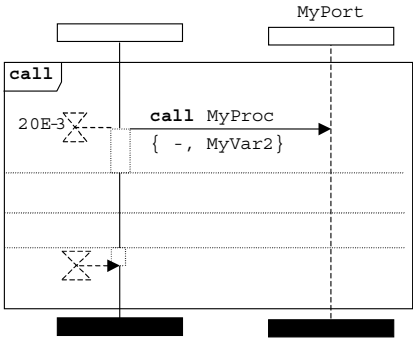



```
MyPort.call(MyProc:{ -, MyVar2}) {
    [] MyPort.getreply(MyProc:{?, ?})
       -> value MyResult { }
    [] MyPort.catch
       (MyProc, MyExceptionType: MyException) { }
}
```

Core

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

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



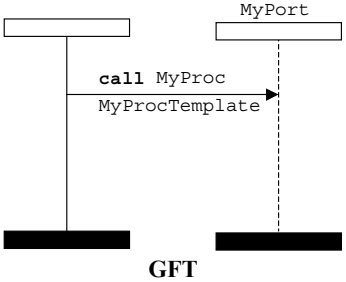

```
MyPort.call(MyProc:{ -, MyVar2},20E-3) {
    [] …
    [] …
    [] MyPort.catch(timeout) {
       …
       }
}
```

Core

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

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).




```
MyPort.call(MyProc: MyProcTemplate, nowait);
```

Core

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

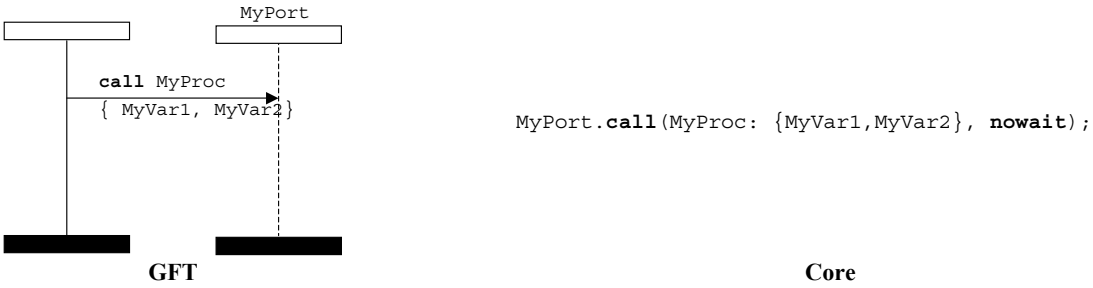


Figure 67/Z.142 – Opération d'appel non bloquant avec modèle en ligne

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).

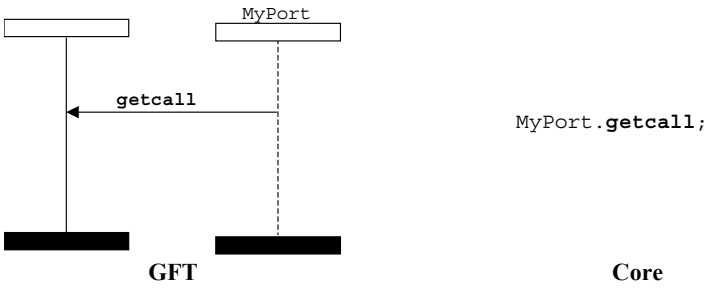


Figure 70/Z.142 – Opération acceptation de tout appel

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).

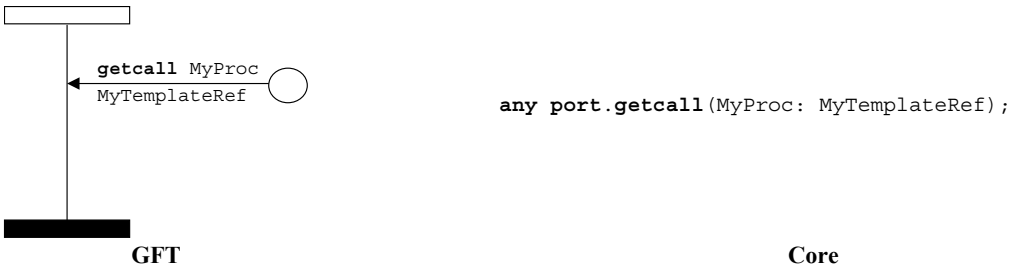
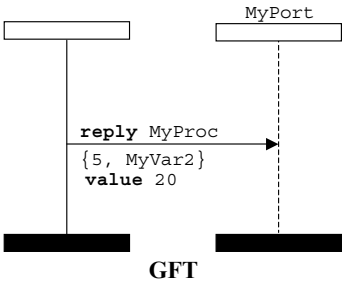


Figure 71/Z.142 – Opération obtention d'appel pour tout port avec modèle de référence

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




```
MyPort.reply(MyProc: {5, MyVar2} value 20);
```

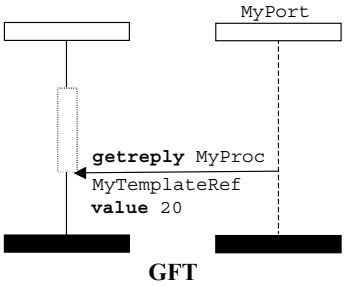
Core

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).

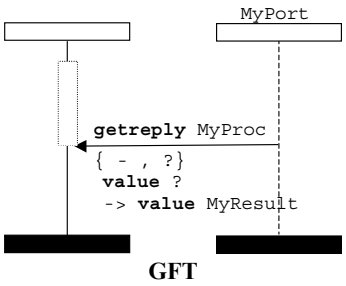
La signature doit être placée au-dessus de la flèche après le mot clé **getreply**. Le modèle (en ligne) doit être placé sous la flèche.




```
:
MyPort.getreply(MyProc: MyTemplateRef value 20);
:
```

Core

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




```
:
MyPort.getreply(MyProc: {-, ?}
value ?) -> value MyResult;
:
```

Core

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

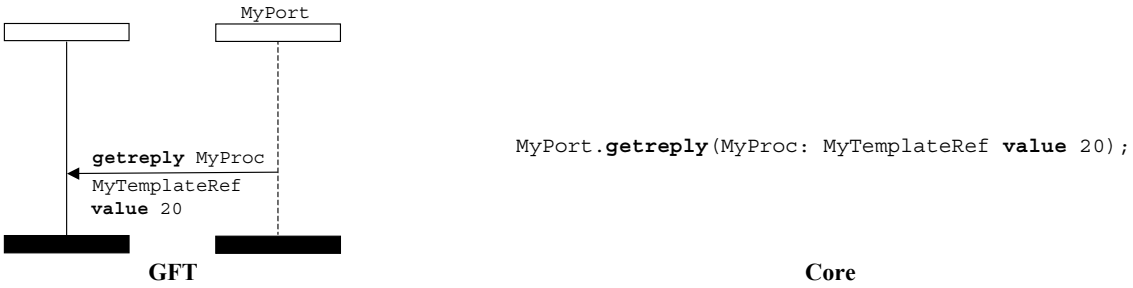


Figure 76/Z.142 – Opération d'obtention de réponse avec référence de modèle (hors d'un symbole d'appel)

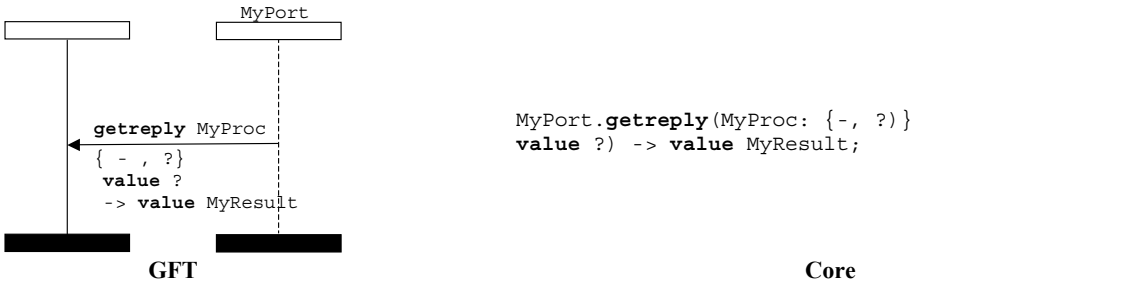


Figure 77/Z.142 – Opération d'obtention de réponse avec modèle en ligne (hors d'un symbole d'appel)

11.8.4.4.1 Acceptation de toute réponse à tout appel

L'opération 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 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).

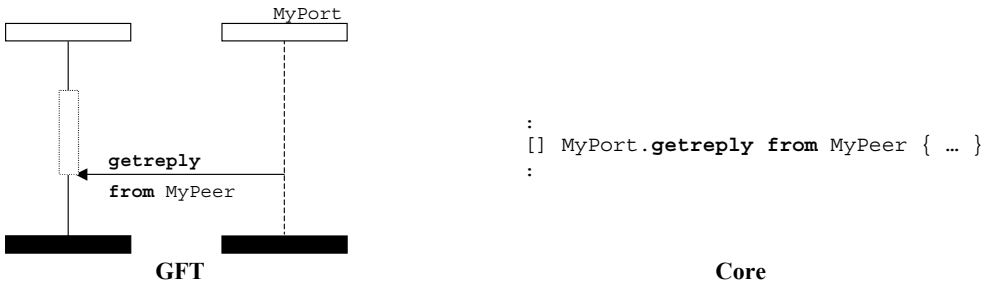


Figure 78/Z.142 – Acceptation de toute réponse à tout appel (dans un symbole d'appel)

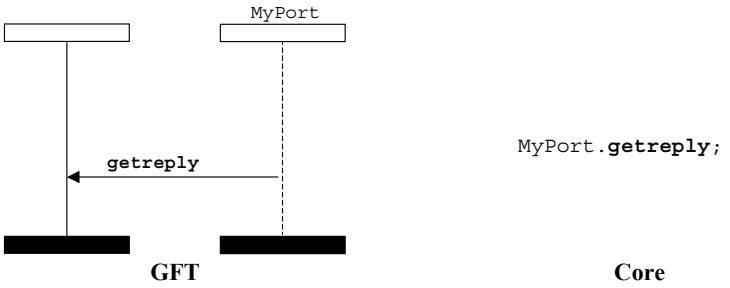


Figure 79/Z.142 – Acceptation de toute réponse à tout appel (hors d'un symbole d'appel)

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.

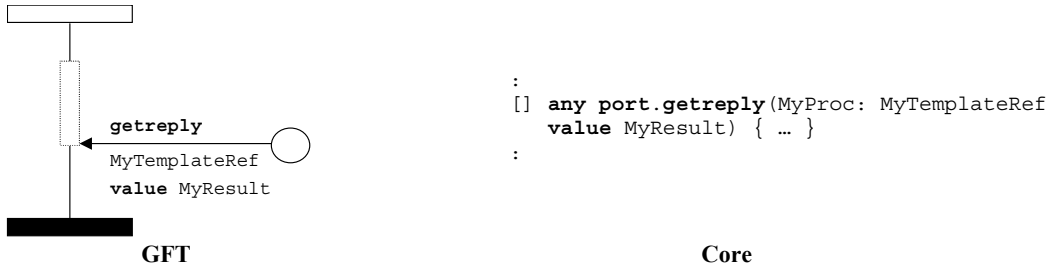


Figure 80/Z.142 – Obtention de réponse pour tout port (dans un symbole d'appel)

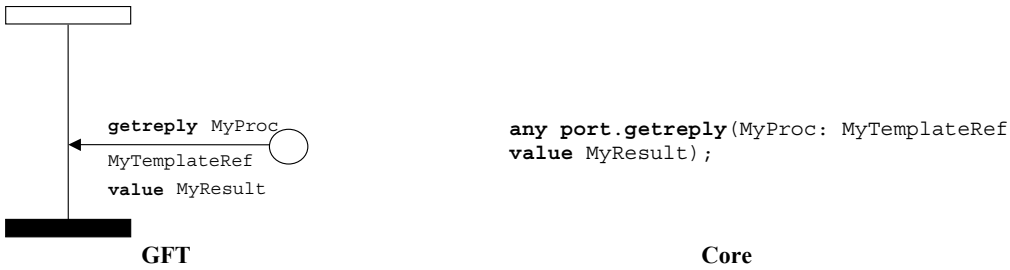


Figure 81/Z.142 – Obtention de réponse pour tout port (hors d'un symbole d'appel)

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).

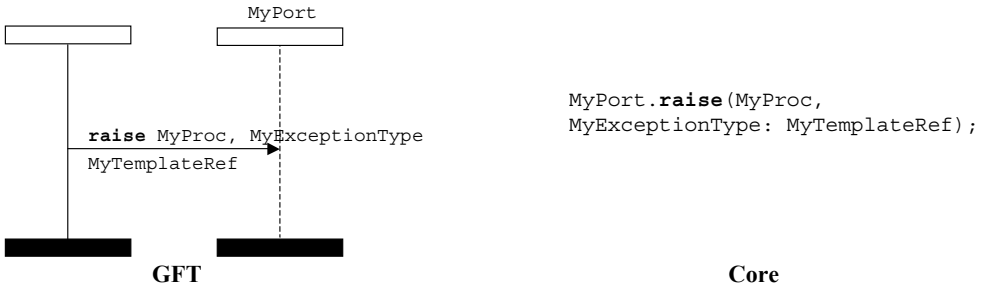


Figure 82/Z.142 – Opération de déclenchement d'une exception avec référence de modèle

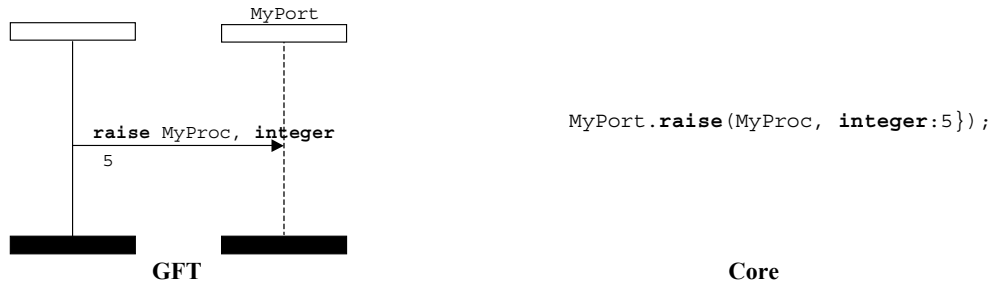


Figure 83/Z.142 – Opération de déclenchement d'une exception avec modèle en ligne

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.

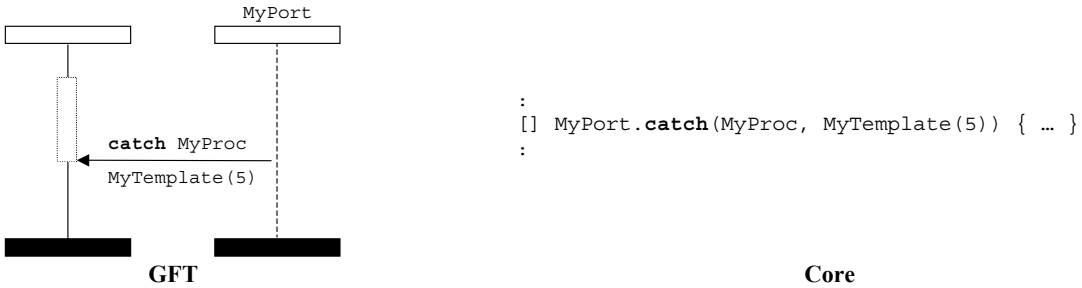


Figure 84/Z.142 – Opération d'acquisition avec référence de modèle (dans un symbole d'appel)

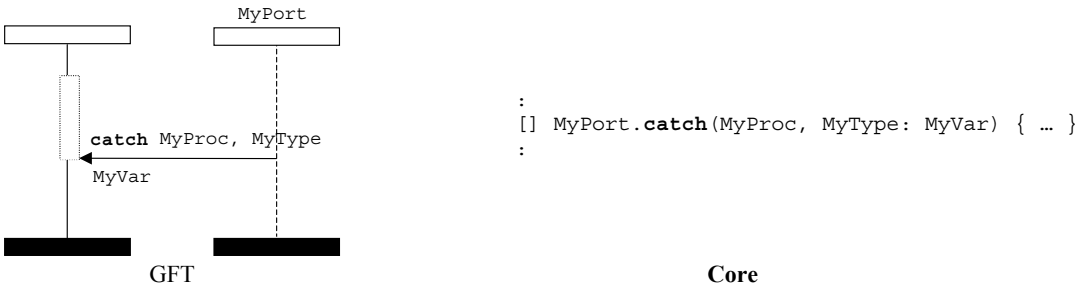


Figure 85/Z.142 – Opération d'acquisition avec modèle en ligne (dans un symbole d'appel)

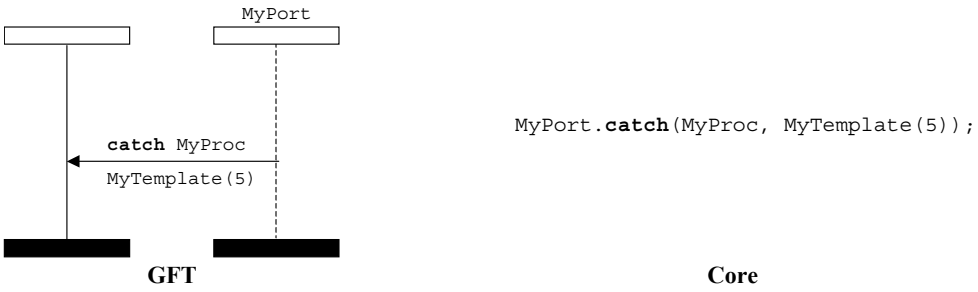


Figure 86/Z.142 – Opération d'acquisition avec référence de modèle (hors d'un symbole d'appel)

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

11.8.4.6.1 Exception de temporisation

L'opération d'exception de temporisation doit être représentée par un symbole de temporisation 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.

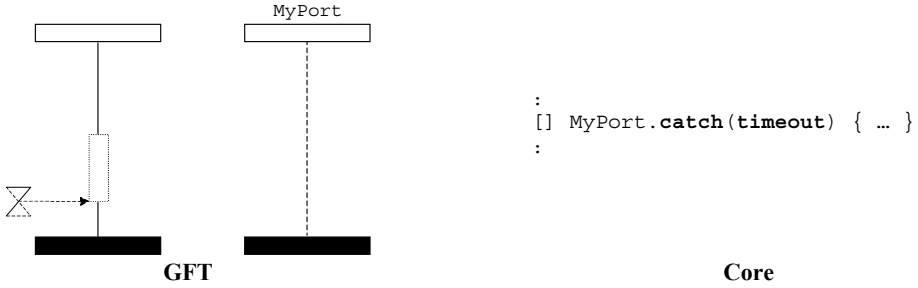


Figure 88/Z.142 – Exception de temporisation (dans un symbole d'appel)

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.

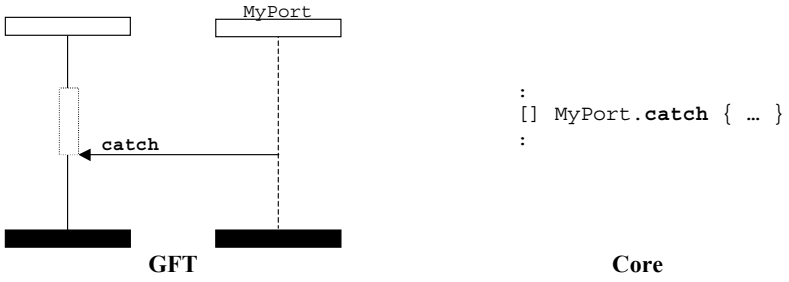


Figure 89/Z.142 – Acquisition de toute exception (dans un symbole d'appel)

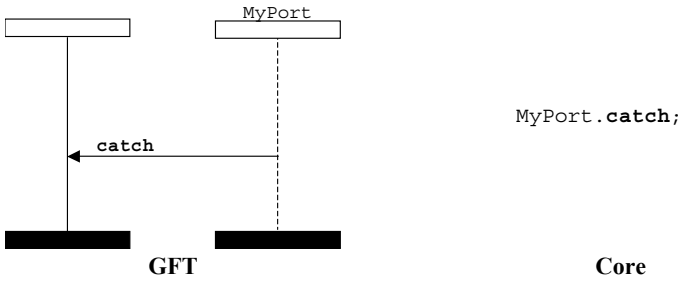


Figure 90/Z.142 – Acquisition de toute exception (hors d'un symbole d'appel)

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.

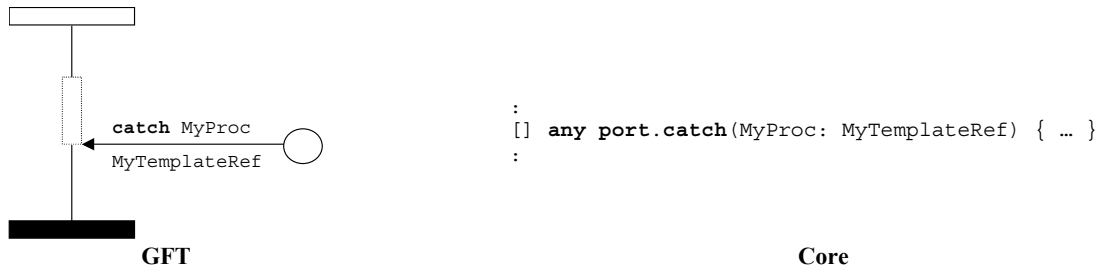


Figure 91/Z.142 – Acquisition pour tout port (dans un symbole d'appel)

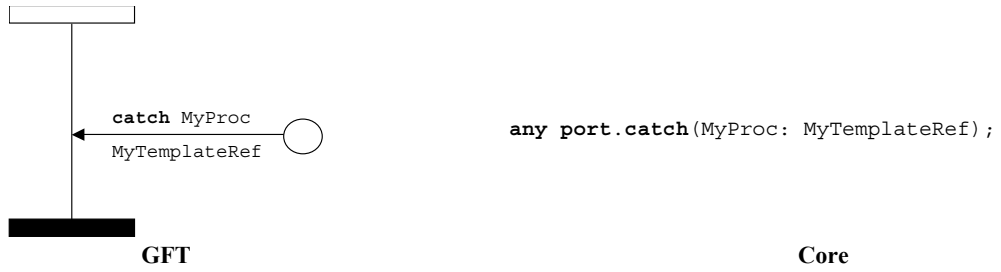


Figure 92/Z.142 – Acquisition pour tout port (hors d'un symbole d'appel)

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

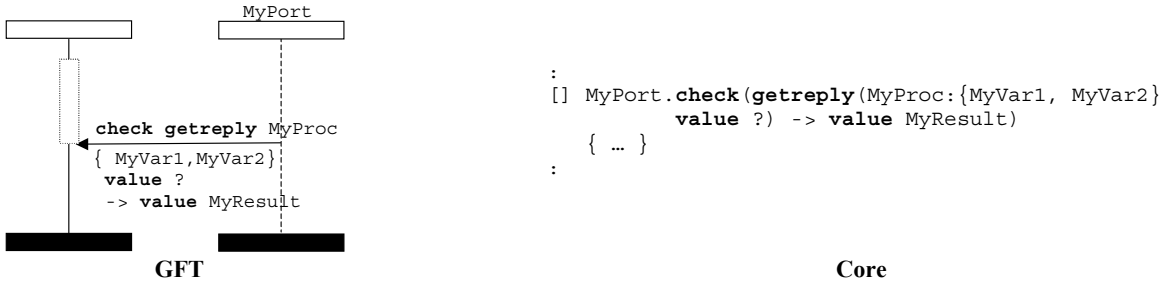


Figure 94/Z.142 – Vérification d'une opération d'obtention de réponse (dans un symbole d'appel)

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.

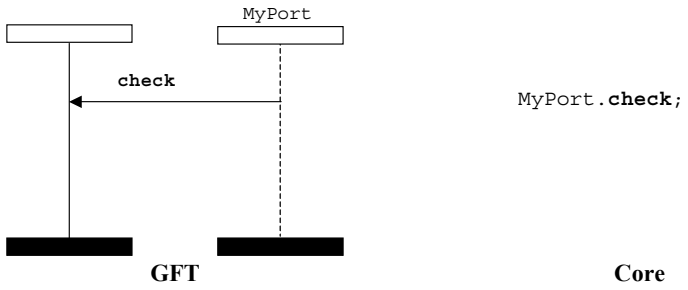


Figure 96/Z.142 – Opération de vérification totale

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.

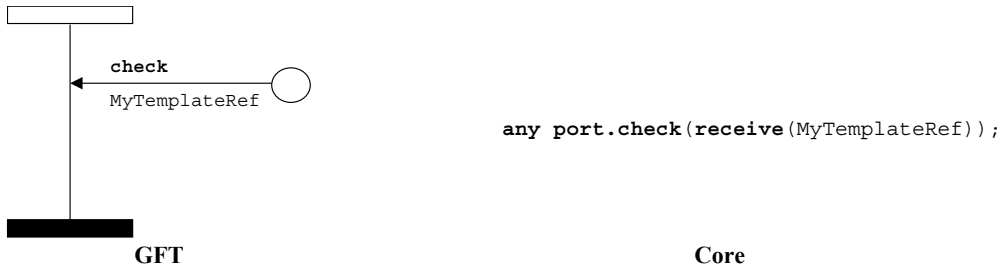


Figure 97/Z.142 – Vérification d'une opération de réception sur tout port quelconque

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).

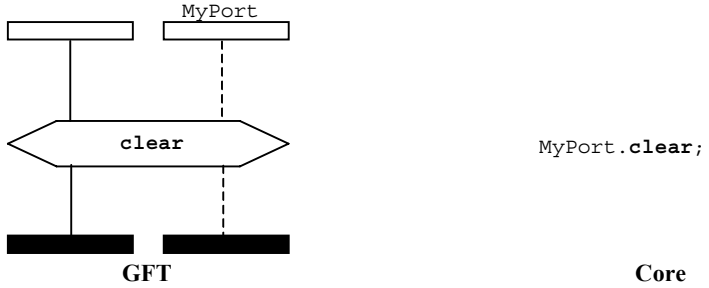


Figure 98/Z.142 – Opération de libération de port

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).

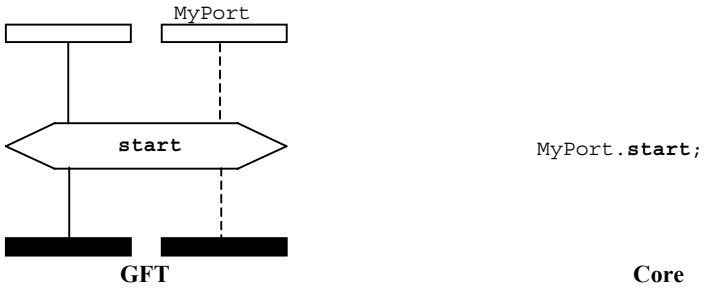


Figure 99/Z.142 – Opération d'ouverture de port

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).

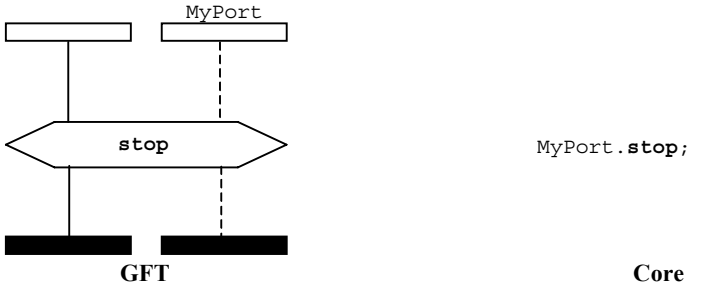


Figure 100/Z.142 – Opération de fermeture de port

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.

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 indiqué par du texte lorsqu'elle est utilisée.

