

International Telecommunication Union

**ITU-T**

TELECOMMUNICATION  
STANDARDIZATION SECTOR  
OF ITU

**Z.163**

(11/2007)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE  
ASPECTS FOR TELECOMMUNICATION SYSTEMS

Formal description techniques (FDT) – Testing and Test  
Control Notation (TTCN)

---

**Testing and Test Control Notation version 3:  
TTCN-3 graphical presentation format (GFT)**

ITU-T Recommendation Z.163



ITU-T Z-SERIES RECOMMENDATIONS  
**LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS**

<b>FORMAL DESCRIPTION TECHNIQUES (FDT)</b>	
Specification and Description Language (SDL)	Z.100–Z.109
Application of formal description techniques	Z.110–Z.119
Message Sequence Chart (MSC)	Z.120–Z.129
Extended Object Definition Language (eODL)	Z.130–Z.139
User Requirements Notation (URN)	Z.150–Z.159
<b>Testing and Test Control Notation (TTCN)</b>	<b>Z.160–Z.199</b>
<b>PROGRAMMING LANGUAGES</b>	
CHILL: The ITU-T high level language	Z.200–Z.209
<b>MAN-MACHINE LANGUAGE</b>	
General principles	Z.300–Z.309
Basic syntax and dialogue procedures	Z.310–Z.319
Extended MML for visual display terminals	Z.320–Z.329
Specification of the man-machine interface	Z.330–Z.349
Data-oriented human-machine interfaces	Z.350–Z.359
Human-machine interfaces for the management of telecommunications networks	Z.360–Z.379
<b>QUALITY</b>	
Quality of telecommunication software	Z.400–Z.409
Quality aspects of protocol-related Recommendations	Z.450–Z.459
<b>METHODS</b>	
Methods for validation and testing	Z.500–Z.519
<b>MIDDLEWARE</b>	
Processing environment architectures	Z.600–Z.609

*For further details, please refer to the list of ITU-T Recommendations.*

## **ITU-T Recommendation Z.163**

### **Testing and Test Control Notation version 3: TTCN-3 graphical presentation format (GFT)**

#### **Summary**

ITU-T Recommendation Z.163 defines the graphical presentation format for the TTCN-3 core language as defined in ITU-T Recommendation Z.161. This presentation format uses a subset of Message Sequence Charts as defined in ITU-T Recommendation Z.120 with test specific extensions.

This Recommendation is based on the core TTCN-3 language defined in ITU-T Recommendation Z.161. It is particularly suited to display tests as GFTs. It is not limited to any particular kind of test specification.

The specification of other formats is outside the scope of this Recommendation.

#### **Source**

ITU-T Recommendation Z.163 was approved on 13 November 2007 by ITU-T Study Group 17 (2005-2008) under the ITU-T Recommendation A.8 procedure.

## FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications, information and communication technologies (ICTs). The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

## NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

Compliance with this Recommendation is voluntary. However, the Recommendation may contain certain mandatory provisions (to ensure e.g. interoperability or applicability) and compliance with the Recommendation is achieved when all of these mandatory provisions are met. The words "shall" or some other obligatory language such as "must" and the negative equivalents are used to express requirements. The use of such words does not suggest that compliance with the Recommendation is required of any party.

## INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementers are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database at <http://www.itu.int/ITU-T/ipr/>.

© ITU 2008

All rights reserved. No part of this publication may be reproduced, by any means whatsoever, without the prior written permission of ITU.