11.9.1 Opération d'armement de temporisateur

Pour l'opération d'armement de temporisateur, le symbole d'armement 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 d'armement de temporisateur

11.9.2 Opération de désarmement de temporisateur

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

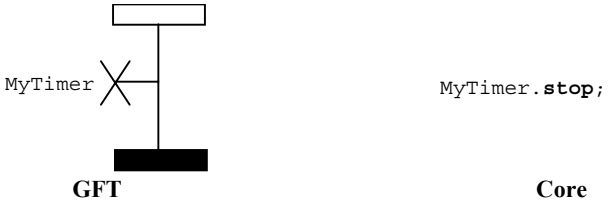


Figure 103/Z.142 – Opération de désarmement de temporisateur

Les symboles d'armement et de désarmement 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 d'armement de temporisateur (Figure 104).

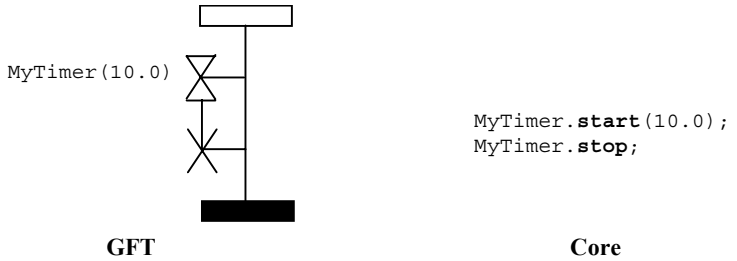


Figure 104/Z.142 – Liaison entre le symbole d'armement et le symbole de désarmement de temporisateur

11.9.3 Opération de fin de temporisation

Pour l'opération de fin de temporisation, le symbole de fin de temporisation 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 de fin de temporisation

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

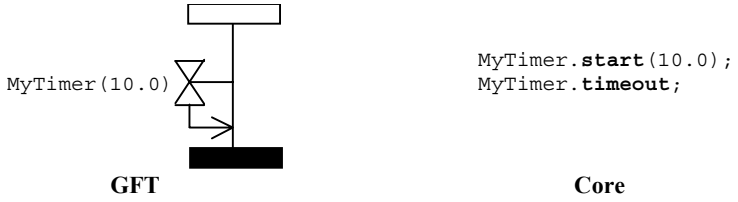


Figure 106/Z.142 – Liaison entre les symboles d'armement de temporisateur et de fin de temporisation

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).

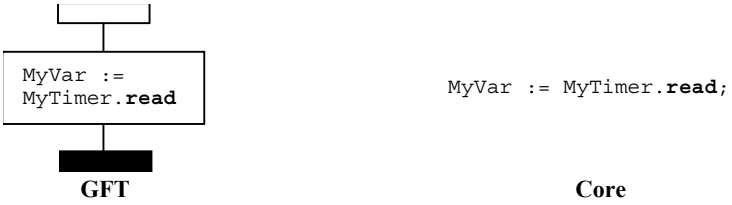


Figure 107/Z.142 – Opération de lecture de temporisateur

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

L'opération de désarmement de temporisateur peut être appliquée à tous (**all**) les temporisateurs (Figure 108).

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

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

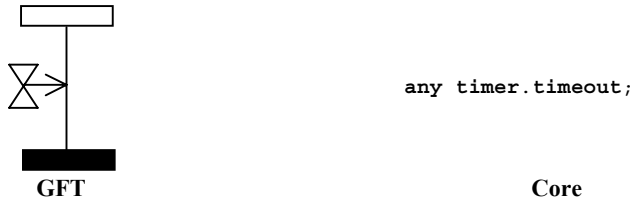


Figure 109/Z.142 – Opération de fin de temporisation à partir d'un temporisateur quelconque

11.10 Opération de verdict de test

L'opération de définition de verdict **setverdict** est représenté 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.

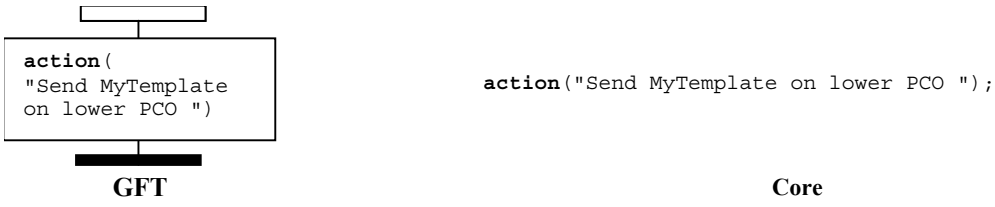
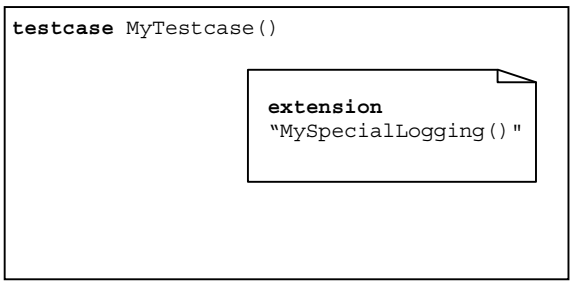


Figure 111/Z.142 – Actions externes

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.




```
testcase MyTestcase() {
    :
}
with {
       extension "MySpecialLogging()"
}
```

GFT **Core**

Figure 112/Z.142 – Spécification d'attributs

Annexe A (normative)

GFT BNF

A.1 Meta-language for GFT

The graphical syntax for GFT is defined on the basis of the graphical syntax of MSC [3]. The graphical syntax definition uses a meta-language, which is explained in 1.3.4/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 [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 abc (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

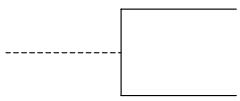
```
TextArea ::=
                TextSymbol
                ***contains*** ( { TTCN3Comments } [ MultiWithAttrib ] { TTCN3Comments } )
```

Note that there is no explicit rule for TTCN3 comments, they are explained in ES 201 873-1 [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 ::=
```

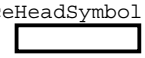
```
LocalDefinition ::=
            ConstDef
        |   VarInstance
        |   TimerInstance
```

/* 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::=
```

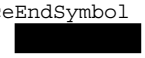

A.3.2.3 Control Instances

```
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 ::=
```

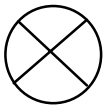


```
StopArea::=
        StopSymbol
        is associated with ( Expression )
```

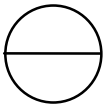
/* STATIC SEMANTICS – The expression shall refer to either the mtc or to self */

```
StopSymbol::=
```

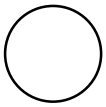
```
ReturnArea::=
        ReturnSymbol
        [ is associated with Expression ]

ReturnSymbol::=
```



```
RepeatSymbol::=
```

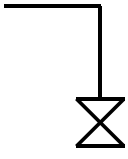


```
GotoArea::=
        GotoSymbol
        contains LabelIdentifier

GotoSymbol::=
```

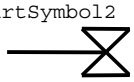


A.3.3 Timer

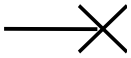
```
InstanceTimerEventArea::=
        InstanceTimerStartArea
    |   InstanceTimerStopArea
    |   InstanceTimeoutArea


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


TimerStartSymbol ::=
        TimerStartSymbol1 | TimerStartSymbol2

TimerStartSymbol1 ::=
```



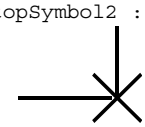
```
TimerStartSymbol2 ::=
```



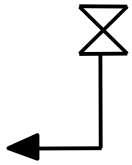
```
InstanceTimerStopArea ::=
        TimerStopArea1 | TimerStopArea2

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

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

TimerStopSymbol1 ::=
```



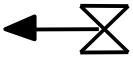
```
TimerStopSymbol2 ::=
```



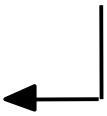
```
InstanceTimeoutArea ::=
        TimeoutArea1 | TimeoutArea2

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
```

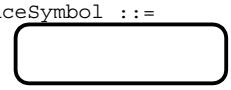
/* STATIC SEMANTICS – Declarations of constants and variables with create, activate, and execute statements as well as with function invocations of user-defined functions must not be made textually within an action box, but must be made graphically within create, default, execute, and reference symbols,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

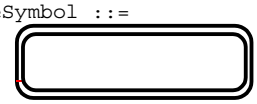
```
PortInvocationBeginSymbol ::=
        VoidSymbol

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 ::=
```

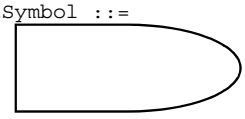


A.3.6 Activation/Deactivation of Defaults

```
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

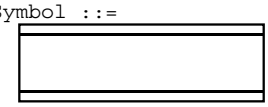
A.3.7.1 Creation of Test Components

```
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 ]
```

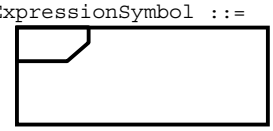
/* STATIC SEMANTICS – The number of InstanceInlineExpressionSeparatorSymbol per component and port instances has to adhere to the number of SeparatorSymbols contained within the inline expression symbol: the InstanceInlineExpressionSeparatorSymbol on component and port instances are used to attach the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

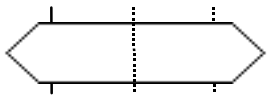
```
InstanceConditionEndSymbol ::=
        VoidSymbol

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 */

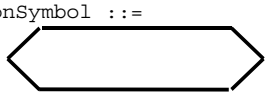
```
PortOperationArea ::=
        ConditionSymbol
        contains PortOperationText
        is attached to InstanceConditionBeginSymbol
        is attached to InstanceConditionEndSymbol
        [   is attached to { PortInlineExpressionBeginSymbol }+ set
            is attached to { PortInlineExpressionEndSymbol }+ set ]
        is attached to InstancePortOperationArea
        is attached to PortConditionArea
```

/* STATIC SEMANTICS – The condition symbol shall be attached to either to 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, its 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
```

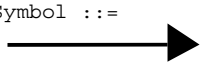
```
InstanceSetVerdictArea ::=
        InstanceConditionBeginSymbol
        is followed by InstanceConditionEndSymbol
        is attached to InstanceAxisSymbol
        is attached to SetVerdictArea

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 ]
```

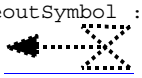
```
                        [ FromClause ] [ PortRedirectWithParam ] )
        is attached to InstanceGetreplyEventArea
        is attached to PortGetreplyOutEventArea

/* 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 – A n 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 – A n 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 – A n 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
```

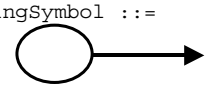
/* STATIC SEMANTICS – The trigger keyword shall be put on top of the message symbol */
/* STATIC SEMANTICS – A type, if existent, shall be put on top of the message symbol */
/* STATIC SEMANTICS – A derived definition, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – A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

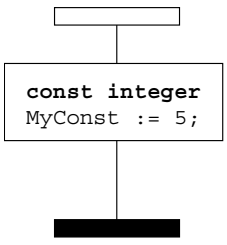
```
InstanceLabellingArea ::=
        LabellingSymbol
        contains LabelIdentifier
        is attached to InstanceAxisSymbol

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

LabellingSymbol ::=
```

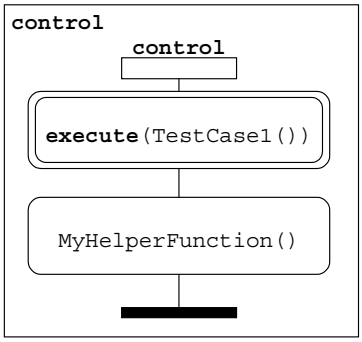
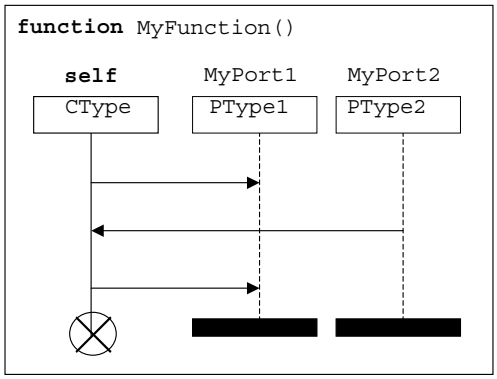
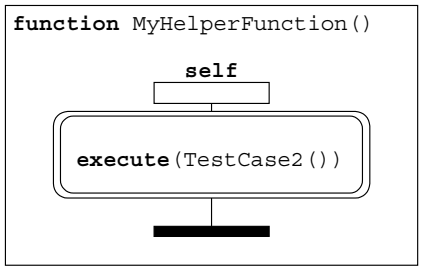
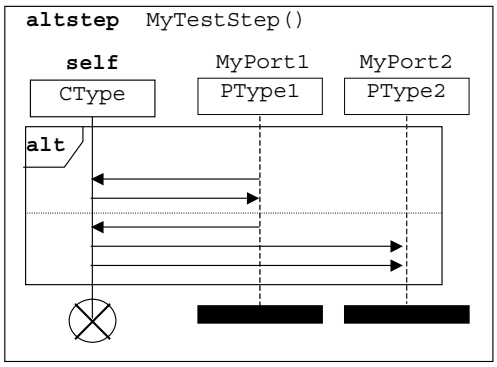


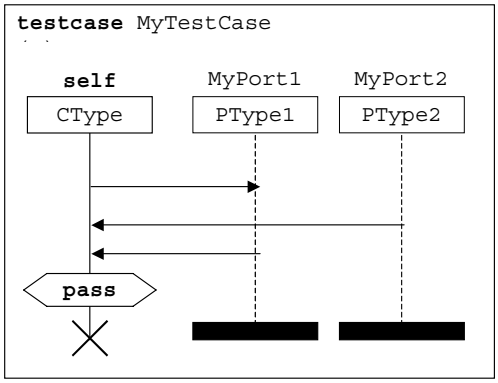
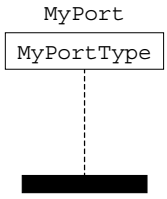
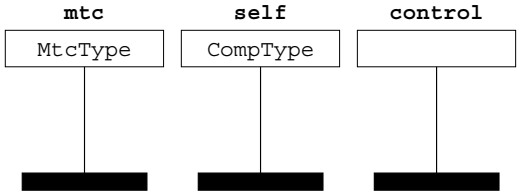
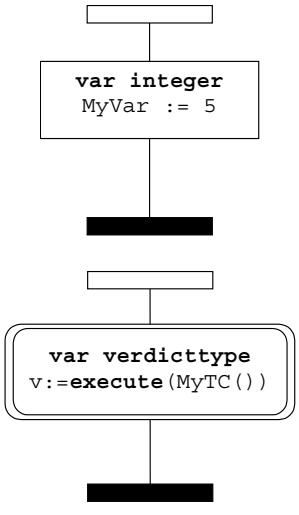
Annexe B (informative)

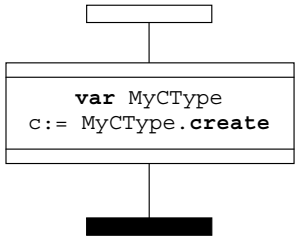
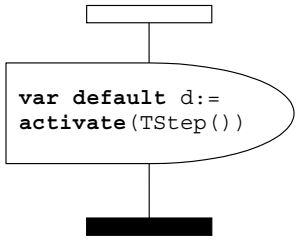
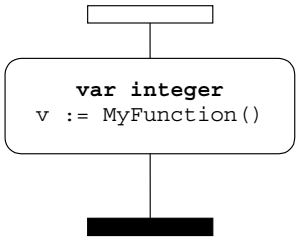
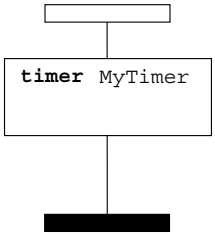
Reference guide for GFT

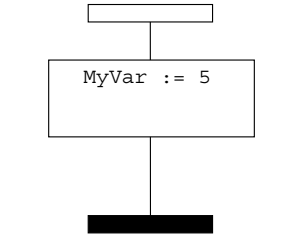
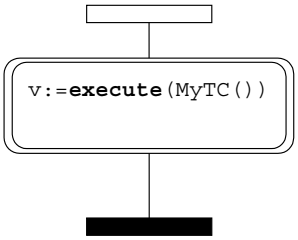
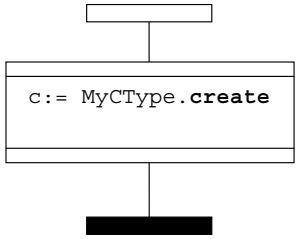
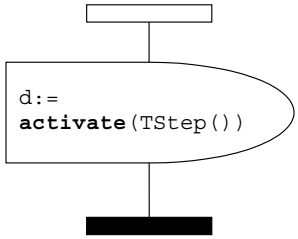
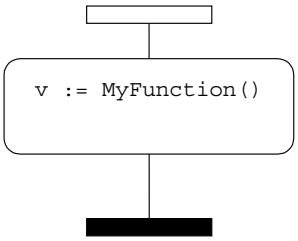
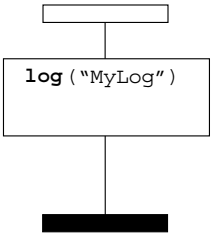
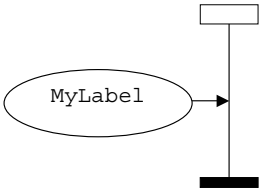
This annex lists the main TTCN-3 language elements and their representation in GFT. For a complete description of the GFT symbols and their use, please refer to the main text.

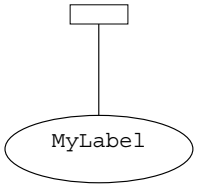
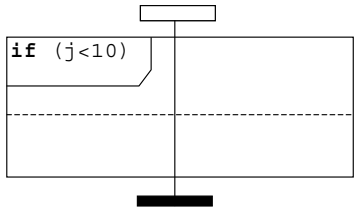
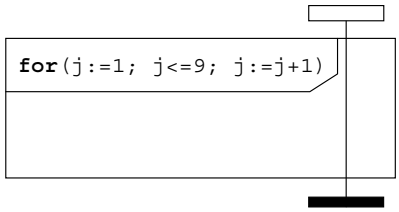
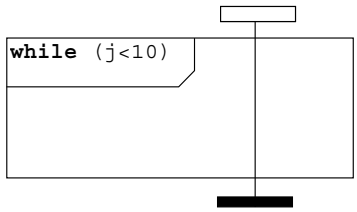
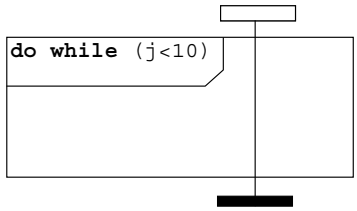
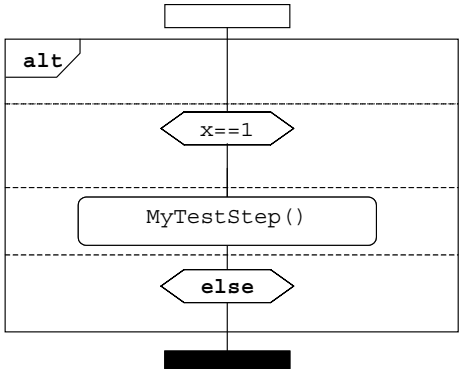
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
Module Definitions			
TTCN-3 module definition	**module**		No special GFT symbol, i.e., the core language or another presentation format may be used.
Import of definitions from other module	**import**		No special GFT 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**	**const** integer MyConst := 5; 	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.

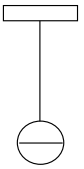
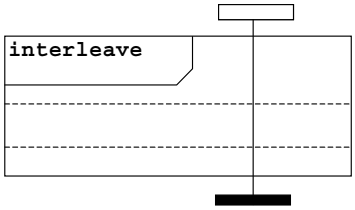
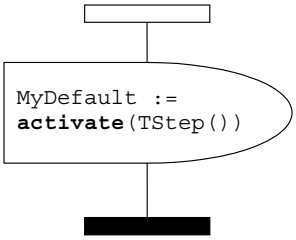
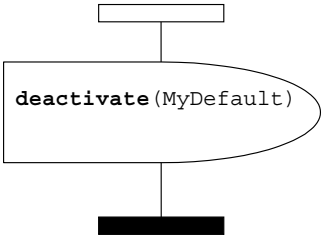
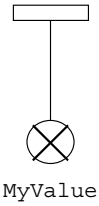
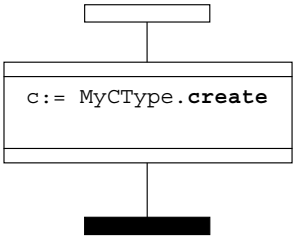
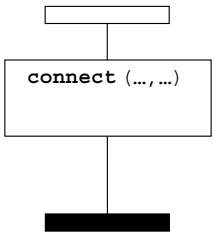
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
Control definitions	**control**		GFT control diagram represents the control part of a TTCN-3 module.
Function definitions	**function**	 	GFT function diagrams are used to represent functions. GFT function diagrams may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

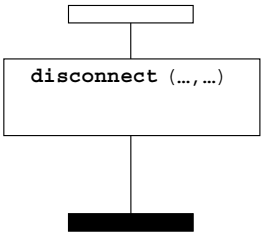
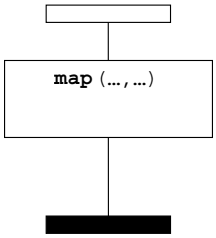
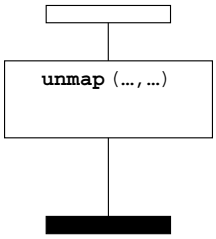
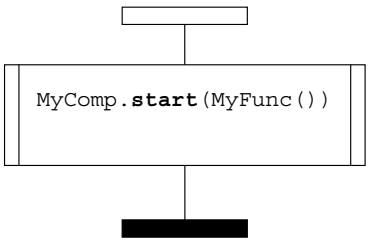
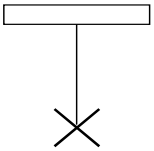
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
Test case definitions	**testcase**		GFT test case diagrams are used to represent test cases.
Usage of Component Instances and Ports			
Port instance			A Port in a test case, test step and function diagram is represented by and 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			An **mtc** instance represents the main test component in a test case diagram. A **self** instance represents a test component in a test step or function diagam. A **control** instance represents the instance that executes the module control part in a control diagram.
Declarations			
Variable declarations	**var**	**var integer** MyVar := 5 	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.

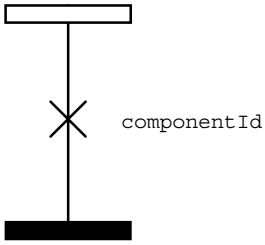
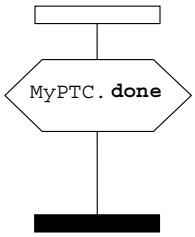
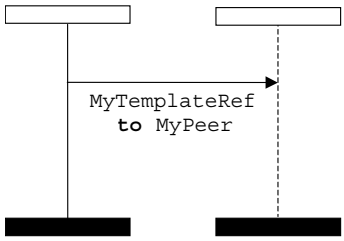
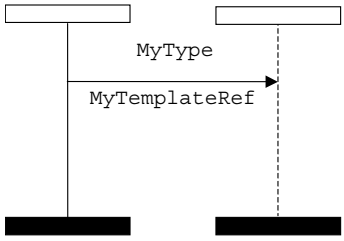
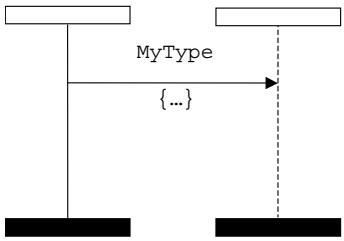
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
			Variable declaration within a test component creation symbol.
			Variable declaration within a default activation symbol.
			Variable declaration within a reference symbol.
Timer declarations	**timer**	**timer** MyTimer 	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.

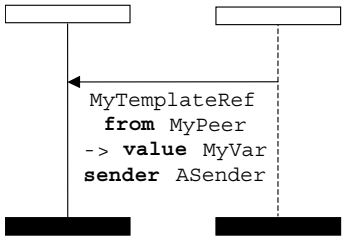
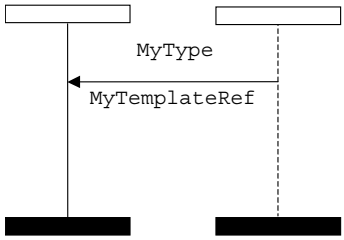
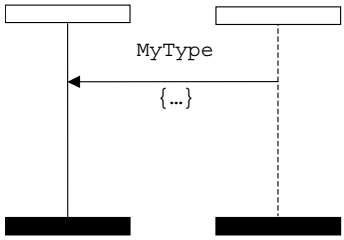
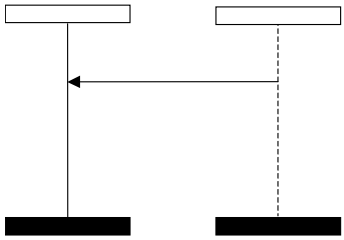
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
Assignments	**:=**		Assignment in an action box.
			Assignment within a test case execution symbol.
			Assignment within a test component creation symbol.
			Assignment within a default activation symbol.
			Assignment within a reference symbol.
Logging	**log**		The log statement is put into an action box.
Label and Goto	**label**		Definition of a label.

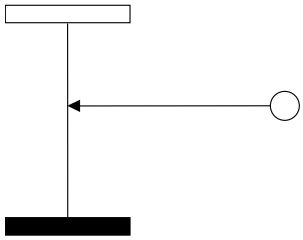
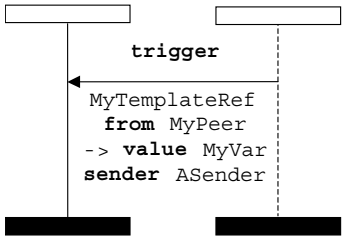
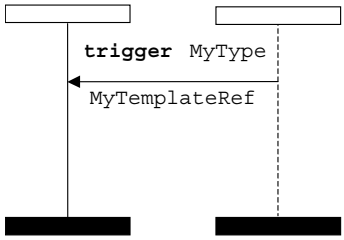
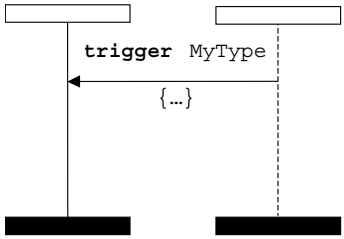
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
	goto		Go to label.
If-else	**if (…) {…} else {…}**		
For loop	**for (…) {…}**		
While loop	**while (…) {…}**		
Do while loop	**do {…} while (…)**		
Behavioural program statements			
Alternative behaviour	**alt {…}**		

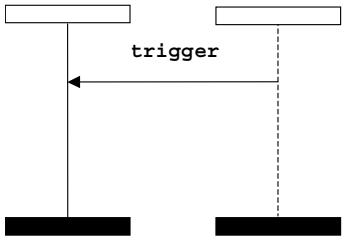
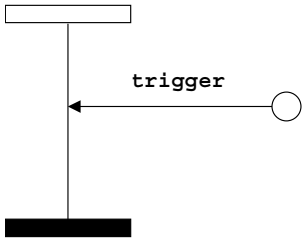
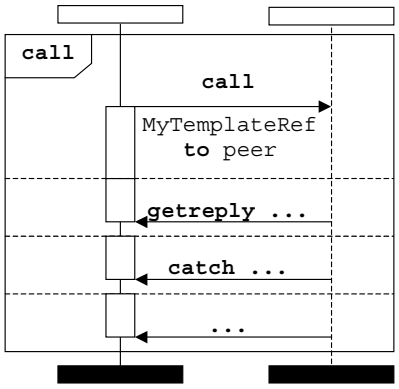
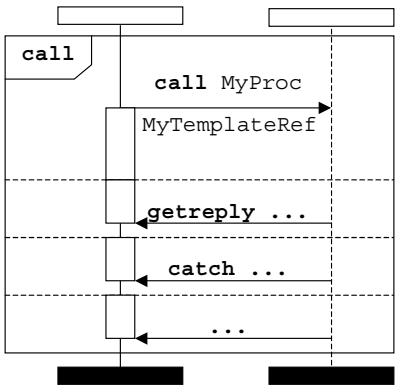
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
Repeat	**repeat**		To be used within alternative behaviour and test steps.
Interleaved behaviour	**interleave** {…}		
Activate a default	**activate**		The activate statement is put into a default symbol.
Deactivate a default	**deactivate**		The deactivate statement is put into a default symbol.
Returning control	**return**		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.

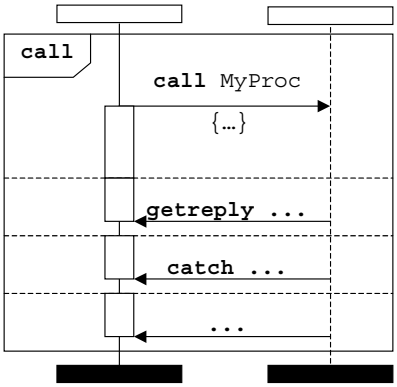
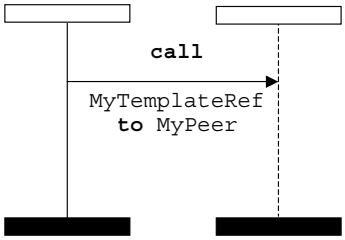
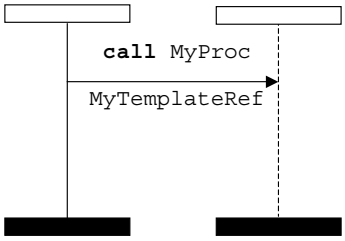
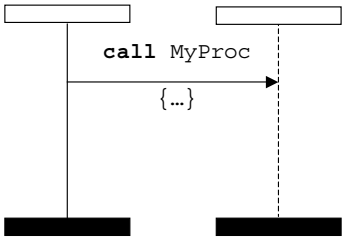
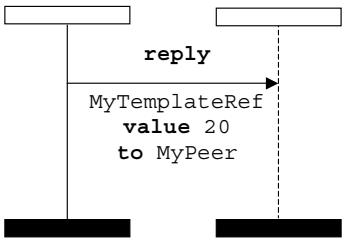
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
Disconnect two components	**disconnect**		The disconnect statement is put into an action box.
Map port to test system interface	**map**		The map statement is put into an action box.
Unmap port from test system interface	**unmap**		The unmap statement is put into an action box.
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.

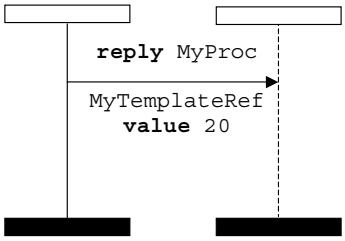
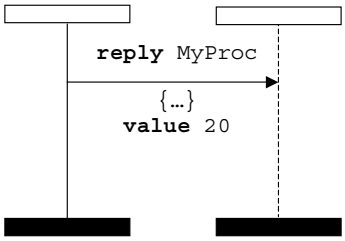
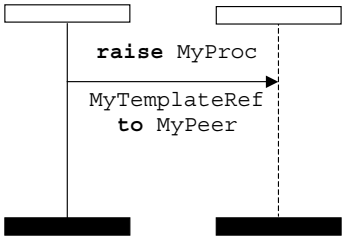
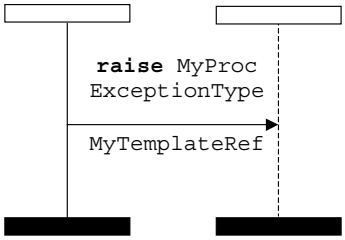
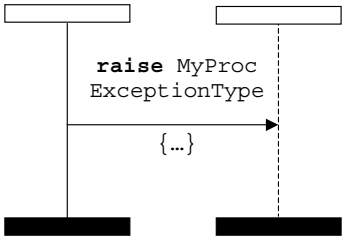
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
of another test component			The component identifier is put near to the stop symbol.
Check termination of a PTC	**running**		No special GFT symbol, used within expressions.
Wait for termination of a PTC	**done**		The 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.
			Send a message defined by a template reference and with type information. An (optional) **to**-directive may be present to identify the peer entity uniquely.
			Send a message defined by an inline template definition. An (optional) **to**-directive may be present to identify the peer entity uniquely.

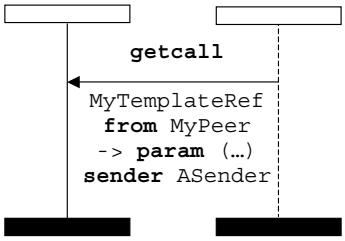
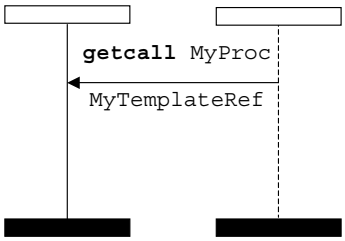
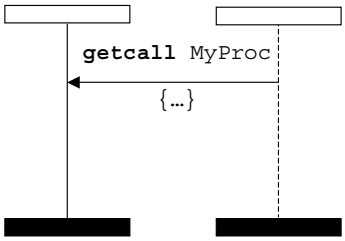
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
Receive message	**receive**		Receive a message with a value defined by a template reference but without type information. The (optional) **from**-directive denotes that the sender of the message shall be identified by variable `MyPeer`. The (optional) **value**-directive assigns received message to variable `MyVar`. The (optional) **sender**-directive retrieves the identifier of the sender and stores it in variable `ASender`.
			Receive a message with a value defined by a template reference and with type information. 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.
			Receive a message with a value defined by an inline template definition. 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.
			Receive any message (no value and no type is specified). 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.

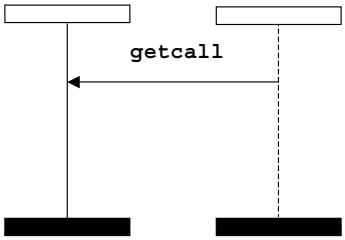
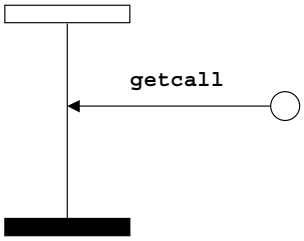
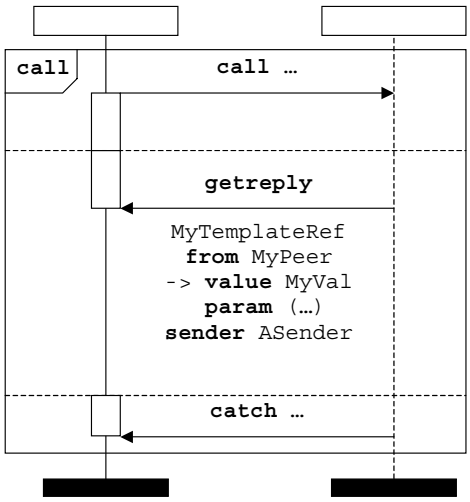
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
			Receive any message (no value and no type is specified) from any port. The message value to be received from any port may be restricted by means referring to templates or by using inline templates. Optional **from**-,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

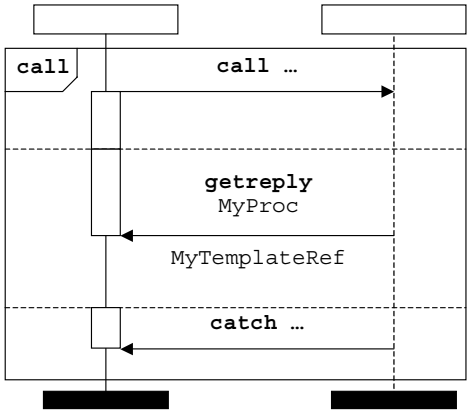
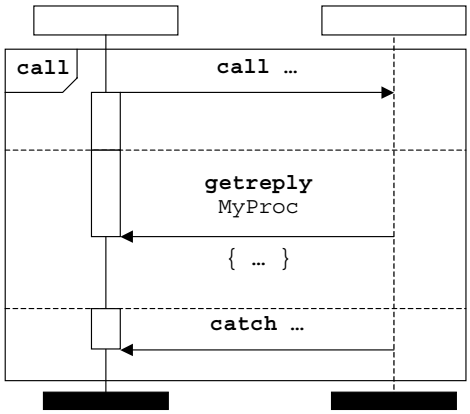
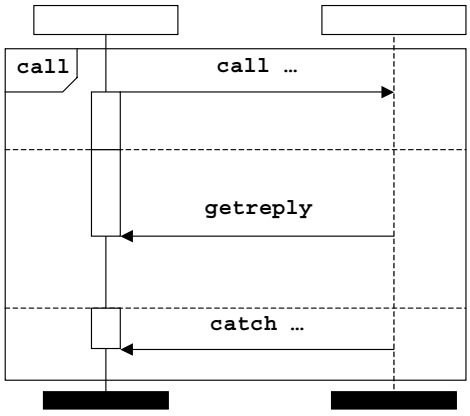
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
		 	Trigger on any message (no value and no type is specified). Optional **from**-, **value**- and **sender**-directives may be present to identify the sender of the message, to assign the message to a variable (of type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

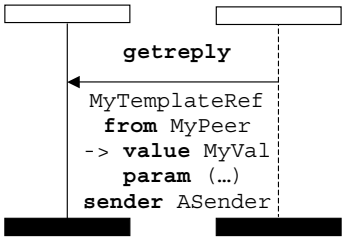
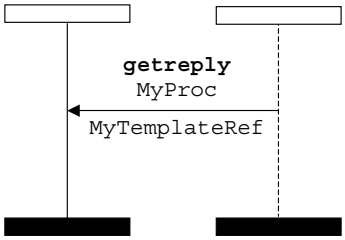
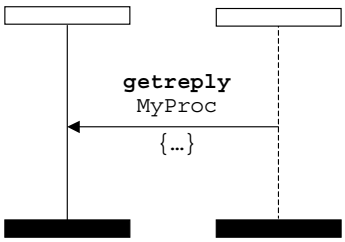
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
			Invoking a blocking procedure by using an inline template. An (optional) **to**-directive may be present to identify the peer entity uniquely. The call body, i.e., possible **getreply** and **catch** operations, is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

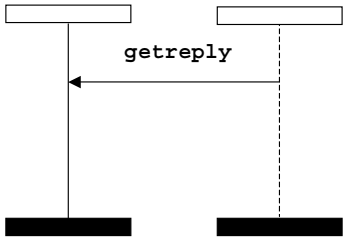
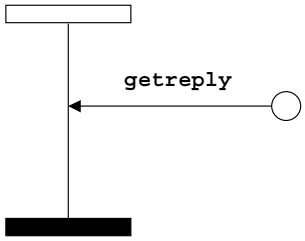
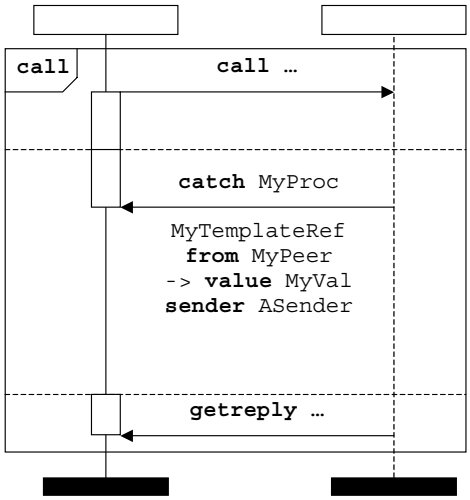
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
			Reply to a remote procedure call of MyProc. The reply is defined by a template reference and the possible return value (**value**-directive). An (optional) **to**-directive may be present to identify the peer entity uniquely.
			Reply to a remote procedure call of MyProc. 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.
Raise exception (to an accepted call)	**raise**		Raise an exception to an accepted call of MyProc. 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.
			Raise an exception to an accepted call of MyProc. 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.
			Raise an exception to an accepted call of MyProc. The exception is defined by its type and an inline template. An (optional) **to**-directive may be present to identify the peer entity uniquely.

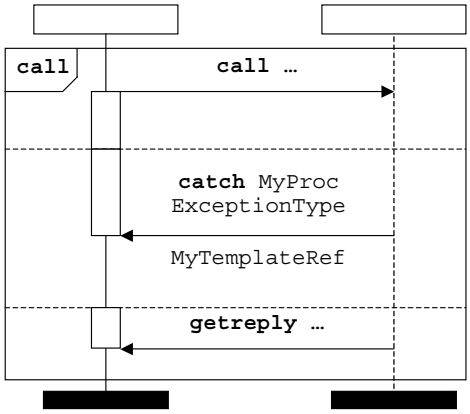
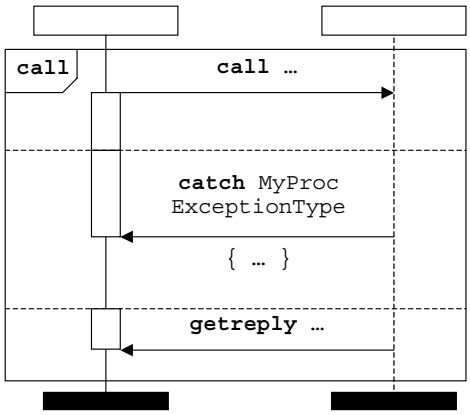
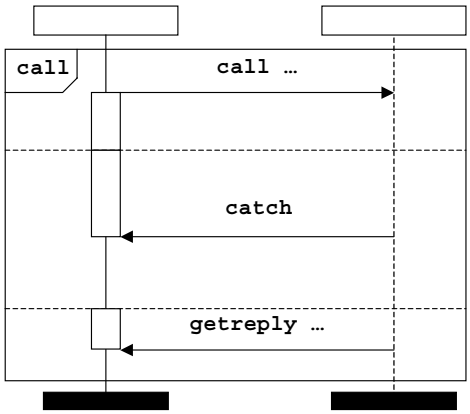
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
Accept procedure call from remote entity	**getcall**	  	Accept a procedure call from a remote entity. The call signature has to match the conditions defined by the template reference. NOTE – The signature information is part of the template definition. The (optional) **from**-directive denotes that the sender of the call shall be identified by variable `MyPeer`. The (optional) **param**-directive assigns in-parameter values to Variables. The (optional) **sender**-directive retrieves the identifier of the sender and stores it in variable `ASender`. Accept a procedure call from a remote entity. The call signature has to match the conditions defined by signature reference and the template reference. 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. 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. 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.

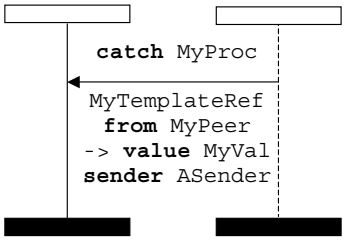
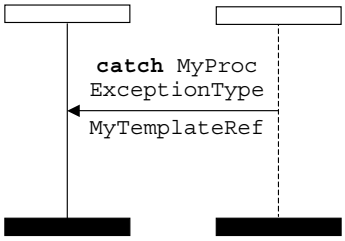
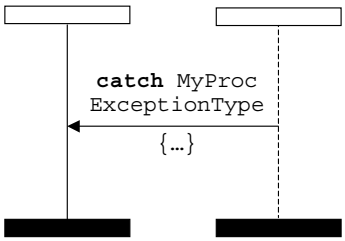
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
			Accept any procedure call from any remote entity. Optional **from**- and **sender**-directives may be present to identify the sender of the call or to retrieve the identifier of the peer entity.
			Accept any procedure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

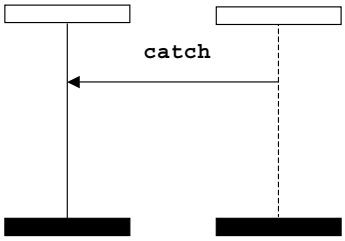
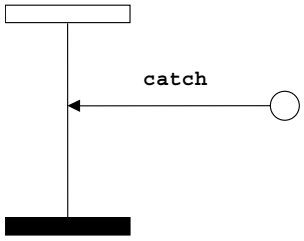
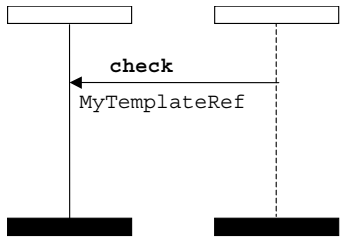
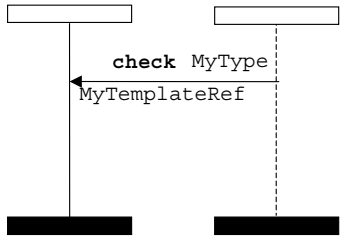
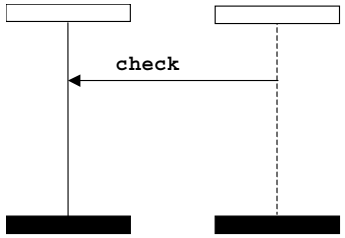
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
		  	Receive a response from a blocking call. The reply has to match the conditions defined by signature reference and the template reference. Optional **from**-, **value**-, **param**-and **sender**-directives may be 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. Receive a response from a blocking call. The reply has to match the conditions defined by signature reference and the inline template definition. 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. Accept any response from a blocking call.

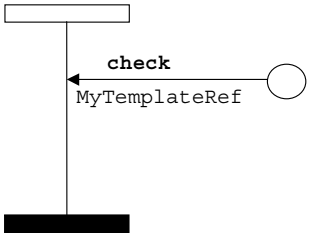
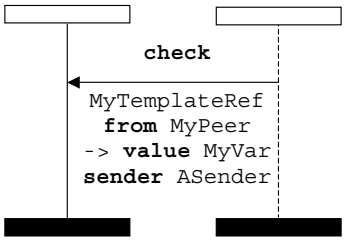
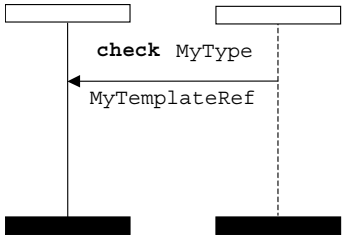
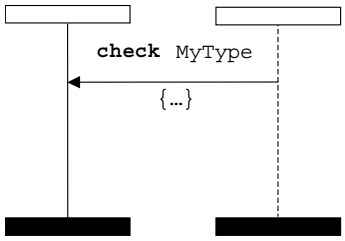
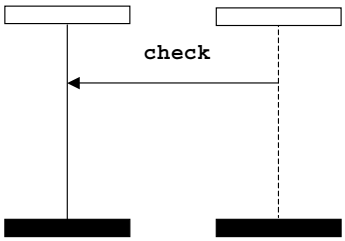
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
Handle response from a previous non-blocking call or independent from a call	**getreply**	  	Receive a response from a previous call. The reply has to match the conditions defined by the template reference. NOTE – The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

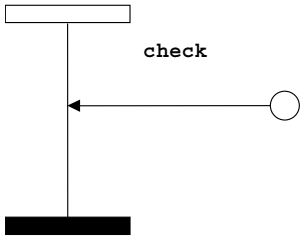
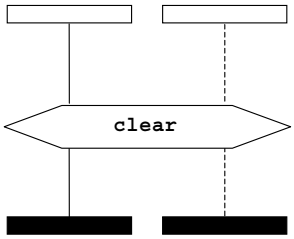
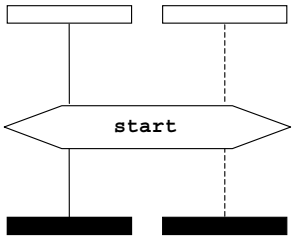
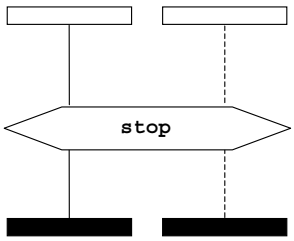
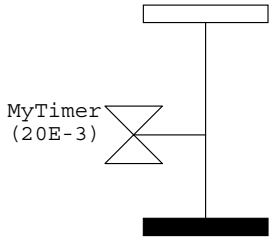
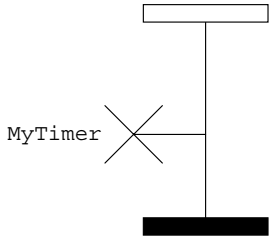
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
			Accept any response from any previous call. Optional **from**- and **sender**-directives may be present to identify the sender of the reply or to retrieve the identifier of the peer entity.
			Accept any response from any previous call at any port. The reply to be received from any port may be restricted by means referring to templates or by using inline templates. 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.
Catch exception from a previous blocking call	**catch**		Catch an exception from a previous call. The exception has to match the conditions defined by the template reference. NOTE – The type information is part of the template definition. The (optional) **from**-directive denotes that the sender of the exception shall be identified by variable `MyPeer`. The (optional) value-directive assigns the value of the exception to variable `MyVal`. The (optional) **sender**-directive retrieves the identifier of the sender and stores it in variable `ASender`.

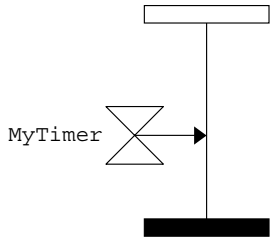
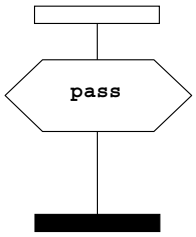
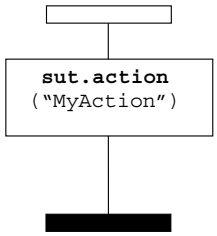
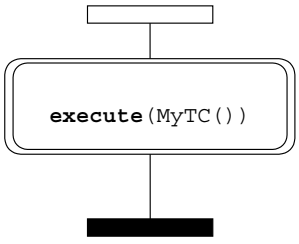
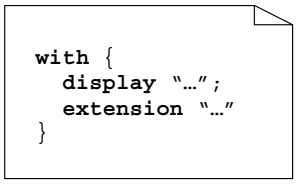
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
			Catch an exception from a previous call. The exception has to match the conditions defined by the exception type and the template reference. Optional **from**-, **value**-, and **sender**-directives may be present to identify the sender of the exception, to retrieve the exception value or to retrieve the identifier of the peer entity.
			Catch an exception from a previous call. The exception has to match the conditions defined by the exception type and the inline template definition. 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.
			Accept any exception from a blocking call. 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.

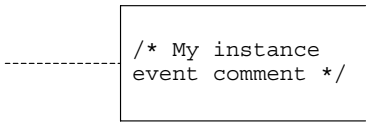
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
Catch exception from a previous non-blocking call or independent from a call	**catch**	  	Catch an exception from a previous call. The exception has to match the conditions defined by the template reference. NOTE – The type information is part of the template definition. The (optional) **from**-directive denotes that the sender of the exception shall be identified by variable `MyPeer`. The (optional) value-directive assigns the value of the exception to variable `MyVal`. The (optional) **sender**-directive retrieves the identifier of the sender and stores it in variable `ASender`. Catch an exception from a previous call. The exception has to match the conditions defined by the exception type and the template reference. 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. Catch an exception from a previous call. The exception has to match the conditions defined by the exception type and the inline template definition. 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.

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
		 	Catch any exception from any previous call. 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. Catch any exception from any previous call at any port. The exception to be received from any port may be restricted by means referring to templates or by using inline templates. 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.
Check (current) message/call received	**check**	  	with template, without type. with template, with type. without template, without type (any message from that port).

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
		 Can be used also in combination with getcall, getreply, and catch	with template, without type, without port (this message from that port).
Check current message, call, reply or exception	**check**	   	Check if a message with a value defined by a template reference has been received. The syntax follows the syntax for the reception of messages. NOTE – Check may also be used in combination with **getcall**, **getreply** and **catch**. Check if a message with a value defined by a template reference has been received. The syntax follows the syntax for the reception of messages. NOTE – Check may also be used in combination with **getcall**, **getreply** and **catch**. Check if a message with a value defined by an inline template definition has been received. The syntax follows the syntax for the reception of messages. NOTE – Check may also be used in combination with **getcall**, **getreply** and **catch**. Check if any message (no value and no type is specified) has been received. The syntax follows the syntax for the reception of messages. NOTE – Check may also be used in combination with **getcall**, **getreply** and **catch**.

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
			Check if any message (no value and no type is specified) has been received at any port. The syntax follows the syntax for the reception of messages. NOTE – Check may also be used in combination with **getcall**, **getreply** and **catch**.
Clear port	**clear**		The clear port statement is put into a condition symbol. The condition shall cover the instance of the port to be cleared only.
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**		

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
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.
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.

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Note
Comments			
Comments within text		`/* My several lines comment */` `// My single line comment`	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.

Annexe C (informative)

Mapping GFT to TTCN-3 Core Language

This annex defines an executable mapping from GFT/gr to the TTCN-3 core language [1]. The purpose behind this activity has been to aid the validation of the graphical grammar and language concepts. It can also be used to verify that the GFT to core and core to GFT mappings are bidirectional.

In order to provide both an abstract and readable implementation we have chosen to use the functional programming language, Standard ML of New Jersey (SML/NJ). SML/NJ is open source and is freely available from Bell Laboratories [http://cm.bell-labs.com/cm/cs/what/smlnj/].

C.1 Approach

The approach for the executable model has been to firstly represent both the core language and GFT grammars as separate SML data types (structures) `cn_type` and `gft_type` respectively. We then define a set of mapping functions that map the GFT data type onto the core data type for each GFT object (i.e., test case diagram, test step diagram, control diagram, and function diagram). The SML signatures for these mapping functions are as follows:

```
gft_testcase_to_cn : gft_type.TestcaseDiagram -> cn_type.TestcaseDef
gft_altstep_to_cn : gft_type.AltstepDiagram -> cn_type.AltstepDef
gft_function_to_cn : gft_type.FunctionDiagram -> cn_type.FunctionDef
gft_control_to_cn  : gft_type.ControlDiagram  -> cn_type.TTCN3Module
```

C.1.2 Overview of SML/NJ

This clause introduces some of the key SML concepts that have been used within the executable mappings.

C.1.2.1 Types and datatypes

SML is a strongly typed language and supports a number of base types, such as integers, strings, lists, etc. It also allows users to define pseudo types using the `type` keyword. For example, `type MyType = string` defines a new type called `MyType`. It also allows the definition of datatypes in which the user may define a set of constructors that give the user the option of defining a choice of sub types or datatypes. For example, `datatype MyDataType = C1 of string | C2 of int` represents the definition of a datatype called `MyDataType`, where C1 represents a constructor function that takes a string argument and returns `MyDataType`, and C2 is a constructor function that takes a string argument and returns `MyDataType`.

C.1.2.2 Functions

Within SML we use functions to define algorithms. The general structure of a function is of the following form: `fun name arguments = expression`, where the arguments may be contained within parentheses, e.g., `fun f (x,y)`, or in a curried fashions e.g., `fun f x y`. For this mapping we tend towards the use of arguments contained within parentheses.

In some cases, the body of the function (the `expression` part) can be defined in the following manner: `fun f = let … in … end`. This form is useful for simplifying the definition of the function by splitting it into sub expressions that are evaluated as needed. The `let…in` clause contains sub expressions, and `in…end` contains the expression representing the function body.

C.1.2.3 Pattern matching

SML has the ability to define patterns representing function arguments. For example, you will find that most of the function definitions contained within the mapping are of the following form.

```
fun f (C1 x) = x_toInt x
  | f (C2 y) = y_toInt 2
  | f (_)    = 3
```

where the function `f` takes a single datatype as an argument and returns an integer. In this example, the function defines three separate patterns `(C1 x)`, `(C2 y)` and `(_)`. If the first pattern is matched `(C1 x)` then the value of the variable `x` is passed to the function `x_toInt`, which in turn returns an integer. If the first argument isn't matched then the second is tested and so on. The last pattern `(_)` represents 'any value', meaning that if none of the previous patterns are matched then the expression associated with this pattern is evaluated. In this case an integer with the value 3 is returned.

C.1.2.4 Recursion

Another useful aspect of SML is its ability to represent recursive functions (e.g., functions that operate over a list of elements). For example, below we show a function `f` that takes a list and recursively applies the function `g` to each element within a list, returning an updated list.