## CONTENTS

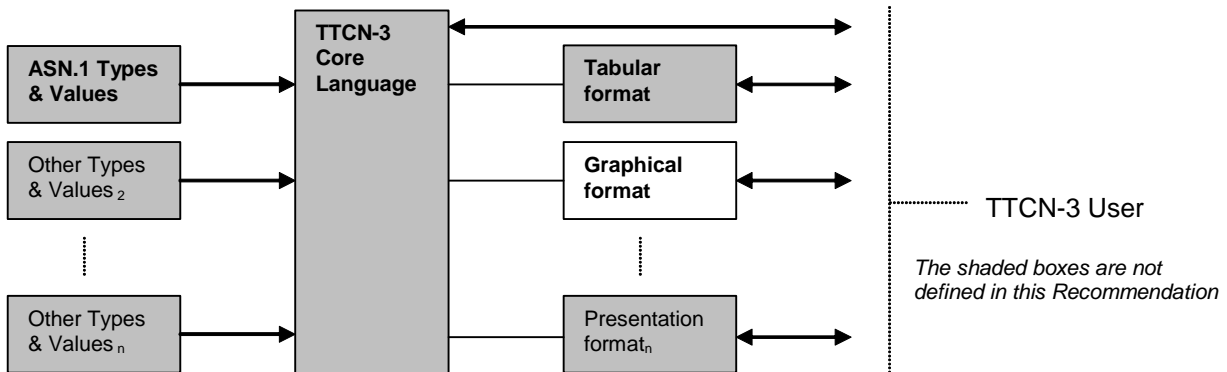
	<b>Page</b>
1 Scope .....	1
2 References.....	1
3 Abbreviations.....	1
4 Overview .....	2
5 GFT language concepts .....	3
6 Mapping between GFT and TTCN-3 Core language .....	5
7 Module structure.....	5
8 GFT symbols .....	7
9 GFT diagrams .....	9
9.1 Common properties .....	9
9.2 Control diagram.....	10
9.3 Test case diagram .....	11
9.4 Function diagram.....	12
9.5 Altstep diagram .....	13
10 Instances in GFT diagrams .....	14
10.1 Control instance.....	14
10.2 Test component instances.....	14
10.3 Port instances.....	15
11 Elements of GFT diagrams .....	15
11.1 General drawing rules.....	15
11.2 Invoking GFT diagrams .....	17
11.3 Declarations.....	19
11.4 Basic program statements.....	21
11.5 Behavioural program statements .....	24
11.6 Default handling .....	29
11.7 Configuration operations .....	30
11.8 Communication operations.....	33
11.9 Timer operations.....	52
11.10 Test verdict operations .....	55
11.11 External actions .....	56
11.12 Specifying attributes.....	56
Annex A – GFT BNF.....	57
A.1 Meta-language for GFT.....	57
A.2 Conventions for the syntax description .....	57
A.3 The GFT grammar .....	58
Annex B – Reference guide for GFT .....	81

	<b>Page</b>
Annex C – Examples .....	110
C.1    The Restaurant example .....	110
C.2    The INRES example.....	119

## Introduction

The graphical presentation format of TTCN-3 (GFT) is based on [ITU-T Z.120] defining Message Sequence Charts (MSC). GFT uses a subset of MSC with test specific extensions. The majority of extensions are textual extensions only. Graphical extensions are defined to ease the readability of GFT diagrams. Where possible, GFT is defined like MSC, so that established MSC tools with slight modifications can be used for the graphical definition of TTCN-3 test cases in terms of GFT.

The core language of TTCN-3 is defined in [ITU-T Z.161] and provides a full text-based syntax, static semantics and operational semantics as well as a definition for the use of the language with ASN.1. The GFT presentation format provides an alternative way of displaying the core language (see Figure 1).



**Figure 1 – User's view of the core language and the various presentation formats**

The core language may be used independently of GFT. However, GFT cannot be used without the core language. Use and implementation of the GFT shall be done on the basis of the core language.

This Recommendation defines:

- the language concepts of GFT;
- the guidelines for the use of GFT;
- the grammar of GFT;
- the mapping from and to the TTCN-3 core language.

Together, these characteristics form the graphical presentation format of TTCN-3.





# ITU-T Recommendation Z.163

## Testing and Test Control Notation version 3: TTCN-3 graphical presentation format (GFT)

### 1 Scope

This Recommendation defines the graphical presentation format for the TTCN-3 core language as defined in [ITU-T Z.161]. This presentation format uses a subset of Message Sequence Charts as defined in [ITU-T Z.120] with test specific extensions.

This Recommendation is based on the core TTCN-3 language defined in [ITU-T Z.161]. It is particularly suited to display tests as GFTs. It is not limited to any particular kind of test specification.

The specification of other formats is outside the scope of this Recommendation.

### 2 References

The following ITU-T Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published. The reference to a document within this Recommendation does not give it, as a stand-alone document, the status of a Recommendation.

[ITU-T X.292] ITU-T Recommendation X.292 (2002), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – The Tree and Tabular Combined Notation (TTCN)*.

ISO/IEC 9646-3:1998, *Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 3: The Tree and Tabular Combined Notation (TTCN)*.

[ITU-T Z.120] ITU-T Recommendation Z.120 (2004), *Message sequence chart (MSC)*.

[ITU-T Z.161] ITU-T Recommendation Z.161 (2007), *Testing and Test Control Notation version 3: TTCN-3 core language*.