```
fun f []     = []
  | f (h::t) = (g h)::f t
```

In this example you will notice that we use `[]` to denote an empty list, and `(h::t)` to denote the head and tail of a list. Where, `'::'` is a function that prepends a list element to a list of elements. In this case the function defines two patterns. The first pattern matches an empty list and returns an empty list. The second pattern matches a non-empty list. In doing so, it firstly binds the head of the list to the variable `h` and the tail to the variable `t`. Secondly it applies the function `g` to the head of the list and prepends to the result of recursively applying `f` on the tail (the remainer of the list).

C.2 Modelling GFT graphical grammar in SML

We have used the following rules to represent the GFT graphical grammar as an SML type:

- Each non-symbol production within GFT is represented as a SML type or datatype.
- Graphical attachments, as defined by the '*is attach to*' meta operator are modelled using the SML datatype `is_attached_to`. This type allows the two ends of graphical attachment to be represented by a pair of string labels. Note that the mapping functions do not need to know about all attachments, therefore the use of `is_attached_to` is not exhaustive.
- The SML `set` type is use to model the meta type '*set'*, where appropriate.
- We only define GFT types to a level where a core type is referenced.
- The SML `option` type is used to denote optional productions. For example, a production `'[Type]'` would model as `Type option`, where the SML constructors `SOME` and `NONE` are used to represent the value of an optional type.

Below is an example of how we model the GFT production PortOperationArea as a SML type.

```
type PortOperationArea  =   (* Condition contains *)
                            PortOperationText *
                            (* is_attached_to InstanceConditionBeginSymbol *)
                            (* is attached to InstanceConditionEndSymbol *)
                            (* is attached to PortConditionBeginSymbol set *)
```

```
(* is attached to PortConditionEndSymbol set *)
is_attached_to *  (* InstancePortOperationArea *)
is_attached_to    (* PortConditionArea *)
```

C.2.1 SML modules

The types and functions needed for the executable mapping are grouped into SML modules, where each module defines a signature and a structure. The signature part defines what types and functions (including signatures) are visible outside the structure in which they are defined. The structure part contains the types and function definitions. For example, below we show an example of a module containing a signature and structure:

```
signature MyStructSig =
sig
type MyStructType
val f : int -> int (* A function f that takes an integer argument and returns an integer *)
end
structure MyStruct : MyStructSig =
struct
    type MyStructType = int
    fun f x = x+1
end
```

Within the executable GFT to core mapping we define the following SML modules: `gft_type.sml,CN_type.sml, gfttocn.sml,given_functions.sml`.

C.2.2 Function naming and references

Each mapping function follows the name of the GFT type which it is mapping. For example, `p_TimeoutArea` is the mapping function for mapping the TimeoutArea production.

References to types and functions outside of the current scope are prefixed with the name of the structure containing their definition. For example, a reference to the SML type within the cn_type structure is prefixed with `cn_type`, e.g., `cn_type.ConstDef`.

C.2.3 Given functions

To simplify the definition of mapping functions we assume that the following functions are given:

```
val extract_TextLayer_from_testcase       : gft_type.TestcaseBodyArea -> gft_type.TextLayer
val extract_comments_from_TextLayer       : gft_type.TextLayer -> cn_type.TTCN3Comments list
val extract_with_statement_from_TextLayer : gft_type.TextLayer -> cn_type.WithStatement option
val get_attached_SendArea                 : gft_type.is_attached_to * gft_type.ConnectorLayer
                                                    -> gft_type.SendArea
val get_attached_ReceiveArea              : gft_type.is_attached_to * gft_type.ConnectorLayer
                                                    -> gft_type.ReceiveArea
val get_attached_NonBlockingCallArea      : gft_type.is_attached_to * gft_type.ConnectorLayer
                                                    -> gft_type.NonBlockingCallArea
val get_attached_GetcallArea              : gft_type.is_attached_to * gft_type.ConnectorLayer
                                                    -> gft_type.GetcallArea
val get_attached_ReplyArea                : gft_type.is_attached_to * gft_type.ConnectorLayer
                                                    -> gft_type.ReplyArea
val get_attached_GetreplyOutsideCallArea  : gft_type.is_attached_to * gft_type.ConnectorLayer
                                                    -> gft_type.GetreplyOutsideCallArea
val get_attached_RaiseArea                : gft_type.is_attached_to * gft_type.ConnectorLayer
                                                    -> gft_type.RaiseArea
val get_attached_CatchOutsideCallArea     : gft_type.is_attached_to * gft_type.ConnectorLayer
                                            -> gft_type.CatchOutsideCallArea
val get_attached_TriggerArea              : gft_type.is_attached_to * gft_type.ConnectorLayer
                                            -> gft_type.TriggerArea
val get_attached_CheckArea                : gft_type.is_attached_to * gft_type.ConnectorLayer
                                            -> gft_type.CheckArea
val get_attached_starttimer               : gft_type.is_attached_to * gft_type.InstanceEventLayer
                                            -> cn_type.TimerRef
val get_attached_PortOperationArea        : gft_type.is_attached_to * gft_type.ConnectorLayer
                                            -> gft_type.PortOperationArea
val get_attached_InvocationArea           : gft_type.is_attached_to * gft_type.ConnectorLayer
                                            -> gft_type.InvocationArea
val get_attached_IfArea                   : gft_type.InstanceInlineExpressionBeginSymbol *
                                            gft_type.ConnectorLayer
                                            -> gft_type.IfArea
val get_attached_ForArea                  : gft_type.InstanceInlineExpressionBeginSymbol *
                                            gft_type.ConnectorLayer
                                            -> gft_type.ForArea
val get_attached_WhileArea                : gft_type.InstanceInlineExpressionBeginSymbol *
                                            gft_type.ConnectorLayer
                                            -> gft_type.WhileArea
```

```
val get_attached_DoWhileArea              : gft_type.InstanceInlineExpressionBeginSymbol *
                                            gft_type.ConnectorLayer
                                            -> gft_type.DoWhileArea
val get_attached_AltArea                  : gft_type.InstanceInlineExpressionBeginSymbol *
                                            gft_type.ConnectorLayer
                                            -> gft_type.AltArea
val get_attached_InterleaveArea           : gft_type.InstanceInlineExpressionBeginSymbol *
                                            gft_type.ConnectorLayer
                                            -> gft_type.InterleaveArea
val get_number_of_port_instances          : gft_type.InstanceLayer -> int
val get_attached_portcondition_set        : gft_type.is_attached_to * gft_type.PortEventLayer
                                            -> gft_type.is_attached_to list
val get_attached_port                     : gft_type.is_attached_to * gft_type.InstanceLayer
                                            -> cn_type.Port
val GuardOpLayer_to_ConnectorLayer        : gft_type.GuardOpLayer -> gft_type.ConnectorLayer
val UnguardOpLayer_to_ConnectorLayer      : gft_type.UnguardedOpLayer -> gft_type.ConnectorLayer
```

C.2.4 GFT and core SML types

The types are contained in gft_type.sml and CNType.sml. They are not given here, but can be provided on request from ETSI MTS.

C.2.5 GFT to CN mapping functions

```
signature GFTTOCN =
sig
    val gft_testcase_to_cn : gft_type.TestcaseDiagram -> cn_type.TestcaseDef
end (* end of signature *)

structure gfttocn : GFTTOCN =
struct

open gft_type

(*****************************************************************************
    p_TextLayer : gft_type.TextLayer -> (cn_type.TTCN3Comment list,cn_type.WithStatement)
*****************************************************************************)
fun p_TextLayer TextLayer = (given_functions.extract_comments_from_TextLayer TextLayer,
                             case (given_functions.extract_with_statement_from_TextLayer TextLayer)
                                of NONE   => []
                                |  SOME x => x)

(*****************************************************************************
    p_SendArea : SendArea * InstanceLayer
                        -> cn_type.FunctionStatementOrDef
*****************************************************************************)
fun p_SendArea  ((Type,
                  (DerivedDef, TemplateBody, ToClause),
                  InstanceSendEventArea,
                  PortInMsgEventArea),p)
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.SENDSTATEMENT(
                                        given_functions.get_attached_port(PortInMsgEventArea,p),
                                        ((Type,DerivedDef,TemplateBody),ToClause)
                                )
                        )
                )
(* end of p_SendArea *)

(*****************************************************************************
    p_ReceiveArea : ReceiveArea * InstanceLayer
                        -> cn_type.FunctionStatementOrDef
*****************************************************************************)
fun p_ReceiveArea  ((Type,
                     (SOME (DerivedDef,TemplateBody),FromClause,PortRedirect),
                     InstanceReceiveEventArea,
                     PortOutMsgEventArea),p)
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.RECEIVESTATEMENT(
                                        given_functions.get_attached_port(PortOutMsgEventArea,p),
                                        (SOME(Type,DerivedDef,TemplateBody),FromClause,PortRedirect)
                                )
                        )
                )
 | p_ReceiveArea   ((Type,
```

```
                    (NONE,FromClause,PortRedirect),
                    InstanceReceiveEventArea,
                    PortOutMsgEventArea),p)
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.RECEIVESTATEMENT(
                                        given_functions.get_attached_port(PortOutMsgEventArea,p),
                                        (NONE,FromClause,PortRedirect)
                                )
                        )
                )
(* end of p_ReceiveArea *)

(*************************************************************************
    p_NonBlockingCallArea : NonBlockingCallArea * Timervalue option * InstanceLayer
                        -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_NonBlockingCallArea
                  ((Signature,
                  (DerivedDef, TemplateBody, ToClause),
                  InstanceCallEventArea,
                  PortCallInEventArea),TimerValue,p)
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.CALLSTATEMENT(
                                        given_functions.get_attached_port(PortCallInEventArea,p),
                                        (((Signature, DerivedDef, TemplateBody),NONE),ToClause),
                                        NONE (* No statement list *)
                                )
                        )
                )
(* end of p_NonBlockingCallArea *)

(*************************************************************************
    p_GetCallArea : GetCallArea * InstanceLayer
                        -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_GetCallArea ((Signature,
                  (SOME (DerivedDef,TemplateBody),
                   FromClause,
                   PortRedirectWithParam),
                   InstanceGetCallEventArea,
                   PortGetCallOutEventArea),p)
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                               cn_type.GETCALLSTATEMENT(
                                    given_functions.get_attached_port(PortGetCallOutEventArea,p),
                                        (
                                        SOME (Signature,DerivedDef,TemplateBody),
                                        FromClause,
                                        PortRedirectWithParam
                                        )
                                )
                        )
                )
 |  p_GetCallArea ((Signature,
                  (NONE,
                   FromClause,
                   PortRedirectWithParam),
                   InstanceGetCallEventArea,
                   PortGetCallOutEventArea),p)
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.GETCALLSTATEMENT(
                                       given_functions.get_attached_port(PortGetCallOutEventArea,p),
                                        (
                                        NONE,
                                        FromClause,
                                        PortRedirectWithParam
                                        )
                                )
                        )
                )
(* end of p_GetCallArea *)

(*************************************************************************
```

```
    p_ReplyArea : ReplyArea * InstanceLayer
                        -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_ReplyArea  ((Signature,
                   (DerivedDef, TemplateBody, ReplyValue,ToClause),
                   InstanceReplyEventArea,
                   PortReplyInEventArea),p)
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.REPLYSTATEMENT(
                                        given_functions.get_attached_port(PortReplyInEventArea,p),
                                        ((Signature,DerivedDef,TemplateBody),ReplyValue,ToClause)
                                )
                        )
                )
(* end of p_ReplyArea *)

(************************************************************************
    p_GetreplyOutsideCallArea : GetreplyOutsideCallArea * InstanceLayer
                        -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_GetreplyOutsideCallArea ((Signature,
                                (SOME (DerivedDef,TemplateBody,ValueMatchSpec),
                                 FromClause,
                                 PortRedirectWithParam),
                                 InstanceGetreplyEventArea,
                                 PortGetreplyOutEventArea),p)
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.GETREPLYSTATEMENT(
                                      given_functions.get_attached_port(PortGetreplyOutEventArea,p),
                                        (
                                        SOME ((Signature,DerivedDef,TemplateBody),ValueMatchSpec),
                                        FromClause,
                                        PortRedirectWithParam
                                        )
                                )
                        )
                )
 |  p_GetreplyOutsideCallArea ((Signature,
                                (NONE,
                                 FromClause,
                                 PortRedirectWithParam),
                                 InstanceGetreplyEventArea,
                                 PortGetreplyOutEventArea),p)
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.GETREPLYSTATEMENT(
                                      given_functions.get_attached_port(PortGetreplyOutEventArea,p),
                                        (
                                        NONE,
                                        FromClause,
                                        PortRedirectWithParam
                                        )
                                )
                        )
                )
(* end of p_GetreplyOutsideCallArea *)

(************************************************************************
    p_RaiseArea : RaiseArea * InstanceLayer
                        -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_RaiseArea ((Signature,
                  (TemplateInstance,ToClause),
                   InstanceRaiseEventArea,
                   PortGetreplyoutEventArea),p)
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.RAISESTATEMENT(
                                      given_functions.get_attached_port(PortGetreplyoutEventArea,p),
                                        (
                                        Signature,
                                        TemplateInstance,
                                        ToClause
                                        )
```

```
                                )
                        )
                )
(* end of p_RaiseArea *)

(*****************************************************************************
    p_CatchOutsideCallArea : CatchOutsideCallArea * InstanceLayer
                        -> cn_type.FunctionStatementOrDef
*****************************************************************************)
fun p_CatchOutsideCallArea ((SOME Signature,
                        (SOME TemplateInstance,FromClause,PortRedirect),
                         InstanceRaiseEventArea,
                         PortCatchoutEventArea),p)
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.CATCHSTATEMENT(
                                        given_functions.get_attached_port(PortCatchoutEventArea,p),
                                        (
                                        SOME (cn_type.CATCHTEMPLATE (Signature,TemplateInstance)),
                                        FromClause,
                                        PortRedirect
                                        )
                                )
                        )
                )
    | p_CatchOutsideCallArea ((NONE,
                          (NONE,FromClause,PortRedirect),
                         InstanceRaiseEventArea,
                         PortCatchoutEventArea),p)
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.CATCHSTATEMENT(
                                        given_functions.get_attached_port(PortCatchoutEventArea,p),
                                        (
                                        NONE,
                                        FromClause,
                                        PortRedirect
                                        )
                                )
                        )
                )
    | p_CatchOutsideCallArea ((_,
                          (_,FromClause,PortRedirect),
                         InstanceRaiseEventArea,
                         PortCatchoutEventArea),p)
        =
        (print ("Error: Catch must have both a signature and type declared. \n");
                OS.Process.exit OS.Process.failure)
(* end of p_CatchOutsideCallArea *)

(*****************************************************************************
    p_TriggerArea : TriggerArea * InstanceLayer
                        -> cn_type.FunctionStatementOrDef
*****************************************************************************)
fun p_TriggerArea ((Type,
                   (SOME (DerivedDef,TemplateBody),
                   FromClause,
                   PortRedirect),
                   InstanceTriggerEventArea,
                   PortOutMsgEventArea),p)
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.TRIGGERSTATEMENT(
                                        given_functions.get_attached_port(PortOutMsgEventArea,p),
                                        (
                                        SOME (Type,DerivedDef,TemplateBody),
                                        FromClause,
                                        PortRedirect
                                        )
                                )
                        )
                )
 |  p_TriggerArea ((Signature,
                  (NONE,
                   FromClause,
                   PortRedirect),
                   InstanceTriggerEventArea,
```

```
                   PortOutMsgEventArea),p)
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.TRIGGERSTATEMENT(
                                        given_functions.get_attached_port(PortOutMsgEventArea,p),
                                        (
                                        NONE,
                                        FromClause,
                                        PortRedirect
                                        )
                                )
                        )
                )
(* End of p_TriggerArea *)

(*****************************************************************************
    p_CheckOpInformation : CheckOpInformation * CheckData -> cn_type.PortCheckOp
*****************************************************************************)
fun p_CheckOpInformation (NONE,FROMCLAUSEONLY(FromClause,SenderSpec))   =
        SOME (cn_type.CHECKPARAMETER1 (FromClause,SenderSpec))

 |  p_CheckOpInformation (SOME (TYPE Type),
             DERIVEDEF (SOME (DerivedDef,TemplateBody,ValueMatchSpec),
                        FromClause,PortRedirect))      =
               SOME (cn_type.PORTRECEIVEOP
               (
                   SOME (SOME Type,DerivedDef,TemplateBody),
                   FromClause,PortRedirect
               )
           )
 |  p_CheckOpInformation (SOME (TYPE Type),
             DERIVEDEF (NONE,FromClause,PortRedirect))     =
               SOME (cn_type.PORTRECEIVEOP
               (
                   NONE,
                   FromClause,PortRedirect
               )
           )
 |  p_CheckOpInformation (SOME (GETCALLOPKEYWORD Signature),
             DERIVEDEFWPARAM (SOME (DerivedDef,TemplateBody,ValueMatchSpec),
                              FromClause,PortRedirectWithParam)) =
               SOME (cn_type.PORTGETCALLOP
               (
                   SOME (Signature,DerivedDef,TemplateBody),
                   FromClause,PortRedirectWithParam
               )
           )
 |  p_CheckOpInformation (SOME (GETCALLOPKEYWORD Signature),
             DERIVEDEFWPARAM (NONE,
                              FromClause,PortRedirectWithParam)) =
               SOME (cn_type.PORTGETCALLOP
               (
               NONE,FromClause,PortRedirectWithParam
               )
           )
 |  p_CheckOpInformation (SOME (GETREPLYOPKEYWORD Signature),
             DERIVEDEFWPARAM (SOME (DerivedDef,TemplateBody,ValueMatchSpec),
                              FromClause,PortRedirectWithParam)) =
               SOME (cn_type.PORTGETREPLYOP
               (
               SOME ((Signature,DerivedDef,TemplateBody),ValueMatchSpec),
               FromClause,PortRedirectWithParam
               )
           )
 |  p_CheckOpInformation (SOME (GETREPLYOPKEYWORD Signature),
             DERIVEDEFWPARAM (NONE,
                              FromClause,PortRedirectWithParam)) =
               SOME (cn_type.PORTGETREPLYOP
               (
               NONE,FromClause,PortRedirectWithParam
               )
           )
 |  p_CheckOpInformation (SOME (CATCHOPKEYWORD (Signature,Type)),
             DERIVEDEF (SOME (DerivedDef,TemplateBody,ValueMatchSpec),
                        FromClause,PortRedirect)) =
               SOME (cn_type.PORTCATCHOP
               (
               SOME (cn_type.CATCHTEMPLATE (Signature,(Type,DerivedDef,TemplateBody))),
               FromClause,PortRedirect
```

```
                )
            )
 |  p_CheckOpInformation (SOME (CATCHOPKEYWORD (Signature,Type)),
             DERIVEDEF (NONE,
                        FromClause,PortRedirect)) =
               SOME (cn_type.PORTCATCHOP
               (
               NONE,FromClause,PortRedirect
               )
            )
(* end of p_CheckOpInformation *)

(*************************************************************************
    p_CheckArea : CheckArea * InstanceLayer
                        -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_CheckArea ((CheckOpInformation,
                  CheckData,
                  InstanceReceiveEventArea,
                  PortOutMsgEventArea),p)
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.CHECKSTATEMENT(
                                        given_functions.get_attached_port(PortOutMsgEventArea,p),
                                        (
                                        p_CheckOpInformation (CheckOpInformation,CheckData)
                                        )
                                )
                        )
                )
(* end of p_CheckArea *)

(*************************************************************************
    p_InstanceSendEventArea : InstanceSendEventArea *
                  (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                        -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstanceSendEventArea (InstanceSendEventArea, (_,InstanceLayer,ConnectorLayer,_)) =
        let
                val SendArea =
            given_functions.get_attached_SendArea (InstanceSendEventArea, ConnectorLayer)
        in
                p_SendArea (SendArea,InstanceLayer)
        end
(* end of p_InstanceSendEventArea *)

(*************************************************************************
    p_InstanceReceiveEventArea : InstanceReceiveEventArea *
                    (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                        -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstanceReceiveEventArea (InstanceReceiveEventArea, (_,InstanceLayer,ConnectorLayer,_)) =
        let
                val ReceiveArea =
        given_functions.get_attached_ReceiveArea (InstanceReceiveEventArea, ConnectorLayer)
        in
                p_ReceiveArea (ReceiveArea,InstanceLayer)
        end
(* end of p_InstanceReceiveEventArea *)

(*************************************************************************
    p_InstanceCallEventArea : InstanceCallEventArea*
                  (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                        -> cn_type.FunctionStatementOrDef
*************************************************************************)

fun p_InstanceCallEventArea (INSTANCENONBLOCKINGCALLEVENTAREA InstanceNonBlockingCallEventArea,
                             (_,InstanceLayer,ConnectorLayer,_)) =
        let
                val NonBlockingCallArea =
 given_functions.get_attached_NonBlockingCallArea (InstanceNonBlockingCallEventArea, ConnectorLayer)
        in
                p_NonBlockingCallArea (NonBlockingCallArea,NONE,InstanceLayer)
        end

(* end of p_InstanceCallEventArea *)

(*************************************************************************
    p_InstanceGetCallEventArea : InstanceGetCallEventArea *
```

```
                    (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                        -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstanceGetCallEventArea (InstanceReceiveEventArea, (_,InstanceLayer,ConnectorLayer,_)) =
        let
                val GetCallArea =
        given_functions.get_attached_GetcallArea (InstanceReceiveEventArea, ConnectorLayer)
        in
                p_GetCallArea (GetCallArea,InstanceLayer)
        end

(* end of p_InstanceGetCallEventArea *)

(************************************************************************
    p_InstanceReplyEventArea : InstanceReplyEventArea*
                        (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                        -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstanceReplyEventArea (InstanceReplyEventArea, (_,InstanceLayer,ConnectorLayer,_)) =
        let
                val ReplyArea =
        given_functions.get_attached_ReplyArea (InstanceReplyEventArea, ConnectorLayer)
        in
                p_ReplyArea (ReplyArea,InstanceLayer)
        end
(* end of p_InstanceReplyEventArea *)

(************************************************************************
    p_InstanceGetreplyOutsideCallEventArea :
                InstanceGetreplyOutsideCallEventArea *
                    (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                    -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstanceGetreplyOutsideCallEventArea (InstanceGetreplyOutsideCallEventArea,
(_,InstanceLayer,ConnectorLayer,_)) =
        let
                val GetreplyOutsideCallArea =
                    given_functions.get_attached_GetreplyOutsideCallArea
                    (InstanceGetreplyOutsideCallEventArea, ConnectorLayer)
        in
                p_GetreplyOutsideCallArea (GetreplyOutsideCallArea,InstanceLayer)
        end
(* end of p_InstanceGetCallEventArea *)

(************************************************************************
    p_InstanceRaiseEventArea : InstanceRaiseEventArea *
                        (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                        -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstanceRaiseEventArea (InstanceRaiseEventArea, (_,InstanceLayer,ConnectorLayer,_)) =
        let
                val RaiseArea =
            given_functions.get_attached_RaiseArea (InstanceRaiseEventArea, ConnectorLayer)
        in
                p_RaiseArea (RaiseArea,InstanceLayer)
        end
(* end of p_InstanceRaiseEventArea *)

(************************************************************************
    p_InstanceCatchOutsideCallEventArea : InstanceCatchOutsideCallEventArea *
                        (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                        -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstanceCatchOutsideCallEventArea (InstanceCatchOutsideCallEventArea,
(_,InstanceLayer,ConnectorLayer,_)) =
        let
                val CatchOutsideCallArea =
            given_functions.get_attached_CatchOutsideCallArea (InstanceCatchOutsideCallEventArea,
ConnectorLayer)
        in
                p_CatchOutsideCallArea (CatchOutsideCallArea,InstanceLayer)
        end
(* end of p_InstanceCatchOutsideCallEventArea *)

(************************************************************************
    p_InstanceCatchTimeoutWithinCallEventArea :
        InstanceCatchTimeoutWithinCallEventArea *
                        (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                        -> cn_type.FunctionStatementOrDef
*************************************************************************)
```

```
fun p_InstanceCatchTimeoutWithinCallEventArea (InstanceCatchTimeoutWithinCallEventArea,
(_,InstanceLayer,ConnectorLayer,_)) =
        let
        in
        cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.CATCHSTATEMENT(
                        given_functions.get_attached_port(PortCatchoutEventArea,p),
                                        (
                                        SOME (cn_type.CATCHTIMEOUT),
                                        NONE,
                                        NONE
                                        )
                                )
                        )
                )
        end
(* end of p_InstanceCatchTimeoutWithinCallEventArea *)

(*************************************************************************
    p_InstanceTriggerEventArea : (InstanceTriggerEventArea *
                        (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                        -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstanceTriggerEventArea (InstanceTriggerEventArea, (_,InstanceLayer,ConnectorLayer,_)) =
        let
                val TriggerArea =
given_functions.get_attached_TriggerArea (InstanceTriggerEventArea, ConnectorLayer)
        in
                p_TriggerArea (TriggerArea,InstanceLayer)
        end

(* end of p_InstanceTriggerEventArea*)

(*************************************************************************
    p_InstanceCheckEventArea : (InstanceCheckEventArea *
                        (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                        -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstanceCheckEventArea (InstanceCheckEventArea, (_,InstanceLayer,ConnectorLayer,_)) =
        let
                val CheckArea =
given_functions.get_attached_CheckArea (InstanceCheckEventArea, ConnectorLayer)
        in
                p_CheckArea (CheckArea,InstanceLayer)
        end

(* end of p_InstanceCheckEventArea*)

(*************************************************************************
    p_TimeoutArea1 : TimeoutArea1
                -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_TimeoutArea1 (NONE)                               = cn_type.FUNCTIONSTATEMENT (
                                cn_type.TIMERSTATEMENTS (
                                    cn_type.TIMEOUTSTATEMENT (cn_type.ANYTIMER)
                                    )
                                )
 |  p_TimeoutArea1 (SOME TimerRef)                      = cn_type.FUNCTIONSTATEMENT (
                                cn_type.TIMERSTATEMENTS (
                                    cn_type.TIMEOUTSTATEMENT (cn_type.TIMEOUTREF TimerRef)
                                    )
                                )
(* end of p_TimeoutArea1 *)

(*************************************************************************
    p_TimeoutArea2 : is_attached_to * InstanceEventLayer
                -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_TimeoutArea2 (is_attached_to,InstanceEventLayer) = cn_type.FUNCTIONSTATEMENT (
                                cn_type.TIMERSTATEMENTS (
                                    cn_type.TIMEOUTSTATEMENT (
                                        cn_type.TIMEOUTREF
                                        (given_functions.get_attached_starttimer
                                            (is_attached_to,InstanceEventLayer))
                                        )
                                    )
                                )
(* end of p_TimeoutArea2 *)
```

```
(*************************************************************************
    p_InstanceTimeoutArea : InstanceTimeoutArea
                -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstanceTimeoutArea (TIMEOUTAREA1 t,_)                              = p_TimeoutArea1 t
 |  p_InstanceTimeoutArea (TIMEOUTAREA2 t,(InstanceEventLayer,_,_,_))     = p_TimeoutArea2
(t,InstanceEventLayer)
(* end of p_InstanceTimeoutArea *)

(*************************************************************************
    p_TimerStopArea1 : TimerStopArea1
                -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_TimerStopArea1 (NONE)                         =
                        cn_type.FUNCTIONSTATEMENT (
                                cn_type.TIMERSTATEMENTS (
                                    cn_type.STOPTIMERSTATEMENT (cn_type.ALLTIMERS)
                                    )
                                )
 |  p_TimerStopArea1 (SOME TimerRef)                =
                        cn_type.FUNCTIONSTATEMENT (
                                cn_type.TIMERSTATEMENTS (
                                    cn_type.STOPTIMERSTATEMENT (cn_type.STOPTIMERREF TimerRef)
                                    )
                                )
(* end of p_TimerStopArea1 *)

(*************************************************************************
    p_TimerStopArea2 : TimerStopArea2 * InstanceEventLayer
                -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_TimerStopArea2 (is_attached_to,InstanceEventLayer) =
                        cn_type.FUNCTIONSTATEMENT (
                              cn_type.TIMERSTATEMENTS (
                                    cn_type.STOPTIMERSTATEMENT (
                                      cn_type.STOPTIMERREF
                                      (given_functions.get_attached_starttimer
                                        (is_attached_to,InstanceEventLayer))
                                    )
                                )
                            )
(* end of p_TimerStopArea2 *)

(*************************************************************************
    p_InstanceTimerStopArea : InstanceTimerStopArea *
                      (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                        -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstanceTimerStopArea (TIMERSTOPAREA1 t,_)                            = p_TimerStopArea1 t
 |  p_InstanceTimerStopArea (TIMERSTOPAREA2 t,(InstanceEventLayer,_,_,_))   = p_TimerStopArea2
(t,InstanceEventLayer)
(* end of p_InstanceTimerStopArea *)

(*************************************************************************
    p_InstanceTimerStartArea : InstanceTimerStartArea
                -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstanceTimerStartArea (TimerRef,TimerValue,_) = cn_type.FUNCTIONSTATEMENT (
                                cn_type.TIMERSTATEMENTS (
                                    cn_type.STARTTIMERSTATEMENT (TimerRef,TimerValue)
                                    )
                                )
(* end of p_InstanceTimerStartArea *)

(*************************************************************************
    p_InstanceTimerEventArea : InstanceTimerEventArea *
                      (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                        -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstanceTimerEventArea (INSTANCETIMERSTARTAREA t,_) = p_InstanceTimerStartArea t
 |  p_InstanceTimerEventArea (INSTANCETIMERSTOPAREA t,p)  = p_InstanceTimerStopArea (t,p)
 |  p_InstanceTimerEventArea (INSTANCETIMEOUTAREA t,p)    = p_InstanceTimeoutArea (t,p)
(* end of p_InstanceTimerEventArea *)

(*************************************************************************
    p_FoundMessage : FoundMessage -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_FoundMessage (Type,(SOME (DerivedDef,TemplateBody),FromClause,PortRedirect))
        =
                cn_type.FUNCTIONSTATEMENT(
```

```
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.RECEIVESTATEMENT(

"any",(SOME(Type,DerivedDef,TemplateBody),FromClause,PortRedirect)
                                )
                        )
                )
 | p_FoundMessage (Type,(NONE,FromClause,PortRedirect))
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.RECEIVESTATEMENT(
                                        "any",(NONE,FromClause,PortRedirect)
                                )
                        )
                )

(*************************************************************************
    p_FoundTrigger : FoundTrigger -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_FoundTrigger (Type,(SOME (DerivedDef,TemplateBody),FromClause,PortRedirect))
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.TRIGGERSTATEMENT(
                                        "any",
                                        (
                                        SOME (Type,DerivedDef,TemplateBody),
                                        FromClause,
                                        PortRedirect
                                        )
                                )
                        )
                )
 |  p_FoundTrigger (Signature,(NONE,FromClause,PortRedirect))
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.TRIGGERSTATEMENT(
                                        "any",(NONE,FromClause,PortRedirect)
                                )
                        )
                )
(* end of p_FoundTrigger *)

(*************************************************************************
    p_FoundGetCall : FoundGetCall -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_FoundGetCall (Signature,(SOME (DerivedDef,TemplateBody),FromClause,PortRedirectWithParam))
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.GETCALLSTATEMENT(
                                        "any",
                                        (
                                        SOME (Signature,DerivedDef,TemplateBody),
                                        FromClause,
                                        PortRedirectWithParam
                                        )
                                )
                        )
                )
 |  p_FoundGetCall (Signature,(NONE,FromClause,PortRedirectWithParam))
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.GETCALLSTATEMENT(
                                        "any",(NONE,FromClause,PortRedirectWithParam)
                                )
                        )
                )
(* end of p_FoundGetCall *)

(*************************************************************************
    p_FoundGetReply : FoundGetReply -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_FoundGetReply (Signature,
                     (SOME (DerivedDef,TemplateBody,ValueMatchSpec),
                     FromClause,
                     PortRedirectWithParam))
```

```
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.GETREPLYSTATEMENT(
                                        "any",
                                        (
                                        SOME ((Signature,DerivedDef,TemplateBody),ValueMatchSpec),
                                        FromClause,
                                        PortRedirectWithParam
                                        )
                                )
                        )
                )
 |  p_FoundGetReply (Signature,(NONE,FromClause,PortRedirectWithParam))
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.GETREPLYSTATEMENT(
                                        "any",(NONE,FromClause,PortRedirectWithParam)
                                )
                        )
                )
(* end of p_FoundGetReply *)
(*************************************************************************
    p_FoundCatch : FoundCatch -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_FoundCatch (SOME Signature,(SOME TemplateInstance,FromClause,PortRedirect))
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.CATCHSTATEMENT(
                                        "any",
                                        (
                                        SOME (cn_type.CATCHTEMPLATE (Signature,TemplateInstance)),
                                        FromClause,
                                        PortRedirect
                                        )
                                )
                        )
                )
    | p_FoundCatch (NONE,(NONE,FromClause,PortRedirect))
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.CATCHSTATEMENT(
                                        "any",(NONE,FromClause,PortRedirect)
                                )
                        )
                )
    | p_FoundCatch (_,(_,FromClause,PortRedirect))
        =
        (print ("Error: Catch must have both a signature and type declared. \n");
                OS.Process.exit OS.Process.failure)
(* end of p_FoundCatch *)

(*************************************************************************
    p_FoundCheck : FoundCheck -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_FoundCheck ((CheckOpInformation,CheckData):FoundCheck)
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.CHECKSTATEMENT(
                                        "any",(p_CheckOpInformation (CheckOpInformation,CheckData))
                                )
                        )
                )
(* end of p_FoundCheck *)

(*************************************************************************
    p_FoundEvent : FoundEvent -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_FoundEvent (FOUNDMESSAGE FoundMessage)    = p_FoundMessage FoundMessage
 |  p_FoundEvent (FOUNDTRIGGER FoundTrigger)    = p_FoundTrigger FoundTrigger
 |  p_FoundEvent (FOUNDGETCALL FoundGetCall)    = p_FoundGetCall FoundGetCall
 |  p_FoundEvent (FOUNDGETREPLY FoundGetReply)  = p_FoundGetReply FoundGetReply
 |  p_FoundEvent (FOUNDCATCH FoundCatch)        = p_FoundCatch FoundCatch
 |  p_FoundEvent (FOUNDCHECK FoundCheck)        = p_FoundCheck FoundCheck
(* end of p_FoundEvent *)
```

```
(*****************************************************************************
    p_InstanceFoundEventArea : InstanceFoundEventArea -> cn_type.FunctionStatementOrDef
*****************************************************************************)
fun p_InstanceFoundEventArea FoundEvent = p_FoundEvent FoundEvent
(* end of p_InstanceFoundEventArea *)

(*****************************************************************************
    p_ActionStatement : ActionStatement -> cn_type.FunctionStatementOrDef
*****************************************************************************)
fun p_ActionStatement (SUTSTATEMENTS SUTStatements)         = cn_type.FUNCTIONSTATEMENT (
                                  cn_type.SUTSTATEMENTS SUTStatements)
 |  p_ActionStatement (CONNECTSTATEMENT ConnectStatement)   = cn_type.FUNCTIONSTATEMENT (
                                  cn_type.CONFIGURATIONSTATEMENTS (
                                      cn_type.CONNECTSTATEMENT ConnectStatement))
 |  p_ActionStatement (MAPSTATEMENT MapStatement)       = cn_type.FUNCTIONSTATEMENT (
                                  cn_type.CONFIGURATIONSTATEMENTS (
                                      cn_type.MAPSTATEMENT MapStatement))
 |  p_ActionStatement (DISCONNECTSTATEMENT DisconnectStatement) = cn_type.FUNCTIONSTATEMENT (
                                  cn_type.CONFIGURATIONSTATEMENTS (
                                      cn_type.DISCONNECTSTATEMENT DisconnectStatement))
 |  p_ActionStatement (UNMAPSTATEMENT UnmapStatement)       = cn_type.FUNCTIONSTATEMENT (
                                  cn_type.CONFIGURATIONSTATEMENTS (
                                      cn_type.UNMAPSTATEMENT UnmapStatement))
 |  p_ActionStatement (CONSTDEF ConstDef)           = cn_type.FUNCTIONLOCALDEF ConstDef
 |  p_ActionStatement (VARINSTANCE VarInstance)         = cn_type.FUNCTIONLOCALINST (
                                  cn_type.VARINSTANCE VarInstance)
 |  p_ActionStatement (TIMERINSTANCE TimerInstance)     = cn_type.FUNCTIONLOCALINST (
                                  cn_type.TIMERINSTANCE TimerInstance)
 |  p_ActionStatement (ASSIGNMENT Assignment)           = cn_type.FUNCTIONSTATEMENT (
                                  cn_type.BASICSTATEMENTS (
                                      cn_type.BASICASSIGNMENT Assignment))
 |  p_ActionStatement (LOGSTATEMENT LogStatement)       = cn_type.FUNCTIONSTATEMENT (
                                  cn_type.BASICSTATEMENTS (
                                      cn_type.LOGSTATEMENT LogStatement))
 |  p_ActionStatement (LOOPCONSTRUCT LoopConstruct)     = cn_type.FUNCTIONSTATEMENT (
                                  cn_type.BASICSTATEMENTS (
                                      cn_type.LOOPCONSTRUCT LoopConstruct))
 |  p_ActionStatement (CONDITIONALCONSTRUCT ConditionalConstruct)=cn_type.FUNCTIONSTATEMENT (
                                  cn_type.BASICSTATEMENTS (
                                      cn_type.CONDITIONALCONSTRUCT ConditionalConstruct))
(* end of p_ActionStatement *)

(*****************************************************************************
    p_InstanceActionArea : InstanceActionArea -> cn_type.FunctionStatementOrDef list
*****************************************************************************)
fun p_InstanceActionArea InstanceActionArea = map p_ActionStatement InstanceActionArea
(* end of p_InstanceActionArea *)

(*****************************************************************************
    p_InstanceLabellingArea : InstanceLabellingArea -> cn_type.FunctionStatementOrDef
*****************************************************************************)
fun p_InstanceLabellingArea LabelIdentifier = cn_type.FUNCTIONSTATEMENT (
                           cn_type.BEHAVIOURSTATEMENTS (
                               cn_type.LABELSTATEMENT LabelIdentifier
                               )
                           )
(* end of p_InstanceLabellingArea *)

(*****************************************************************************
    p_InstanceDoneArea : InstanceDoneArea -> cn_type.FunctionStatementOrDef
*****************************************************************************)
fun p_InstanceDoneArea DoneStatement =
        cn_type.FUNCTIONSTATEMENT (
            cn_type.CONFIGURATIONSTATEMENTS (
                cn_type.DONESTATEMENT DoneStatement))

(* end of p_InstanceDoneArea *)

(*****************************************************************************
    p_SetVerdictArea : SetVerdictArea -> cn_type.FunctionStatementOrDef
*****************************************************************************)
fun p_SetVerdictArea (SETVERDICTKEYWORD SingleExpression) =
                cn_type.FUNCTIONSTATEMENT (
            cn_type.VERDICTSTATEMENTS SingleExpression)
 |  p_SetVerdictArea (PASSKEYWORD)      =
                cn_type.FUNCTIONSTATEMENT (
            cn_type.VERDICTSTATEMENTS "pass")
 |  p_SetVerdictArea (FAILKEYWORD)      =
                cn_type.FUNCTIONSTATEMENT (
            cn_type.VERDICTSTATEMENTS "fail")
```

```
 |  p_SetVerdictArea (INCONCKEYWORD)    =
               cn_type.FUNCTIONSTATEMENT (
           cn_type.VERDICTSTATEMENTS "inconc")
 |  p_SetVerdictArea (NONEKEYWORD)      =
               cn_type.FUNCTIONSTATEMENT (
           cn_type.VERDICTSTATEMENTS "none")
(* end of p_SetVerdictArea *)

(*************************************************************************
    p_InstanceSetVerdictArea : InstanceSetVerdictArea -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstanceSetVerdictArea SetVerdictArea = p_SetVerdictArea SetVerdictArea
(* end of p_InstanceSetVerdictArea *)

(*************************************************************************
    p_PortOperationArea : PortOperationArea -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_PortOperationArea ((PortOperationText,InstancePortOperationArea,PortConditionArea),
                         (PortEventLayer,InstanceLayer)) =
        let
                val number_of_ports   = given_functions.get_number_of_port_instances InstanceLayer
                val attached_port_list= given_functions.get_attached_portcondition_set
(PortConditionArea,PortEventLayer)

                fun map_operation (PortOrAll,STARTKEYWORD)   =
                        cn_type.STARTSTATEMENT (PortOrAll,cn_type.STARTKEYWORD)
                 |  map_operation (PortOrAll,STOPKEYWORD)    =
                        cn_type.STOPSTATEMENT (PortOrAll,cn_type.STOPKEYWORD)
                 |  map_operation (PortOrAll,CLEAROPKEYWORD) =
                        cn_type.CLEARSTATEMENT (PortOrAll,cn_type.CLEAROPKEYWORD)
        in
                (* If condition symbol covers all ports then perform operation on all ports *)
                if number_of_ports = (List.length attached_port_list) then
                        cn_type.FUNCTIONSTATEMENT (
                                cn_type.COMMUNICATIONSTATEMENTS (
                                        map_operation ("all",PortOperationText)
                                )
                        )

                (* A port operation can either be connected to one or all ports *)
                else if (List.length attached_port_list) > 1 then
                        (print ("Error: Port operation is attached to incorrect number of
ports.\n");
                         OS.Process.exit OS.Process.failure)

                (* Operation to be performed on a single port *)
                else if (List.length attached_port_list) = 1 then
                        cn_type.FUNCTIONSTATEMENT (
                                cn_type.COMMUNICATIONSTATEMENTS (
                                       map_operation (hd attached_port_list,PortOperationText)
                                )
                        )

                (* Condition symbol must be attached to at least on port instance *)
                else
                        (print ("Error: Port operation is not attached to any port instance(s).\n");
                         OS.Process.exit OS.Process.failure)
        end
(* end of p_PortOperationArea *)

(*************************************************************************
    p_InstancePortOperationArea : InstancePortOperationArea *

(InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                    -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstancePortOperationArea
(InstancePortOperationArea,(_,InstanceLayer,ConnectorLayer,PortEventLayer)) =
        let
                val PortOperationArea =
            given_functions.get_attached_PortOperationArea (InstancePortOperationArea,
ConnectorLayer)
        in
                p_PortOperationArea (PortOperationArea,(PortEventLayer,InstanceLayer))
        end
(* end of p_InstancePortOperationArea *)

(*************************************************************************
    p_InstanceConditionArea : InstanceConditionArea *
                                (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
```

```
                                    -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstanceConditionArea ((INSTANCEDONEAREA InstanceDoneArea),_) =
                        p_InstanceDoneArea InstanceDoneArea
 |  p_InstanceConditionArea ((INSTANCESETVERDICTAREA InstanceSetVerdictArea),_) =
                        p_InstanceSetVerdictArea InstanceSetVerdictArea
 |  p_InstanceConditionArea ((INSTANCEPORTOPERATIONAREA InstancePortOperationArea),p) =
                        p_InstancePortOperationArea  (InstancePortOperationArea,p)
(* end of p_InstanceConditionArea *)

(************************************************************************
    p_InvocationArea : InvocationArea -> cn_type.FunctionStatementOrDef
************************************************************************)
fun p_InvocationArea (INVOCATIONFUNCTIONINSTANCE FunctionInstance,_) = cn_type.FUNCTIONSTATEMENT (
                                cn_type.BEHAVIOURSTATEMENTS (
                                    cn_type.FUNCTIONINSTANCE FunctionInstance))
 |  p_InvocationArea (INVOCATIONALTSTEPINSTANCE AltstepInstance,_) = cn_type.FUNCTIONSTATEMENT (
                                cn_type.BEHAVIOURSTATEMENTS (
                                    cn_type.ALTSTEPINSTANCE AltstepInstance))
 |  p_InvocationArea (INVOCATIONCONSTDEF ConstDef,_)              = cn_type.FUNCTIONLOCALDEF ConstDef
 |  p_InvocationArea (INVOCATIONVARINSTANCE VarInstance,_)       = cn_type.FUNCTIONLOCALINST (
                                cn_type.VARINSTANCE VarInstance)
 |  p_InvocationArea (INVOCATIONASSIGNMENT Assignment,_)             = cn_type.FUNCTIONSTATEMENT (
                                cn_type.BASICSTATEMENTS (
                                    cn_type.BASICASSIGNMENT Assignment))
(* end of p_InvocationArea *)

(************************************************************************
    p_InstanceInvocationArea : InstanceInvocationArea *
                                (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                    -> cn_type.FunctionStatementOrDef
************************************************************************)
fun p_InstanceInvocationArea (InstanceInvocationArea,(_,_,ConnectorLayer,_)) =
        let
                val InvocationArea =
            given_functions.get_attached_InvocationArea(InstanceInvocationArea, ConnectorLayer)
        in
                p_InvocationArea InvocationArea
        end
(* end of p_InstanceInvocationArea *)

(************************************************************************
    p_InstanceDefaultHandlingArea : InstanceInvocationArea
                                    -> cn_type.FunctionStatementOrDef
************************************************************************)
fun p_InstanceDefaultHandlingArea (DEACTIVATESTATEMENT DeactivateStatement)     =
cn_type.FUNCTIONSTATEMENT (
                    cn_type.BEHAVIOURSTATEMENTS (
                        cn_type.DEACTIVATESTATEMENT DeactivateStatement))
 |  p_InstanceDefaultHandlingArea (DEFAULTCONSTDEF ConstDef)            =
                    cn_type.FUNCTIONLOCALDEF ConstDef
 |  p_InstanceDefaultHandlingArea (DEFAULTVARINSTANCE VarInstance)      =
                    cn_type.FUNCTIONLOCALINST (
                        cn_type.VARINSTANCE VarInstance)
 |  p_InstanceDefaultHandlingArea (DEFAULTASSIGNMENT Assignment)        =
                    cn_type.FUNCTIONSTATEMENT (
                        cn_type.BASICSTATEMENTS (
                            cn_type.BASICASSIGNMENT Assignment))
(* end of p_InstanceDefaultHandlingArea *)

(************************************************************************
    p_InstanceComponentCreateArea : InstanceComponentCreateArea
                                    -> cn_type.FunctionStatementOrDef
************************************************************************)
fun p_InstanceComponentCreateArea (CREATIONCONSTDEF ConstDef)           =
                cn_type.FUNCTIONLOCALDEF ConstDef
 |  p_InstanceComponentCreateArea (CREATIONVARINSTANCE VarInstance)     =
                cn_type.FUNCTIONLOCALINST (
                        cn_type.VARINSTANCE VarInstance)
 |  p_InstanceComponentCreateArea (CREATIONASSIGNMENT Assignment)       =
                cn_type.FUNCTIONSTATEMENT (
                        cn_type.BASICSTATEMENTS (
                                cn_type.BASICASSIGNMENT Assignment))
(* end of p_InstanceComponentCreateArea *)

(************************************************************************
    p_InstanceComponentStartArea : InstanceComponentStartArea
                                    -> cn_type.FunctionStatementOrDef
************************************************************************)
fun p_InstanceComponentStartArea StartTCStatement =
```

```
                    cn_type.FUNCTIONSTATEMENT (
                            cn_type.CONFIGURATIONSTATEMENTS (
                                cn_type.STARTTCSTATEMENT StartTCStatement))
(* end of p_InstanceComponentStartArea *)

(*************************************************************************
    p_InstanceEventArea : InstanceEventArea *
(InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                -> cn_type.FunctionStatementOrDef list
*************************************************************************)
fun p_InstanceEventArea (INSTANCESENDEVENTAREA (InstanceSendEventArea, comments),p) =
                [p_InstanceSendEventArea (InstanceSendEventArea,p)]
 |  p_InstanceEventArea (INSTANCERECEIVEEVENTAREA (InstanceReceiveEventArea, comments),p) =
                [p_InstanceReceiveEventArea (InstanceReceiveEventArea,p)]
 |  p_InstanceEventArea (INSTANCECALLEVENTAREA (InstanceCallEventArea, comments),p) =
                [p_InstanceCallEventArea (InstanceCallEventArea,p)]
 |  p_InstanceEventArea (INSTANCEGETCALLEVENTAREA (InstanceGetCallEventArea, comments),p) =
                [p_InstanceGetCallEventArea (InstanceGetCallEventArea,p)]
 |  p_InstanceEventArea (INSTANCEREPLYEVENTAREA (InstanceReplyEventArea, comments),p) =
                [p_InstanceReplyEventArea (InstanceReplyEventArea,p)]
 |  p_InstanceEventArea (INSTANCEGETREPLYOUTSIDECALLEVENTAREA (InstanceGetreplyOutsideCallEventArea,
comments),p) =
                [p_InstanceGetreplyOutsideCallEventArea (InstanceGetreplyOutsideCallEventArea,p)]
 |  p_InstanceEventArea (INSTANCERAISEEVENTAREA (InstanceRaiseEventArea, comments),p) =
                [p_InstanceRaiseEventArea (InstanceRaiseEventArea,p)]
 |  p_InstanceEventArea (INSTANCECATCHTIMEOUTWITHINCALLEVENTAREA
(InstanceCatchTimeoutWithinCallEventArea, comments),p) =
                [p_InstanceCatchTimeoutWithinCallEventArea
(InstanceCatchTimeoutWithinCallEventArea,p)]
 |  p_InstanceEventArea (INSTANCECATCHOUTSIDECALLEVENTAREA (InstanceCatchOutsideCallEventArea,
comments),p) =
                [p_InstanceCatchOutsideCallEventArea (InstanceCatchOutsideCallEventArea,p)]
 |  p_InstanceEventArea (INSTANCETRIGGEREVENTAREA (InstanceTriggerEventArea, comments),p) =
                [p_InstanceTriggerEventArea (InstanceTriggerEventArea,p)]
 |  p_InstanceEventArea (INSTANCECHECKEVENTAREA (InstanceCheckEventArea, comments),p) =
                [p_InstanceCheckEventArea (InstanceCheckEventArea,p)]
 |  p_InstanceEventArea (INSTANCEFOUNDEVENTAREA (InstanceFoundEventArea, comments),p) =
                [p_InstanceFoundEventArea InstanceFoundEventArea]
 |  p_InstanceEventArea (INSTANCETIMEREVENTAREA (InstanceTimerEventArea, comments),p) =
                [p_InstanceTimerEventArea (InstanceTimerEventArea,p)]
 |  p_InstanceEventArea (INSTANCEACTIONEVENTAREA (InstanceActionArea, comments),p) =
                 p_InstanceActionArea InstanceActionArea
 |  p_InstanceEventArea (INSTANCELABELLINGEVENTAREA (InstanceLabellingArea, comments),p) =
                [p_InstanceLabellingArea InstanceLabellingArea]
 |  p_InstanceEventArea (INSTANCECONDITIONEVENTAREA (InstanceConditionArea, comments),p) =
                [p_InstanceConditionArea (InstanceConditionArea,p)]
 |  p_InstanceEventArea (INSTANCEINVOCATIONEVENTAREA (InstanceInvocationArea, comments),p) =
                [p_InstanceInvocationArea (InstanceInvocationArea,p)]
 |  p_InstanceEventArea (INSTANCEDEFAULTHANDLINGAREA (InstanceDefaultHandlingArea, comments),p) =
                [p_InstanceDefaultHandlingArea InstanceDefaultHandlingArea]
 |  p_InstanceEventArea (INSTANCECOMPONENTCREATEAREA (InstanceComponentCreateArea, comments),p) =
                [p_InstanceComponentCreateArea InstanceComponentCreateArea]
 |  p_InstanceEventArea (INSTANCECOMPONENTSTARTAREA (InstanceComponentStartArea, comments),p) =
                [p_InstanceComponentStartArea InstanceComponentStartArea]
 |  p_InstanceEventArea (INSTANCEINLINEEXPRESSIONEVENTAREA (InstanceInlineExpressionEventArea,
comments),p) =
      let

        (*************************************************************************
    p_InstanceEventAreaList : InstanceEventArea list *
            (InstanceEventLayer, InstanceLayer,ConnectorLayer,PortEventLayer)
                                    -> cn_type.FunctionStatementOrDef list
    *************************************************************************)
    fun p_InstanceEventAreaList []  p                       = []
     |  p_InstanceEventAreaList (InstanceEventArea::t) p    =
p_InstanceEventArea (InstanceEventArea,p)@p_InstanceEventAreaList t p
    (* end of p_InstanceEventAreaList *)

        (*************************************************************************
    p_IfArea : IfArea * InstanceEventArea list * InstanceEventArea list option *
            (InstanceEventLayer, InstanceLayer,ConnectorLayer,PortEventLayer)
                                    -> cn_type.FunctionStatementOrDef
    *************************************************************************)
    fun p_IfArea (((BooleanExpression,OperandaArea1,NONE),_,SOME PortInlineExpressionBeginSymbol),
            IfInstanceEventAreaList,                                    (* If Event list *)
            NONE,                                                       (* Else Event list *)
           (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer))
        =
            cn_type.FUNCTIONSTATEMENT (
                cn_type.BASICSTATEMENTS (
```

```
                cn_type.CONDITIONALCONSTRUCT
                    (BooleanExpression,
                     p_InstanceEventAreaList IfInstanceEventAreaList
                     (InstanceEventLayer,InstanceLayer,OperandaArea1,PortEventLayer),
                     [],     (* No if else clauses - represented by nested inline expressions *)
                     NONE))) (* No else clause in this case *)
|  p_IfArea (((BooleanExpression,OperandaArea1,SOME OperandaArea2),_,SOME
            PortInlineExpressionBeginSymbol),                        (* IfArea *)
            IfInstanceEventAreaList,                                 (* If Event list *)
            SOME ElseInstanceEventAreaList,                          (* Else Event list *)
           (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer))
        cn_type.FUNCTIONSTATEMENT (
            cn_type.BASICSTATEMENTS (
                cn_type.CONDITIONALCONSTRUCT
                    (BooleanExpression,
                     p_InstanceEventAreaList IfInstanceEventAreaList
                    (InstanceEventLayer,InstanceLayer,OperandaArea1,PortEventLayer),
                     [],     (* No if else clauses - represented by nested inline expressions *)
                     SOME (p_InstanceEventAreaList ElseInstanceEventAreaList
                    (InstanceEventLayer,InstanceLayer,OperandaArea2,PortEventLayer))
                     )
                )
            )
(* end of p_IfArea *)

(*************************************************************************
p_InstanceIfArea : InstanceIfArea *
            (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                  -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstanceIfArea
            ((InstanceInlineExpressionBeginSymbol,IfInstanceEventArea,ElseInstanceEventArea),
              p as (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)) =
    let
        val IfArea = given_functions.get_attached_IfArea
                    (InstanceInlineExpressionBeginSymbol,ConnectorLayer)
    in
        p_IfArea (IfArea,IfInstanceEventArea,ElseInstanceEventArea,p)
    end
(* end of p_InstanceIfArea *)

(*************************************************************************
p_ForArea : ForArea * InstanceEventArea list  *
        (InstanceEventLayer, InstanceLayer,ConnectorLayer,PortEventLayer)
                                  -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_ForArea (((Initial, Final, Step, OperandaArea),_,_),  (* ForArea *)
         InstanceEventAreaList,                              (* Event list *)
        (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer))
    =
        cn_type.FUNCTIONSTATEMENT (
            cn_type.BASICSTATEMENTS (
                cn_type.LOOPCONSTRUCT (
                    cn_type.FORSTATEMENT (Initial, Final, Step,
                    p_InstanceEventAreaList InstanceEventAreaList
                    (InstanceEventLayer,InstanceLayer,OperandaArea,PortEventLayer)
                    )
                )
            )
        )
(* end of p_ForArea *)

(*************************************************************************
p_InstanceForArea : InstanceForArea *
            (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                  -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstanceForArea ((InstanceInlineExpressionBeginSymbol,InstanceEventArealist),
              p as (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)) =
    let
        val ForArea = given_functions.get_attached_ForArea
                     (InstanceInlineExpressionBeginSymbol,ConnectorLayer)
    in
        p_ForArea (ForArea,InstanceEventArealist,p)
    end
(* end of p_InstanceForArea *)

(*************************************************************************
p_WhileArea : WhileArea * InstanceEventArea list  *
        (InstanceEventLayer, InstanceLayer,ConnectorLayer,PortEventLayer)
```

```
                                -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_WhileArea (((BooleanExpression, OperandaArea),_,_),   (* WhileArea *)
         InstanceEventAreaList,                             (* Event list *)
        (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer))
    =
        cn_type.FUNCTIONSTATEMENT (
            cn_type.BASICSTATEMENTS (
                cn_type.LOOPCONSTRUCT (
                    cn_type.WHILESTATEMENT (BooleanExpression,
                    p_InstanceEventAreaList InstanceEventAreaList
                    (InstanceEventLayer,InstanceLayer,OperandaArea,PortEventLayer)
                    )
                )
            )
        )
(* end of p_WhileArea *)

(************************************************************************
p_InstanceWhileArea : InstanceWhileArea *
            (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstanceWhileArea ((InstanceInlineExpressionBeginSymbol,InstanceEventArealist),
              p as (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)) =
    let
        val WhileArea = given_functions.get_attached_WhileArea
                        (InstanceInlineExpressionBeginSymbol,ConnectorLayer)
    in
        p_WhileArea (WhileArea,InstanceEventArealist,p)
    end
(* end of p_InstanceWhileArea *)


    (************************************************************************
p_DoWhileArea : DoWhileArea * InstanceEventArea list  *
        (InstanceEventLayer, InstanceLayer,ConnectorLayer,PortEventLayer)
                                -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_DoWhileArea (((BooleanExpression, OperandaArea),_,_),
         InstanceEventAreaList,                             (* Event list *)
        (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer))
    =
        cn_type.FUNCTIONSTATEMENT (
            cn_type.BASICSTATEMENTS (
                cn_type.LOOPCONSTRUCT (
                    cn_type.WHILESTATEMENT (BooleanExpression,
                    p_InstanceEventAreaList InstanceEventAreaList
                    (InstanceEventLayer,InstanceLayer,OperandaArea,PortEventLayer)
                    )
                )
            )
        )
(* end of p_DoWhileArea *)

(************************************************************************
p_InstanceDoWhileArea : InstanceDoWhileArea *
            (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstanceDoWhileArea ((InstanceInlineExpressionBeginSymbol,InstanceEventArealist),
              p as (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)) =
    let
        val DoWhileArea = given_functions.get_attached_DoWhileArea
                        (InstanceInlineExpressionBeginSymbol,ConnectorLayer)
    in
        p_DoWhileArea (DoWhileArea,InstanceEventArealist,p)
    end
(* end of p_InstanceDoWhileArea *)

(************************************************************************
p_BooleanExpressionConditionArea : BooleanExpressionConditionArea
                    -> cn_type.AltGuardChar
*************************************************************************)
fun p_BooleanExpressionConditionArea BooleanExpression = BooleanExpression
(* end of p_BooleanExpressionConditionArea *)

(************************************************************************
p_InstanceBooleanExpressionConditionArea :InstanceBooleanExpressionConditionArea
                    -> cn_type.AltGuardChar
```

```
    ***************************************************************************)
    fun p_InstanceBooleanExpressionConditionArea x = p_BooleanExpressionConditionArea x
    (* end of p_InstanceBooleanExpressionConditionArea *)

    (***************************************************************************
    p_InstanceGuardOpArea : InstanceGuardOpArea -> cn_type.GuardOp
    ***************************************************************************)
    fun p_InstanceGuardOpArea ((INSTANCEGUARDRECEIVEEVENTAREA x),p) =
        let
           val (cn_type.FUNCTIONSTATEMENT(
                cn_type.COMMUNICATIONSTATEMENTS(
                    cn_type.RECEIVESTATEMENT ReceiveStatement
                    )
                )
            ) = p_InstanceReceiveEventArea (x,p)
        in
            cn_type.GUARDRECEIVESTATEMENT ReceiveStatement
        end
    (* end of p_InstanceGuardOpArea *)

    (***************************************************************************
    p_GuardArea : GuardArea  *
            (InstanceEventLayer * InstanceLayer* (GuardOpLayer * ConnectorLayer) * PortEventLayer)
                -> cn_type.GuardStatement
    ***************************************************************************)
    fun p_GuardArea (INSTANCEGUARDOPAREA (InstanceGuardOpArea,
                          InstanceEventAreaList),
                 (InstanceEventLayer,InstanceLayer,(GuardOpLayer,ConnectorLayer),PortEventLayer))
    =
       cn_type.GUARDOPSTATEMENT (
              p_InstanceGuardOpArea (InstanceGuardOpArea,(InstanceEventLayer,InstanceLayer,
                           given_functions.GuardOpLayer_to_ConnectorLayer
                            GuardOpLayer,PortEventLayer)
                          ),
              p_InstanceEventAreaList InstanceEventAreaList
                       (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
           )
    (* end of p_GuardArea *)

    (***************************************************************************
    p_InstanceGuardArea : InstanceGuardArea *
        (InstanceEventLayer * InstanceLayer* (GuardOpLayer * ConnectorLayer) * PortEventLayer)
                -> cn_type.GuardStatement
    ***************************************************************************)
    fun p_InstanceGuardArea  ((SOME InstanceBooleanExpressionConditionArea,GuardArea),p) =
          (SOME (p_InstanceBooleanExpressionConditionArea InstanceBooleanExpressionConditionArea),
                  p_GuardArea (GuardArea,p))
          | p_InstanceGuardArea  ((NONE,GuardArea),p) =
          (NONE,
           p_GuardArea (GuardArea,p))
    (* end of p_InstanceGuardArea *)

    (***************************************************************************
    p_InstanceElseGuardArea : InstanceElseGuardArea *
        (InstanceEventLayer * InstanceLayer* (GuardOpLayer * ConnectorLayer) * PortEventLayer)
                -> cn_type.GuardStatement
    ***************************************************************************)
    fun p_InstanceElseGuardArea (SOME (ElseConditionArea,InstanceEventAreaList),
                    (InstanceEventLayer,InstanceLayer,SOME (_,ConnectorLayer),PortEventLayer))
                =
                     SOME (p_InstanceEventAreaList InstanceEventAreaList
                    (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer))
     |  p_InstanceElseGuardArea (NONE,_)
                =    NONE
    (* end of p_InstanceElseGuardArea *)

    (***************************************************************************
    p_InstanceGuardAreaList : InstanceGuardArea list ->
    (InstanceEventLayer * InstanceLayer * (GuardOpLayer * ConnectorLayer) list * PortEventLayer)
                        -> cn_type.GuardStatement list
    ***************************************************************************)
    fun p_InstanceGuardAreaList ([],_)= []
     |  p_InstanceGuardAreaList
(InstanceGuardArea::t,(InstanceEventLayer,InstanceLayer,h::t',PortEventLayer)) =
            p_InstanceGuardArea
(InstanceGuardArea,(InstanceEventLayer,InstanceLayer,h,PortEventLayer))::
            p_InstanceGuardAreaList (t,(InstanceEventLayer,InstanceLayer,t',PortEventLayer))
     (* end of p_InstanceGuardAreaList *)

    (***************************************************************************
```

```
    p_AltArea : AltArea * InstanceGuardArea * InstanceGuardArea list * InstanceElseGuardArea option*
            (InstanceEventLayer, InstanceLayer,ConnectorLayer,PortEventLayer)
                                    -> cn_type.FunctionStatementOrDef
    ************************************************************************)
    fun p_AltArea (((GuardedOperandArea,GuardedOperandAreaList,ElseOperandArea),_,_),
             InstanceGuardArea,InstanceGuardAreaList,InstanceElseGuardArea,  (* Event lists *)
            (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer))
        =
            cn_type.FUNCTIONSTATEMENT (
                cn_type.BEHAVIOURSTATEMENTS (
                    cn_type.ALTCONSTRUCT (
                        p_InstanceGuardArea
(InstanceGuardArea,(InstanceEventLayer,InstanceLayer,GuardedOperandArea,PortEventLayer))
                            ::p_InstanceGuardAreaList
(InstanceGuardAreaList,(InstanceEventLayer,InstanceLayer,GuardedOperandAreaList,PortEventLayer)),
                        p_InstanceElseGuardArea
(InstanceElseGuardArea,(InstanceEventLayer,InstanceLayer,ElseOperandArea,PortEventLayer))
                    )
                )
            )
    (* end of p_AltArea *)

    (************************************************************************
    p_InstanceAltArea : InstanceAltArea *
                (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                    -> cn_type.FunctionStatementOrDef
    ************************************************************************)
    fun p_InstanceAltArea
((InstanceInlineExpressionBeginSymbol,InstanceGuardArea,InstanceGuardArealist,InstanceElseGuardArea)
,
                  p as (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)) =
        let
            val AltArea = given_functions.get_attached_AltArea
                         (InstanceInlineExpressionBeginSymbol,ConnectorLayer)
        in
            p_AltArea (AltArea,InstanceGuardArea,InstanceGuardArealist,InstanceElseGuardArea,p)
        end
    (* end of p_InstanceAltArea *)

    (************************************************************************
    p_InstanceGuardOpArea : InstanceGuardOpArea *
            (InstanceEventLayer * InstanceLayer* (GuardOpLayer * ConnectorLayer) * PortEventLayer)
                -> cn_type.InterleavedGuard
    ************************************************************************)
    fun p_InstanceGuardOpArea ((INSTANCEGUARDRECEIVEEVENTAREA x),p) =
        let
           val (cn_type.FUNCTIONSTATEMENT(
                cn_type.COMMUNICATIONSTATEMENTS(
                    cn_type.RECEIVESTATEMENT ReceiveStatement
                    )
                )
            ) = p_InstanceReceiveEventArea (x,p)
        in
            cn_type.GUARDRECEIVESTATEMENT ReceiveStatement
        end
    (* p_InstanceGuardOpArea *)

    (************************************************************************
    p_InstanceInterleaveGuardArea : InstanceInterleaveGuardArea *
                (InstanceEventLayer * InstanceLayer* (GuardOpLayer * ConnectorLayer) *
PortEventLayer)
                -> cn_type.InterleavedGuardElement
    ************************************************************************)
    fun p_InstanceInterleaveGuardArea  ((InstanceGuardOpArea,InstanceEventAreaList),
        (InstanceEventLayer,InstanceLayer,(UnguardOpLayer,ConnectorLayer),PortEventLayer)) =
            (p_InstanceGuardOpArea (InstanceGuardOpArea,(InstanceEventLayer,InstanceLayer,
                            given_functions.UnguardOpLayer_to_ConnectorLayer
                                        UnguardOpLayer,PortEventLayer)),
             p_InstanceEventAreaList InstanceEventAreaList
                            (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer))
    (* end of p_InstanceInterleaveGuardArea *)

     (************************************************************************
    p_InstanceInterleaveGuardAreaList : InstanceInterleaveGuardArea list ->
        (InstanceEventLayer * InstanceLayer * (GuardOpLayer * ConnectorLayer) list * PortEventLayer)
                        -> cn_type.InterleavedGuardElement list
    ************************************************************************)
    fun p_InstanceInterleaveGuardAreaList ([],_)= []
     |  p_InstanceInterleaveGuardAreaList
(InstanceInterleaveGuardArea::t,(InstanceEventLayer,InstanceLayer,h::t',PortEventLayer)) =
```

```
            p_InstanceInterleaveGuardArea
(InstanceInterleaveGuardArea,(InstanceEventLayer,InstanceLayer,h,PortEventLayer))::
            p_InstanceInterleaveGuardAreaList
(t,(InstanceEventLayer,InstanceLayer,t',PortEventLayer))
     (* end of p_InstanceInterleaveGuardAreaList *)

    (*************************************************************************
    p_InterleaveArea : InterleaveArea * InstanceGuardArea * InstanceGuardArea list *
InstanceElseGuardArea option*
            (InstanceEventLayer, InstanceLayer,ConnectorLayer,PortEventLayer)
                                    -> cn_type.FunctionStatementOrDef
    *************************************************************************)
    fun p_InterleaveArea (((UnguardedOperandArea,UnguardedOperandAreaList),_,_):InterleaveArea,
        (* InterleaveArea *)
             InstanceInterleaveGuardArea,InstanceInterleaveGuardAreaList,
            (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer))
        =
            cn_type.FUNCTIONSTATEMENT (
                cn_type.BEHAVIOURSTATEMENTS (
                    cn_type.INTERLEAVEDCONSTRUCT (
                        p_InstanceInterleaveGuardArea
(InstanceInterleaveGuardArea,(InstanceEventLayer,InstanceLayer,UnguardedOperandArea,PortEventLayer))
                            ::p_InstanceInterleaveGuardAreaList
(InstanceInterleaveGuardAreaList,(InstanceEventLayer,InstanceLayer,UnguardedOperandAreaList,PortEven
tLayer))
                    )
                )
            )
    (* end of p_InterleaveArea *)

    (*************************************************************************
    p_InstanceInterleaveArea : InstanceInterleaveArea *
                (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                    -> cn_type.FunctionStatementOrDef
    *************************************************************************)
    fun p_InstanceInterleaveArea
((InstanceInlineExpressionBeginSymbol,InstanceInterleaveGuardArea,InstanceInterleaveGuardAreaList)
    :InstanceInterleaveArea,
                 p as (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)) =
        let
            val InterleaveArea = given_functions.get_attached_InterleaveArea
                            (InstanceInlineExpressionBeginSymbol,ConnectorLayer)
        in
            p_InterleaveArea
                    (InterleaveArea,InstanceInterleaveGuardArea,InstanceInterleaveGuardAreaList,p)
        end
    (* end of p_InstanceInterleaveArea *)

    (*************************************************************************
        p_InstanceInlineExpressionEventArea :
                InstanceInlineExpressionEventArea *

(InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                        -> cn_type.FunctionStatementOrDef
    *************************************************************************)
    fun p_InstanceInlineExpressionEventArea ((INSTANCEIFAREA InstanceIfArea),p)         =
p_InstanceIfArea (InstanceIfArea,p)
     |  p_InstanceInlineExpressionEventArea ((INSTANCEFORAREA InstanceForArea),p)       =
p_InstanceForArea (InstanceForArea,p)
     |  p_InstanceInlineExpressionEventArea ((INSTANCEWHILEAREA InstanceWhileArea),p)   =
p_InstanceWhileArea (InstanceWhileArea,p)
     |  p_InstanceInlineExpressionEventArea ((INSTANCEDOWHILEAREA InstanceDoWhileArea),p)   =
p_InstanceDoWhileArea (InstanceDoWhileArea,p)
     |  p_InstanceInlineExpressionEventArea ((INSTANCEALTAREA InstanceAltArea),p)       =
p_InstanceAltArea (InstanceAltArea,p)
     |  p_InstanceInlineExpressionEventArea ((INSTANCEINTERLEAVEAREA InstanceInterleaveArea),p) =
p_InstanceInterleaveArea (InstanceInterleaveArea,p)

       in
              [p_InstanceInlineExpressionEventArea (InstanceInlineExpressionEventArea,p)]
       end
 (* end of p_InstanceEventArea *)

(*************************************************************************
p_InstanceEventLayer : InstanceEventArea list *
                              -> cn_type.StatementBlock
*************************************************************************)
fun p_InstanceEventLayer [] p                       = []
 |  p_InstanceEventLayer (InstanceEventArea::t) p   = p_InstanceEventArea (InstanceEventArea,p)@
                                p_InstanceEventLayer t p
```

```
(* end of p_InstanceEventLayer *)

(*****************************************************************************
    p_TestcaseBodyArea : TestcaseBodyArea -> cn_type.StatementBlock

        The InstanceEventlayer defines the order in which the events are placed
        upon an instance axis. Therefore, this function recursively iterates
        over the InstanceEventlayer list.

*****************************************************************************)
fun p_TestcaseBodyArea (InstanceLayer,TextLayer,InstanceEventLayer,PortEventLayer, ConnectorLayer) =
    p_InstanceEventLayer InstanceEventLayer
(InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
(* end of p_TestcaseBodyArea *)

(*****************************************************************************
    p_TestcaseHeading : (TestcaseHeading, TestcaseBodyArea) -> cn_type.TestcaseDef
*****************************************************************************)
fun p_TestcaseHeading ((TestcaseIdentifier,TestcaseFormalParList,ConfigSpec,LocalDefinitions),
                       TestcaseBodyArea) =
        let
                (* Place GFT testcase diagram in display attribute and          *)
                (* append those attributes that are defined on the diagram.    *)
                (* Note that Comments are currently ignored                     *)
                val TextLayer                                =
                        given_functions.extract_TextLayer_from_testcase TestcaseBodyArea
                val (testcase_comments,testcase_attributes) = p_TextLayer TextLayer
                val withstatement   = SOME (
                                        [
                                         [cn_type.DISPLAY "ETSI TTCN-3 GFT v1.0",
                                          cn_type.DISPLAY ("Testcase Diagram -
                                                           "^TestcaseIdentifier)
                                         ]
                                        ]@testcase_attributes
                                      )
        in
               (
               TestcaseIdentifier,
               TestcaseFormalParList,
               ConfigSpec,
               p_TestcaseBodyArea TestcaseBodyArea, (* Don't forget local definitions *)
               withstatement
               )
        end
(* end of p_TestcaseHeading *)

(*****************************************************************************
    gft_testcase_to_cn : testcaseDiagram -> cn_type.TestcaseDef
*****************************************************************************)
fun gft_testcase_to_cn (TestcaseHeading,TestcaseBodyArea) =
                p_TestcaseHeading (TestcaseHeading,TestcaseBodyArea)
(* end of gft_testcase_to_cn *)

end (* end of gfttocn structure *)
```

For further details, please refer to the file gfttocn.sml. It is not given here, but can be provided on request from ETSI MTS.

Annexe D (informative)

Mapping TTCN-3 core language to GFT

This annex defines an executable mapping from TTCN-3 core language [1] to GFT. The purpose behind this activity has been to aid the validation of the graphical grammar and language concepts. It can also be used to verify that the GFT to core and core to GFT mappings are bidirectional.

In order to provide both an abstract and readable implementation we have chosen to use the functional programming language, Standard ML of New Jersey (SML/NJ). SML/NJ is open source and is freely available from Bell Laboratories [http://cm.bell-labs.com/cm/cs/what/smlnj/].

D.1 Approach

The approach for the executable model has been to firstly represent both the core language and GFT grammars as separate SML data types (structures) `cn_type` and `gft_type` respectively. We then define a set of mapping functions that map the GFT data type onto the core data type for each GFT object (i.e., test case diagram, test step diagram, control diagram, and function diagram). The SML signatures for these mapping functions are as follows:

```
cn_testcase_to_gft : cn_type.TestcaseDef -> gft_type.TestcaseDiagram
cn_teststep_to_gft : cn_type.TeststepDef -> gft_type.TeststepDiagram
cn_function_to_gft : cn_type.FunctionDef -> gft_type.FunctionDiagram
cn_control_to_gft  : cn_type.TTCN3Module -> gft_type.ControlDiagram
```

D.1.2 Overview of SML/NJ

Please refer to C.1.2 for an overview on SML/NJ.

D.2 Modelling GFT graphical grammar in SML

D.2.1 SML modules

Please refer to C.2.1 for an overview on the used SML modules.

D.2.2 Function naming and references

Please refer to C.2.2 for an overview on function naming and references.

D.2.3 Given functions

Please refer to C.2.1 for an overview on given functions.

D.2.4 Core and GFT SML types

The types are contained in gft_type.sml and CNType.sml. They are not given here, but can be provided on request from ETSI MTS.

D.2.5 Core to GFT mapping functions

Since the core to GFT mapping functions are symmetric to the GFT to core mapping functions, only their signatures are given here.

```
(*****************************************************************************)
p_StartTimerStatement : StartTimerStatement
                        -> gft_type.InstanceTimerStartArea
(*****************************************************************************)

(*****************************************************************************)
p_StopTimerStatement : StopTimerStatement
                        -> gft_type.InstanceTimerStopArea
(*****************************************************************************)

(*****************************************************************************)
p_TimeoutStatement : TimeoutStatement
                        -> gft_type.InstanceTimeoutArea
(*****************************************************************************)

(*****************************************************************************)
p_TimerStatements : TimerStatements
                        -> gft_type.InstanceTimerEventArea
(*****************************************************************************)
```

```
*****************************************************************************
p_BehaviourStatements : BehaviourStatements -> gft_type.TestcaseBodyArea
*****************************************************************************

*****************************************************************************
p_TestcaseInstance : TestcaseInstance -> gft_type.TestcaseExecution
*****************************************************************************

*****************************************************************************
p_FunctionInstance : TestcaseIntance -> gft_type.Invocation
*****************************************************************************

*****************************************************************************
p_TeststepInstance : TeststepInstance -> gft_type.Invocation
*****************************************************************************

*****************************************************************************
p_ReturnStatements : ReturnStatements -> gft_type.ReturnArea
*****************************************************************************

*****************************************************************************
p_GuardStatement : GuardStatements -> gft_type.GuardArea
*****************************************************************************

*****************************************************************************
p_AltConstruct :  AltConstruct -> gft_type.AltArea
*****************************************************************************

*****************************************************************************
p_InterleavedConstruct: InterleavedConstruct -> gft_type.InterleaveArea
*****************************************************************************

*****************************************************************************
p_LabelStatement : LabelStatements -> gft_type.LabellingEventArea
*****************************************************************************

*****************************************************************************
p_GotoStatement : GotoStatements -> gft_type.GotoArea
*****************************************************************************

*****************************************************************************
p_WhileStatement : WhileStatements -> gft_type.WhileArea
*****************************************************************************

*****************************************************************************
p_DoWhileStatement : DoWhileStatements -> gft_type.DoWhileArea
*****************************************************************************

*****************************************************************************
p_ForStatement : ForStatements -> gft_type.ForArea
*****************************************************************************

*****************************************************************************
p_RepeatStatement : RepeatStatements -> gft_type.RepeatSymbol
*****************************************************************************

*****************************************************************************
p_DeactivateStatement : DeactivateStatements -> gft_type.DefaultHandling
*****************************************************************************

*****************************************************************************
p_ConnectStatements : ConnectStatements -> gft_type.ActionStatement
*****************************************************************************

*****************************************************************************
p_MapStatements : MapStatements -> gft_type.ActionStatement
*****************************************************************************

*****************************************************************************
p_DisconnectStatements : DisConnectStatements -> gft_type.ActionStatement
*****************************************************************************

*****************************************************************************
p_UnmapStatements : UnmapStatements -> gft_type.ActionStatement
*****************************************************************************

*****************************************************************************
p_DoneStatements : DoneStatements -> gft_type.ActionStatement
*****************************************************************************
```

```
*****************************************************************************
p_StartTCStatements : StartTCStatements -> gft_type.ActionStatement
*****************************************************************************

*****************************************************************************
p_StopTCStatements : StopTCStatements -> gft_type.ActionStatement
*****************************************************************************

*****************************************************************************
p_ComponentType : ComponentType -> gft_type.ComponentInstanceArea
*****************************************************************************

*****************************************************************************
p_ConfigurationStatements : ConfigurationStatements
                                                -> gft_type.ActionStatement
*****************************************************************************

*****************************************************************************
p_SUTStatements : SUTStatements
               -> gft_type.ActionStatement * gft_type.ControlActionStatement
*****************************************************************************

*****************************************************************************
p_VerdictStatements : VerdictSatement -> gft_type.SetVerdictArea
*****************************************************************************

*****************************************************************************
p_SendStatement : SendStatement * TestcaseFormalPar -> gft_type.SendArea
*****************************************************************************

*****************************************************************************
p_CallStatement : CallStatement * TestcaseFormalPar -> gft_type.CallArea
*****************************************************************************

*****************************************************************************
p_GetCallStatement : GetCallStatement * TestcaseFormalPar
                                                    -> gft_type.GetcallArea
*****************************************************************************

*****************************************************************************
p_ReplyStatement : ReplyStatement * TestcaseFormalPar -> gft_type.ReplyArea
*****************************************************************************

*****************************************************************************
p_RaiseStatement : RaiseStatement * TestcaseFormalPar -> gft_type.RaiseArea
*****************************************************************************

*****************************************************************************
p_ReceiveStatement : ReceiveStatement * TestcaseFormalPar
                                            -> gft_type.ReceiveArea
*****************************************************************************

*****************************************************************************
p_TriggerStatement : TriggerStatement * TestcaseFormalPar
                                            -> gft_type.TriggerArea
*****************************************************************************

*****************************************************************************
p_GetCallStatement : GetCallStatement * TestcaseFormalPar
                                                -> gft_type.GetCallArea
*****************************************************************************

*****************************************************************************
p_GetReplyStatement : TestcaseFormalPar * GetReplyStatement
        -> gft_type.GetReplyWithInCallArea * gft_type.GetReplyOutsideCallArea
*****************************************************************************

*****************************************************************************
p_CatchStatement : CatchStatement * TestcaseFormalPar
         -> gft_type.CatchWithInCallArea * gft_type.CatchOutsideCallArea
*****************************************************************************

*****************************************************************************
p_CheckStatement : CheckStatement * TestcaseFormalPar
                                            -> gft_type.CheckArea
*****************************************************************************

*****************************************************************************
p_ClearStatement : ClearStatement -> gft_type.ClearOpKeyWord
```

```
*******************************************************************************

*******************************************************************************
p_StartStatement : StartStatement -> gft_type.PortOperationText
*******************************************************************************

*******************************************************************************
p_StopStatement : StopStatement -> gft_type.PortOperationText
*******************************************************************************

*******************************************************************************
p_BasicStatement : BasicStatement
                          -> gft_type.ActionStatement
*******************************************************************************

*******************************************************************************
p_CommunicationStatements : CommunicationStatements
                 -> gft_type.ConnectorLayer * gft_type.InstanceLayer
*******************************************************************************

*******************************************************************************
p_ControlTimerStatements : ControlTimerStatements
                          -> gft_type.InstanceTimerEventArea
*******************************************************************************

*******************************************************************************
p_PortGetReplyOp : PortGetReplyOp * CommunicationStatement
                                                  -> gft_type.CheckData
*******************************************************************************

*******************************************************************************
p_CallBodyOps : CallBodyOps -> gft_type.CallBodyOpsLayer
*******************************************************************************

*******************************************************************************
p_ControlStatement : ControlStatement * FunctionLocalDef
                  -> gft_type.ControlEventArea * gft_type.ControlActionArea
*******************************************************************************

*******************************************************************************
p_ControlStatementOrDef : ControlStatementOrDef
                          -> gft_type.ControlActionStatement
*******************************************************************************

*******************************************************************************
p_ControlStatementOrDefList : ControlStatementOrDefList
                          -> gft_type.ControlActionArea
*******************************************************************************


*******************************************************************************
p_ModuleControlBody : ModuleControlBody
                          -> gft_type.ControlBodyArea
*******************************************************************************


*******************************************************************************
p_ModuleControlPart : ModuleControlPart
                          -> gft_type.ControlHeading
*******************************************************************************


*******************************************************************************
p_FunctionLocalInst: FunctionLocalInst ->  string
*******************************************************************************

*******************************************************************************
p_SingleWithAttrib : SingleWithAttrib -> String
*******************************************************************************

*******************************************************************************
p_WithStatement : WithStatement -> gft_type.TextLayer
*******************************************************************************

*******************************************************************************
p_TestcaseFormalPar: TestcaseFormalPar -> TestcaseFormalValuePar
*******************************************************************************

*******************************************************************************
p_FormalValuePar : FormalValuePar -> Type
```

```
*******************************************************************************

*******************************************************************************
p_FunctionStatement : FunctionStatement -> gft_type.TestcaseBodyArea
*******************************************************************************

*******************************************************************************
p_FunctionStatementOrDef : FunctionStatementOrDef
                                      ->  gft_type.LocalDefinition
*******************************************************************************

*******************************************************************************
p_StatementBlock : StatementBlock * WithStatement

         -> gft_type.TestcaseBodyArea
*******************************************************************************

*******************************************************************************
p_TeststepDef : TeststepDef -> gft_type.TeststepDiagram
*******************************************************************************

*******************************************************************************
p_FunctionDef : FunctionDef -> gft_type.FunctionDiagram
*******************************************************************************

*******************************************************************************
p_TTCN3Module : TTCN3Module -> gft_type.ControlDiagram
*******************************************************************************

*******************************************************************************
p_TestcaseDef : TestcaseDef -> gft_type.TestcaseDiagram
*******************************************************************************

*******************************************************************************
cn_teststep_to_gft : TeststepDef -> gft_type.TeststepDiagram
*******************************************************************************

*******************************************************************************
cn_function_to_gft : FunctionDef -> gft_type.FunctionDiagram
*******************************************************************************

*******************************************************************************
cn_control_to_gft : TTCN3Module -> gft_type.ControlDiagram
*******************************************************************************

*******************************************************************************
cn_testcase_to_gft : TestcaseDef -> gft_type.TestcaseDiagram
*******************************************************************************
```

For further details, please refer to the file cntogft.sml. It is not given here, but can be provided on request from ETSI MTS.

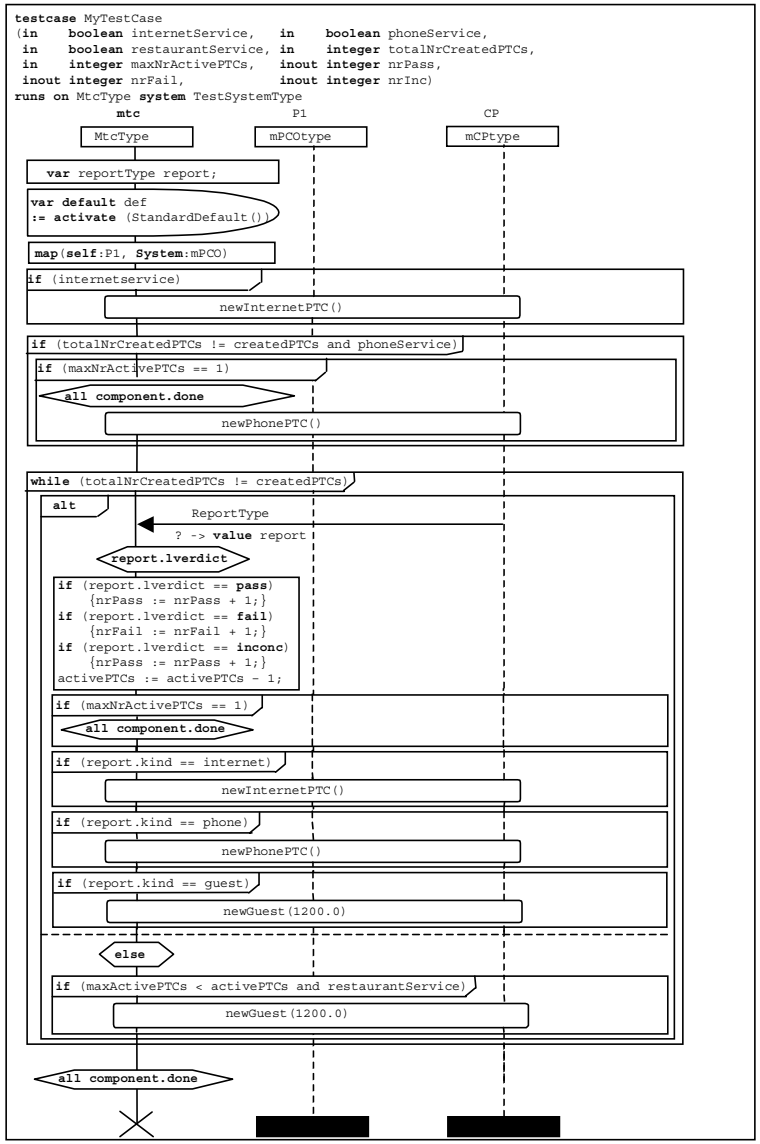
Annexe E (informative)

Examples

E.1 The Restaurant example: Figures E.1 to E.9

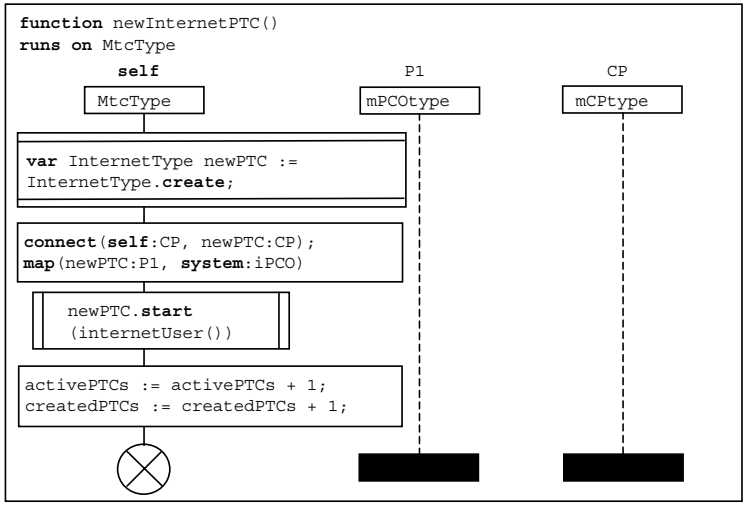
E.2 The INRES example: Figures E.10 to E.15

E.1 The restaurant example

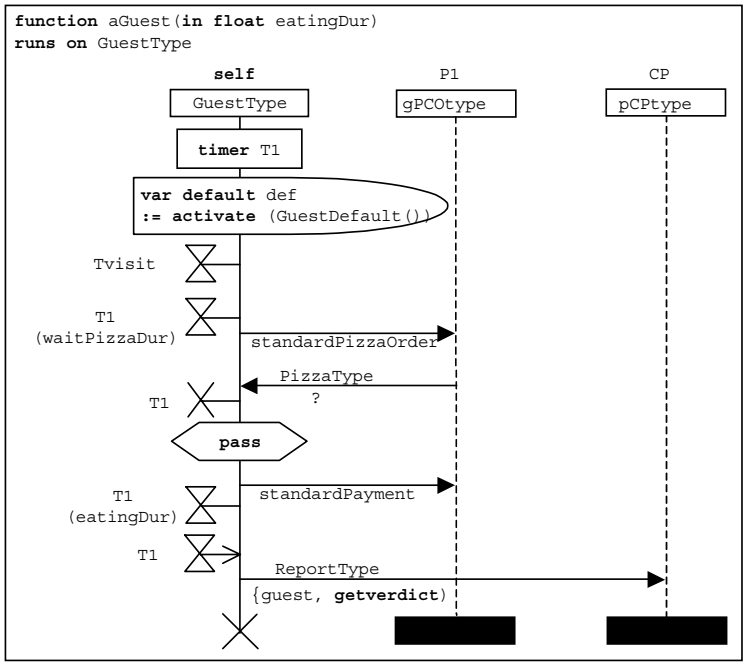



```
testcase MyTestCase (
    in boolean           internetService,     // SERVICES
    in boolean           phoneService,
    in boolean           restaurantService,
    in integer           totalNrCreatedPTCs,  // TERMINATION
    in integer           maxNrActivePTCs,     // CONTROL
    inout integer        nrPass,              // RETURN
    inout integer        nrFail,
    inout integer        nrInc
    )
  runs on MtcType
  system TestSystemType
{
    var ReportType report;

    var default def := activate (StandardDefault());

    map(self:P1, system:mPCO);

    if (internetService)  {
        newInternetPTC();
    }

    if (totalNrCreatedPTCs != createdPTCs
        and phoneService) {

        if (maxNrActivePTCs == 1) {
            all component.done;
        }
        newPhonePTC();
    }

    while ( totalNrCreatedPTCs != createdPTCs ) {

        alt {
            []  CP.receive(ReportType:?) -> value report {

                setverdict(report.lverdict);

                if (report.lverdict == pass) { nrPass := nrPass + 1; }
                if (report.lverdict == fail) { nrFail := nrFail + 1; }
                if (report.lverdict == inconc) { nrInc := nrInc + 1; }
                activePTCs := activePTCs - 1;

                if (maxNrActivePTCs == 1) {
                    all component.done;
                }
                if (report.kind == internet) {
                    newInternetPTC();
                }
                if (report.kind == phone) {
                    newPhonePTC();
                }
                if (report.kind == guest) {
                    newGuest(1200.0);
                }
            }


            [else] {

                if (maxNrActivePTCs < activePTCs
                    and restaurantService) {
                      newGuest(1200.0);
                    }
            }
        }
    }

    all component.done;
    stop;
  }
```

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

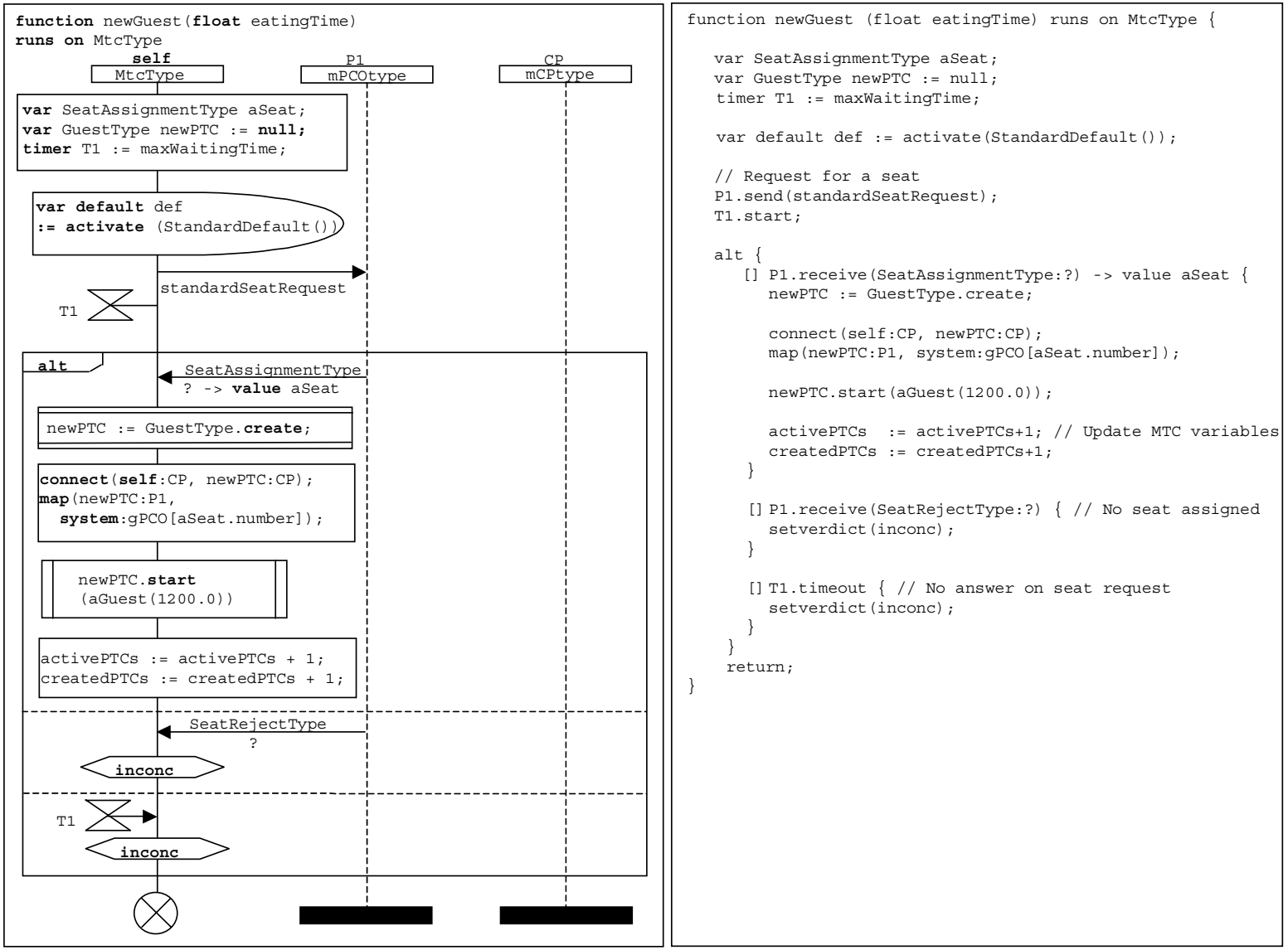
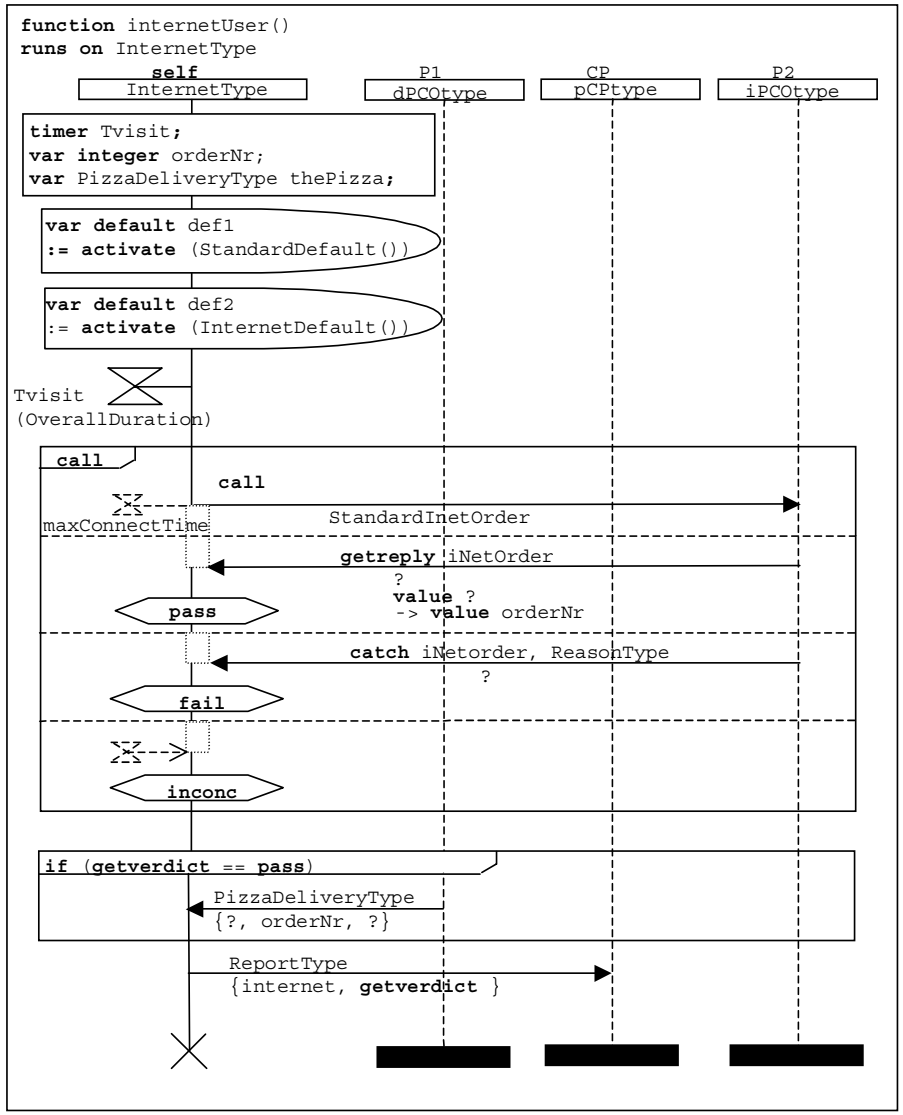
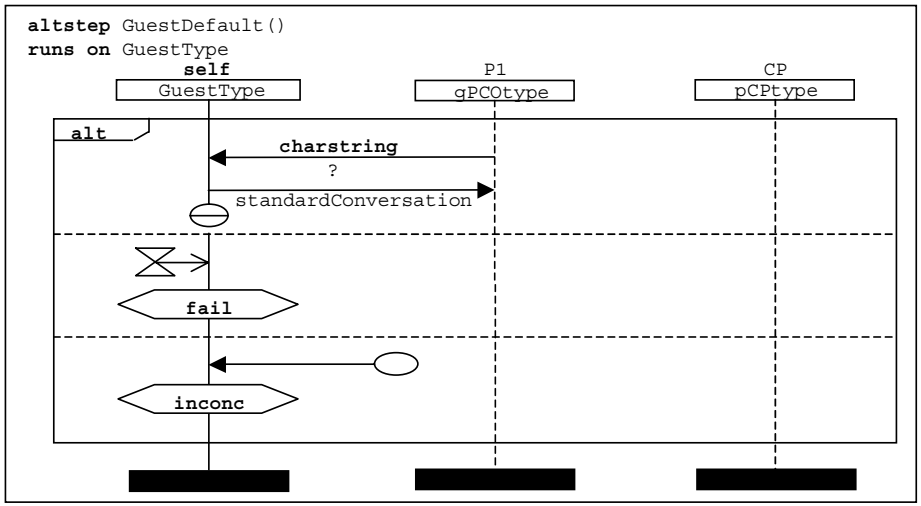


Figure E.3/Z.142 – Restaurant example – newGuest function

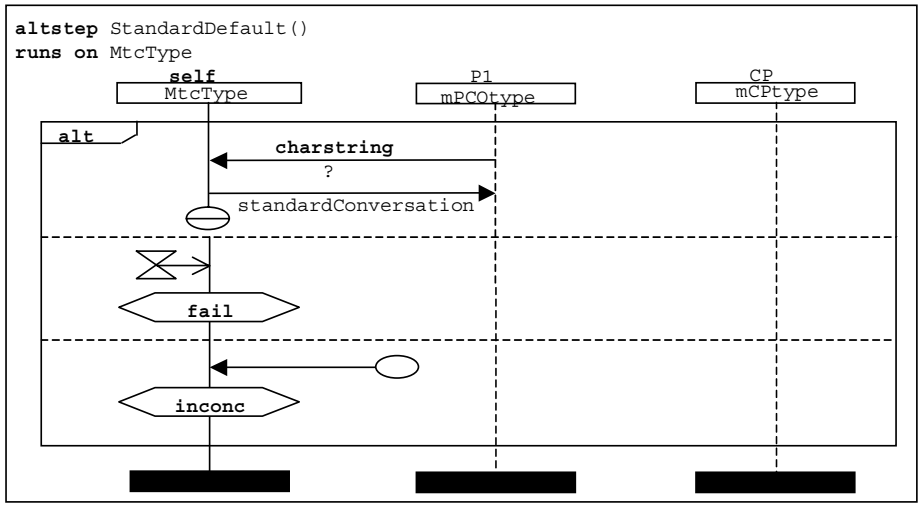



```
function internetUser () runs on InternetType {
  // ***
  // *** Purpose: Specifies the behaviour of an
  // *** internet guest
  // ***

  timer Tvisit;
  var integer orderNr;
  var PizzaDeliveryType thePizza;

  var default def1 := activate(StandardDefault());
  var default def2 := activate(InternetDefault());
  Tvisit.start(OverallDuration);

  P2.call(StandardINetOrder, maxConnectTime) {
     [] P2.getreply (iNetOrder:? value ?)
              -> value orderNr {
           setverdict(pass);
        }
     [] P2.catch (iNetOrder, ReasonType : ?) {

           setverdict(fail);
        }
     [] P2.catch (timeout) {
           setverdict(inconc);
        }
   };
   if (getverdict == pass) {
     P1.receive(PizzaDeliveryType
           : { ?, orderNr, ?});
   }
   CP.send(ReportType : {internet, getverdict});
   stop;
 }
```

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

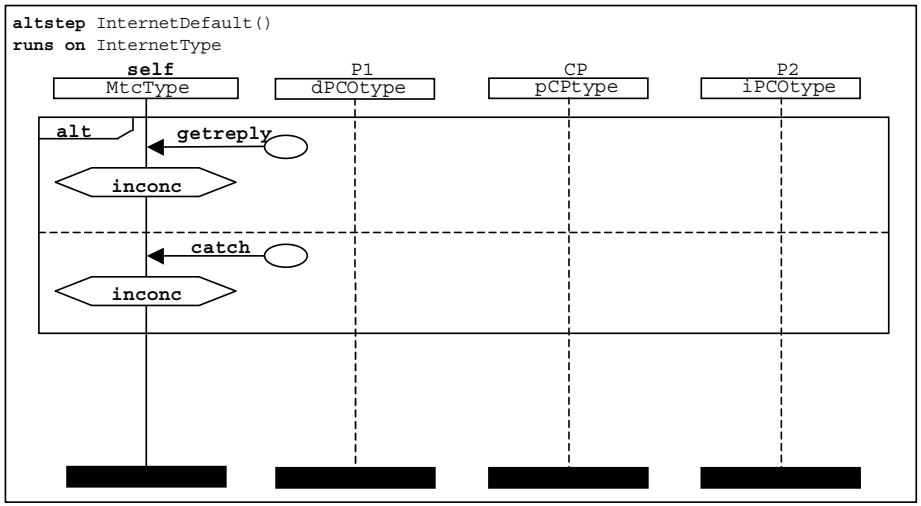



```
altstep GuestDefault() runs on GuestType {
   // ***
   // *** Purpose: Default behaviour for
   // *** message based ports
   // ***

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

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

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

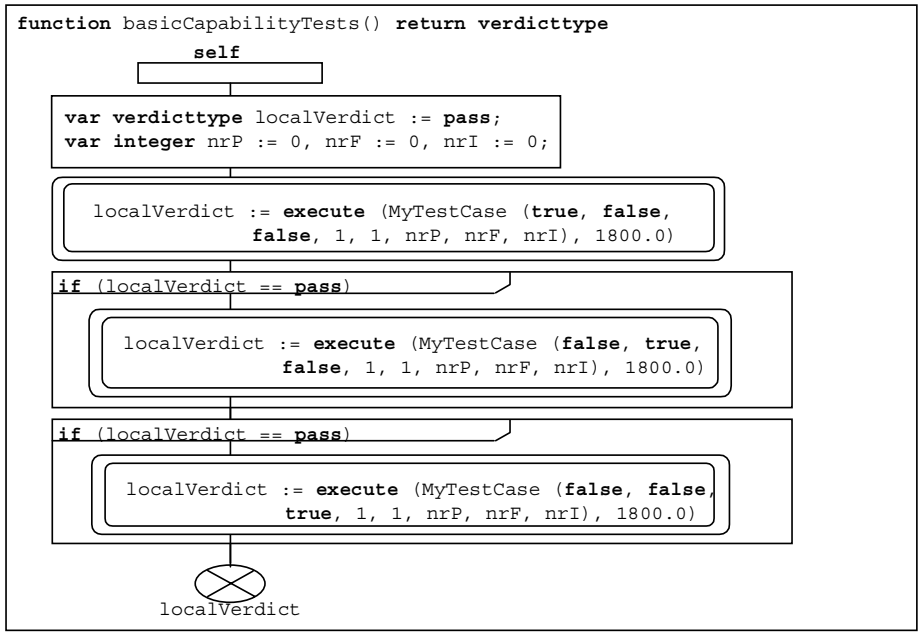



```
altstep StandardDefault() runs on MtcType {
   // ***
   // *** Purpose: Default behaviour for
   // **** message based ports
   // ***

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

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

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

Figure E.5/Z.142 – Restaurant example – GuestDefaut 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 E.6/Z.142 – Restaurant example – internetDefault altstep and basicCapabilityTests function

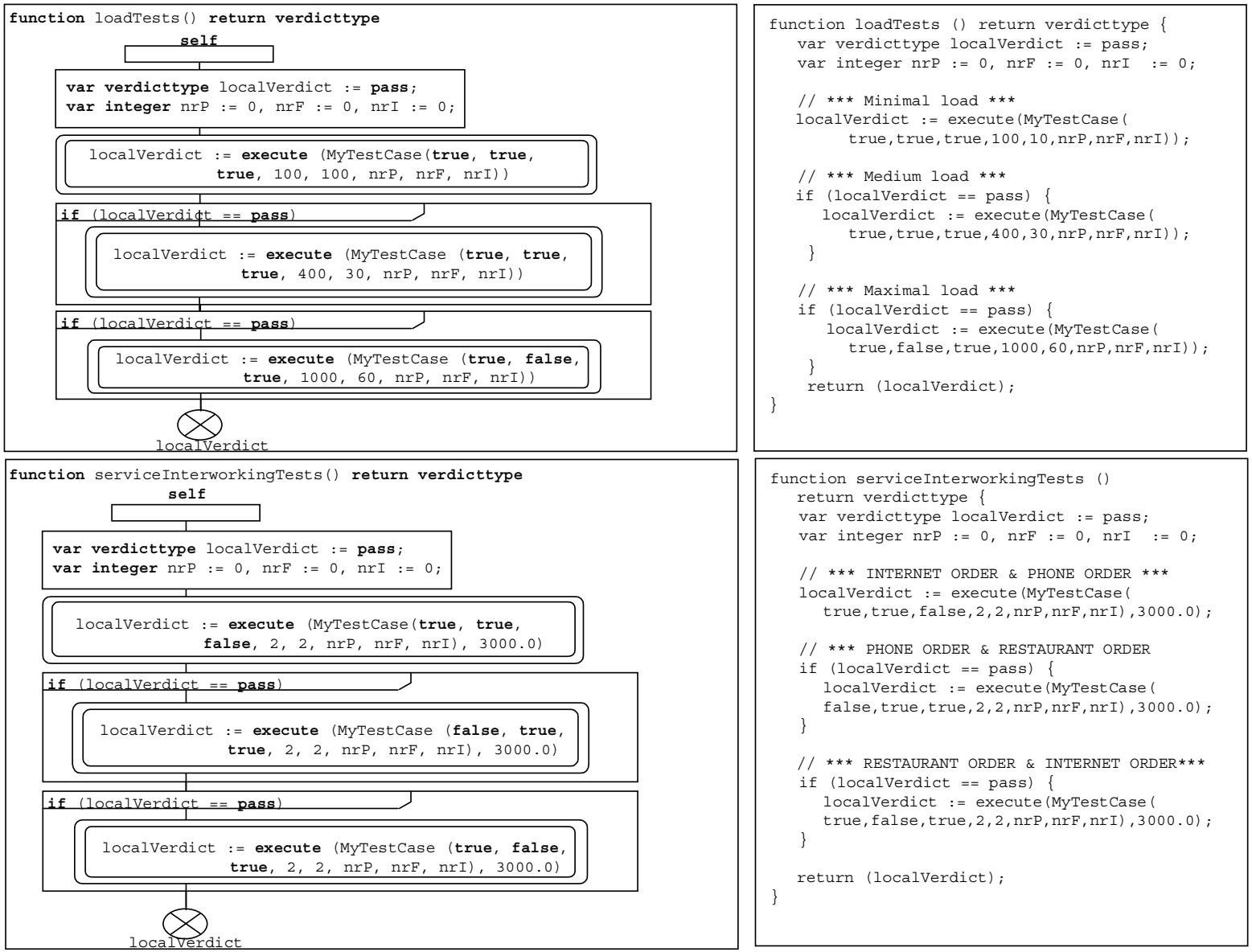
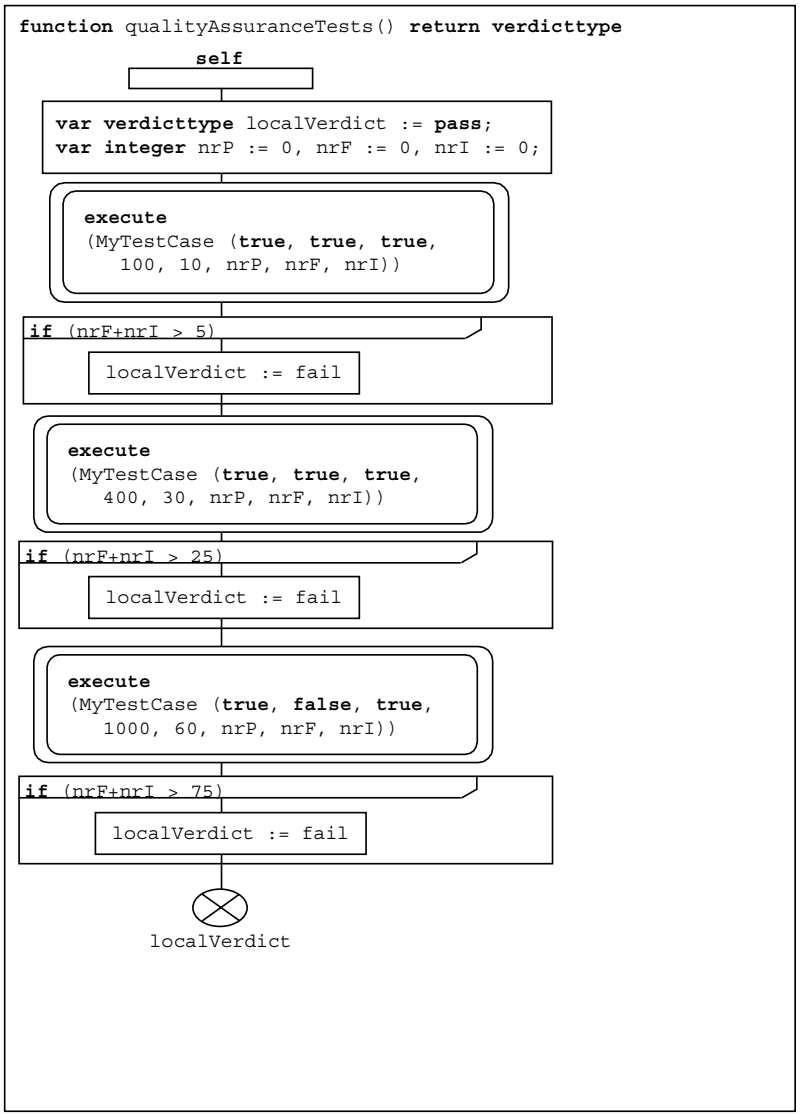


Figure E.7/Z.142 – Restaurant example – loadTests and serviceInterworkingTests functions




```
function qualityAssuranceTests ()
   return verdicttype {

   var verdicttype localVerdict := pass;
   var integer nrP := 0,
               nrF := 0,
               nrI  := 0;

   // *** Quality under Minimal load ***
   execute(MyTestCase(true,true,true,100,10,
         nrP,nrF,nrI));

   if ( nrF + nrI > 5 ) {
      localVerdict := fail;
   }

   // *** Quality under Medium load ***
   execute(MyTestCase(true,true,true,400,30,
         nrP,nrF,nrI));

   if ( nrF + nrI > 25 ) {
      localVerdict := fail;
   }

   // *** Quality under Maximal load ***
   execute(MyTestCase(true,false,true,1000,60,
         nrP,nrF,nrI));

   if ( nrF + nrI > 75 ) {
      localVerdict := fail;
   }

   return (localVerdict);

} // end function qualityAssuranceTests
```

Figure E.8/Z.142 – Restaurant example – qualityAssuranceTests function

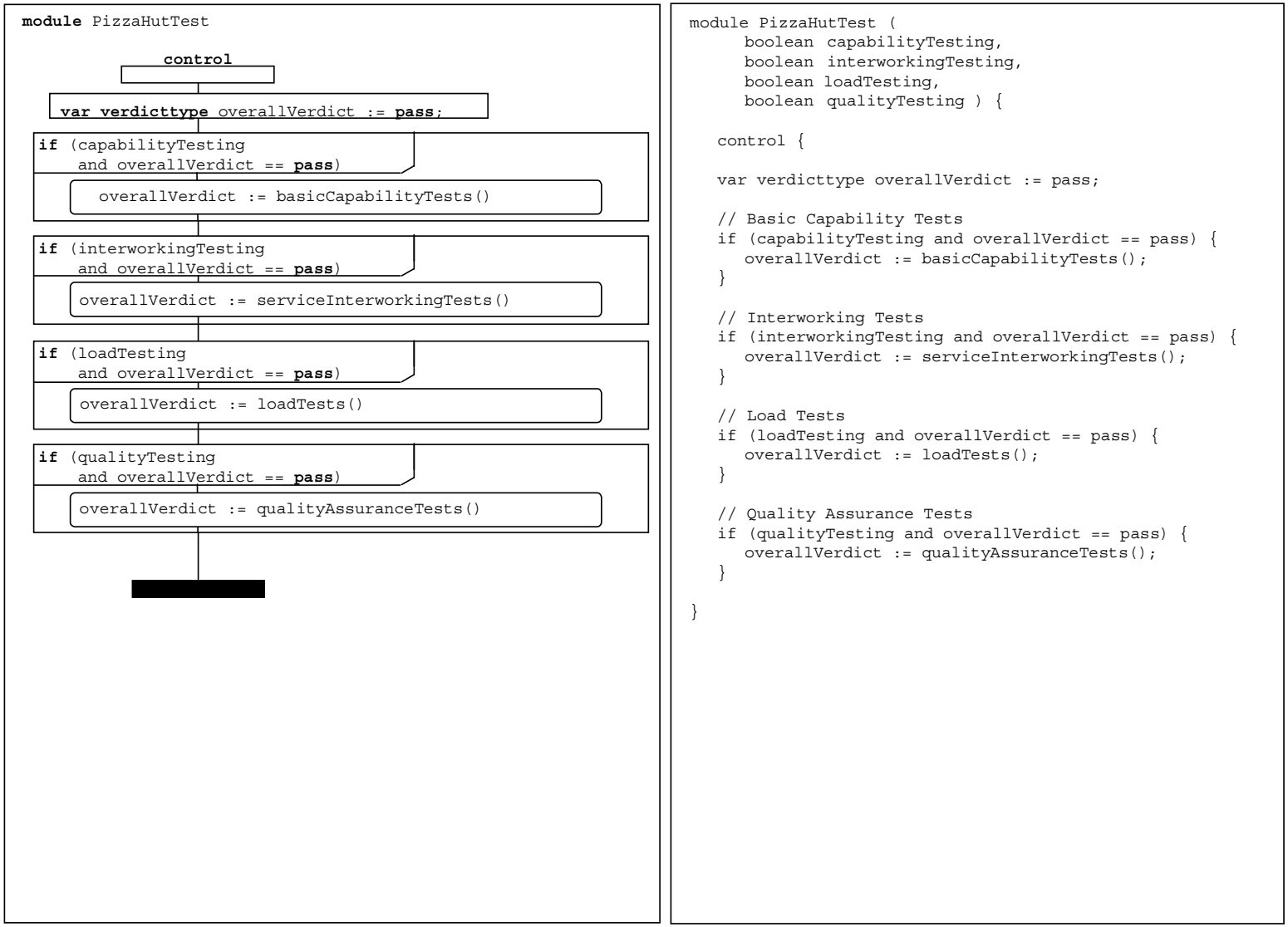
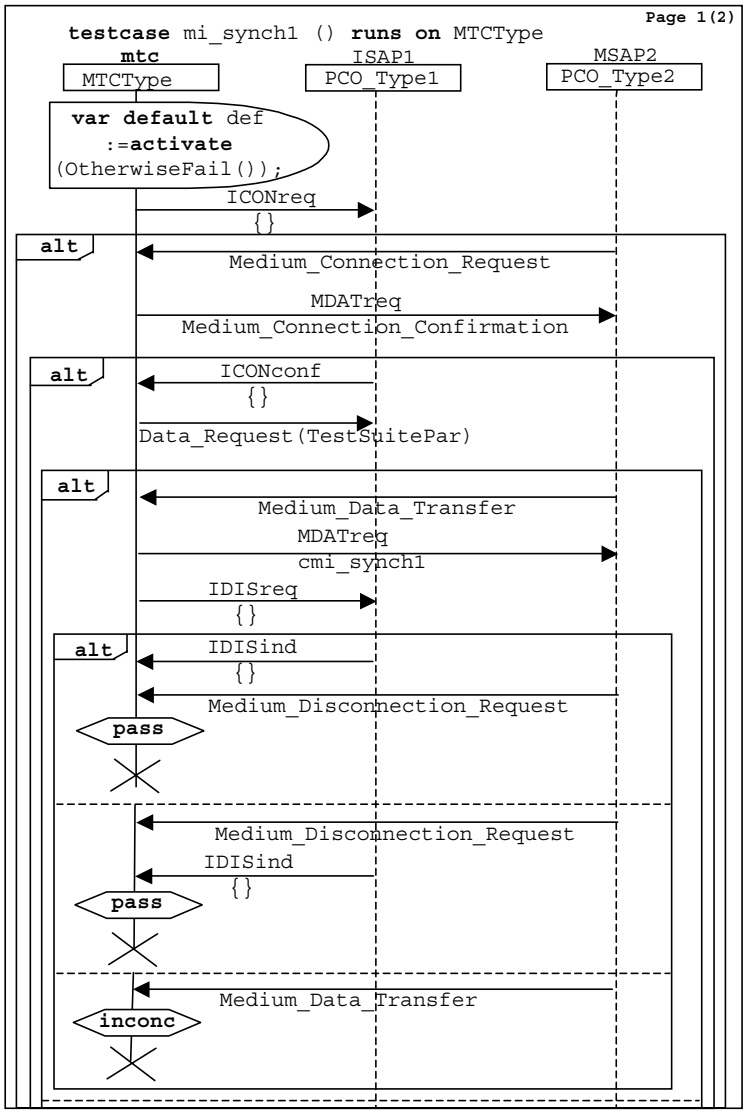


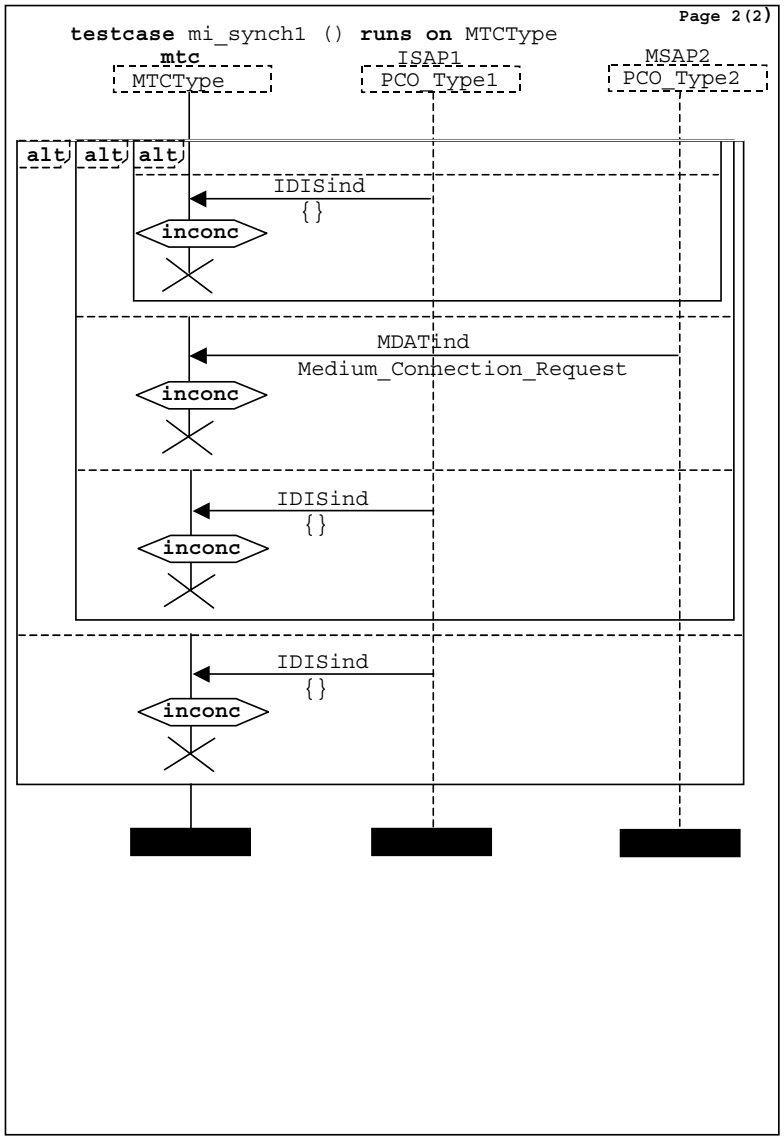
Figure E.9/Z.142 – Restaurant example – PizzaHutTest module

E.2 The INRES example



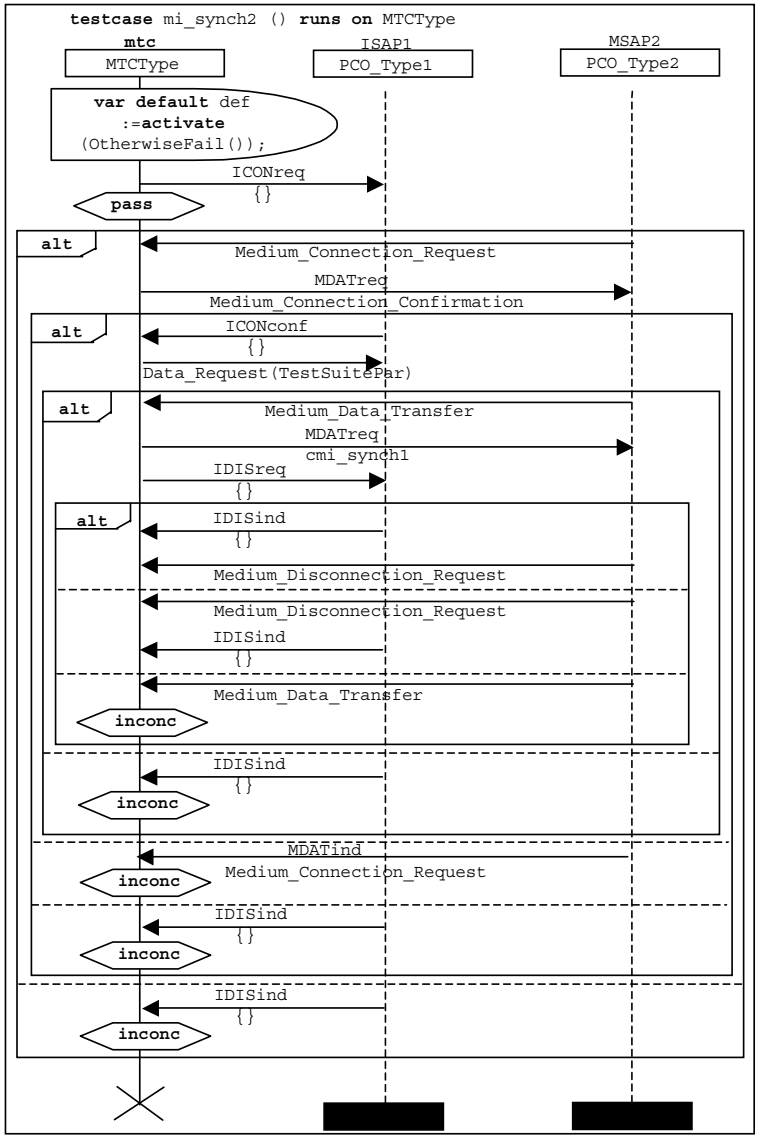

```
testcase mi_synch1 () runs on MTCType {

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

   /* Inline template definition */
   ISAP1.send( ICONreq:{} );

   alt { /* alt1 */
      [] MSAP2.receive( Medium_Connection_Request ) {
                          /* use of a template */
         MSAP2.send( MDATreq:Medium_Connection_Confirmation );
                          /*optional template type*/

      alt { /* alt2 */
         [] ISAP1.receive ( ICONconf:{} ) {
            ISAP1.send ( Data_Request(TestSuitePar) );


            alt { /* alt3 */
               [] MSAP2.receive( Medium_Data_Transfer ) {
                  MSAP2.send ( MDATreq:cmi_synch1() );
                  ISAP1.send ( IDISreq:{} );



                  alt { /* alt4 */
                     [] ISAP1.receive (IDISind:{} ) {
                        MSAP2.receive(
                              Medium_Disconnection_Request );
                        setverdict(pass);
                        stop;
                     }

                     [] MSAP2.receive(
                              Medium_Disconnection_Request ){
                        ISAP1.receive( IDISind:{} );
                        setverdict(pass);
                        stop;
                     }

                     [] MSAP2.receive(Medium_Data_Transfer ) {
                        setverdict(inconclusive);
                        stop;
                     }
                  } /* end alt4 */
               }
```

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



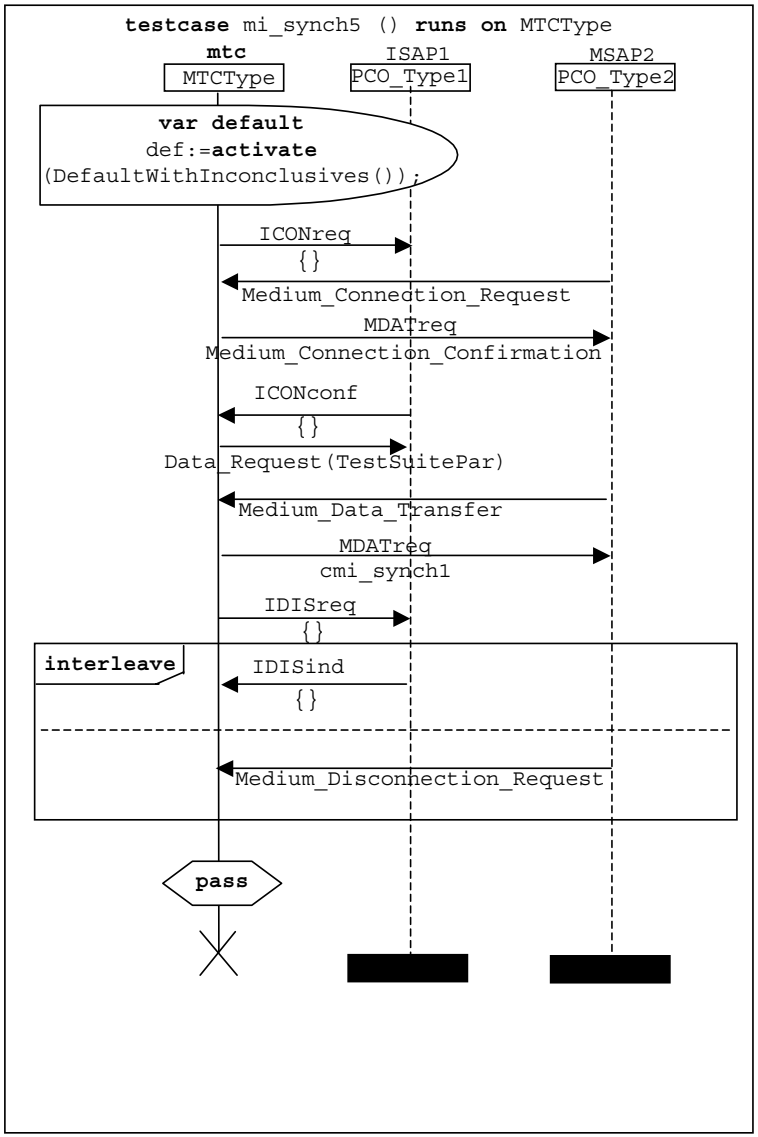

```
/* testcase mi_synch1 () continuation */




              [] ISAP1.receive( IDISind:{} ) {
                 setverdict(inconclusive);
                 stop;
              }
           } /* end alt3 */
        }

        [] MSAP2.receive(
                MDATind:Medium_Connection_Request) {
           setverdict(inconclusive);
        stop;
        }

        [] ISAP1.receive( IDISind:{} ) {
        setverdict(inconclusive);
        stop;
        }
     }
   } /* end alt2 */


[] ISAP1.receive( IDISind:{} ) {
      setverdict(inconclusive);
      stop;
    }
   } /* end alt1 */
 } /* End testcase mi_synch1 */
```

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



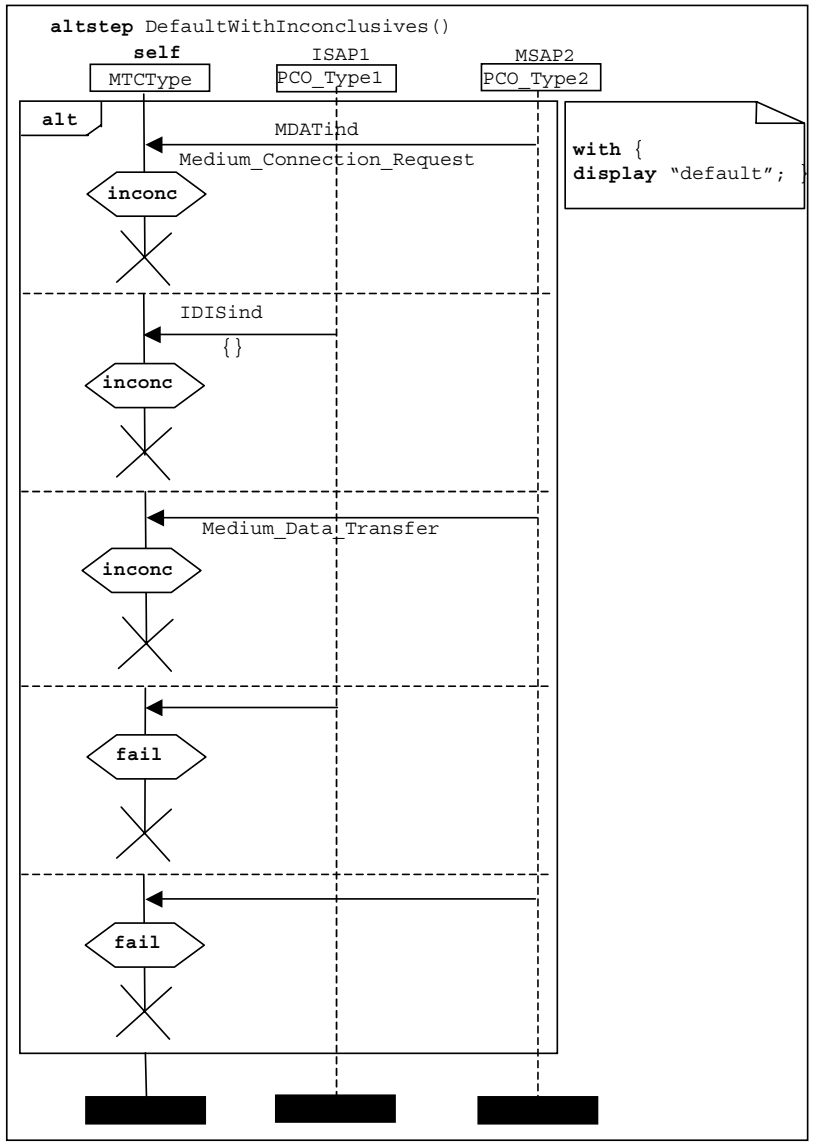

```
testcase mi_synch2 () runs on MTCType {



   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);
            }
         }
      }
      [] ISAP1.receive( IDISind:{} ) {
         setverdict(inconclusive);
      }
   }
   stop; } /* End testcase mi_synch2 */
```

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



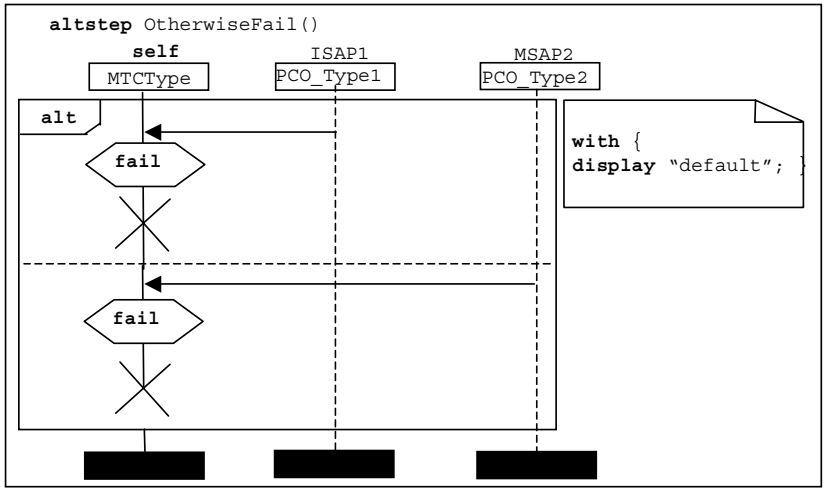

```
testcase mi_synch5 () runs on MTCType {



  var default
       def := activate(DefaultWithInconclusives );
                    /* Default activation */
  /* message ONE and response to ONE */
  ISAP1.send( ICONreq:{} );
  MSAP2.receive(Medium_Connection_Request );

  /* message TWO and response to TWO */
  MSAP2.send(
          MDATreq:Medium_Connection_Confirmation );
  ISAP1.receive ( ICONconf:{} );

  /* message THREE and response to THREE */
  ISAP1.send ( Data_Request(TestSuitePar) );
  MSAP2.receive ( Medium_Data_Transfer );

  /* messages FOUR and FIVE */
  MSAP2.send ( MDATreq:cmi_synch1 );
  ISAP1.send ( IDISreq:{} );

 interleave {
    /* the two responses to messages FOUR and
       FIVE can arrive in any order */
    [] ISAP1.receive(IDISind:{}) {};
    [] MSAP2.receive(
         Medium_Disconnection_Request ) {};
  }

  setverdict(pass);

  stop;

} /* End testcase mi_synch5 */
```

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

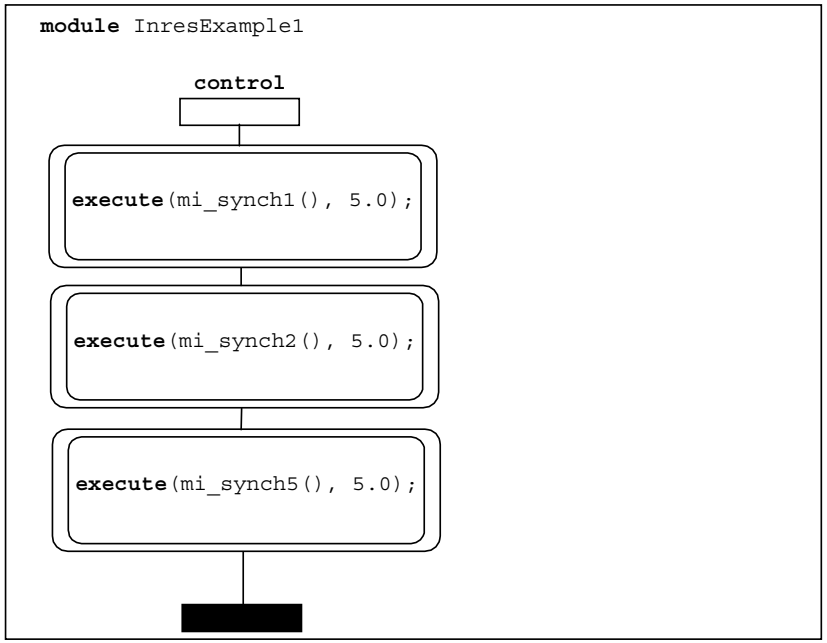



```
altstep DefaultWithInconclusives() {

    /* INCONCLUSIVE CASES */


   [] MSAP2.receive( MDATind:Medium_Connection_Request) {

      setverdict(inconclusive);
      stop;
   }


   [] ISAP1.receive ( IDISind:{} ) {

      setverdict(inconclusive);
      stop;
   }



   [] MSAP2.receive ( Medium_Data_Transfer ) {

      setverdict(inconclusive);
      stop;
   }


  /* FAIL CASES */

   [] ISAP1.receive {

      setverdict(fail);
      stop;

   }


   [] MSAP2.receive {

      setverdict(fail);
      stop;
   }


} with { display "default"; }
```

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




```
altstep OtherwiseFail() {

   [] ISAP1.receive {

      setverdict(fail);

      stop;
   }

   [] MSAP2.receive {

      setverdict(fail);

      stop;
  }
} with { display "default"; }
```




```
module InresExample1 {
   ...

   control InresExample {

      execute (mi_synch1(), 5.0);

      execute (mi_synch2(), 5.0);

      execute (mi_synch5(), 5.0);

   } // end control part

}
```

Figure E.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**

Imprimé en Suisse
Genève, 2006