[ITU-T Z.162] ITU-T Recommendation Z.162 (2007), *Testing and Test Control Notation version 3: TTCN-3 tabular presentation format (TFT)*.

### 3 Abbreviations

This Recommendation uses the following abbreviations and acronyms:

BNF	Backus-Naur Form
CATG	Computer Aided Test Generation
GFT	Graphical presentation Format of TTCN-3
MSC	Message Sequence Chart
MTC	Main Test Component
PTC	Parallel Test Component
SUT	System Under Test

TFT Tabular presentation Format of TTCN-3

TTCN Testing and Test Control Notation

#### 4 Overview

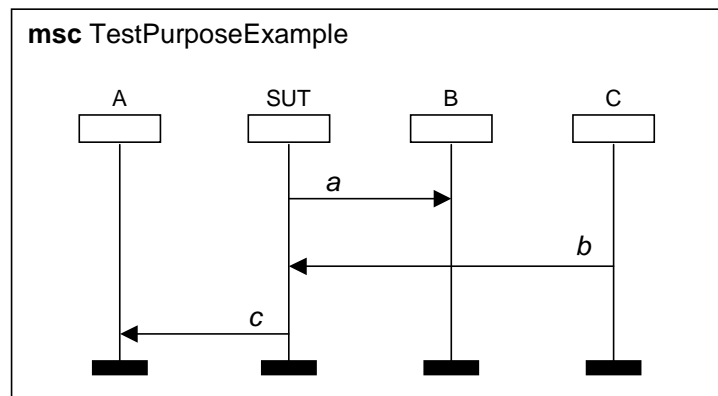
According to the OSI conformance testing methodology defined in [ITU-T X.292], testing normally starts with the identification of test purposes. A test purpose is defined as:

*"A prose description of a well-defined objective of testing, focusing on a single conformance requirement or a set of related conformance requirements as specified in the appropriate OSI specification".*

Having identified all test purposes an abstract test suite is developed that consists of one or more abstract test cases. An abstract test case defines the actions of the tester processes necessary to validate part (or all) of a test purpose.

Applying these terms to Message Sequence Charts (MSCs) we can define two categories for their usage:

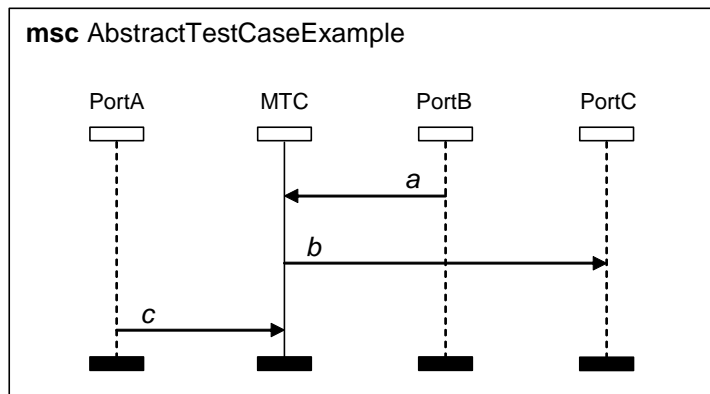
- 1) *Using MSCs for the definition of test purposes* – Typically, an MSC specification that is developed as a use-case or as part of a system specification can be viewed as test purpose, i.e., it describes a requirement for the SUT in the form of a behaviour description that can be tested. For example, Figure 2 presents a simple MSC describing the interaction between instances representing the SUT and its interfaces A, B and C. In a real implementation of such a system the interfaces A, B and C may map onto service access points or ports. The MSC in Figure 2 only describes the interaction with the SUT and does not describe the actions of the test components necessary to validate the SUT behaviour, i.e., it is a test purpose description.



**Figure 2 – MSC describing the interaction of an SUT with its interfaces**

- 2) *Using MSCs for the definition of abstract test cases* – An MSC specification describing an abstract test case specifies the behaviour of the test components necessary to validate a corresponding test purpose. Figure 3 presents a simple MSC abstract test case description. It shows one Main Test Component (MTC) that exchanges the messages *a*, *b* and *c* with the SUT via the ports PortA, PortB and PortC in order to reach the test purpose shown in Figure 2. The messages *a* and *c* are sent by the SUT via the ports A and B (Figure 2) and received by the MTC (Figure 3) via the same ports. The message *b* is sent by the MTC and received by the SUT.

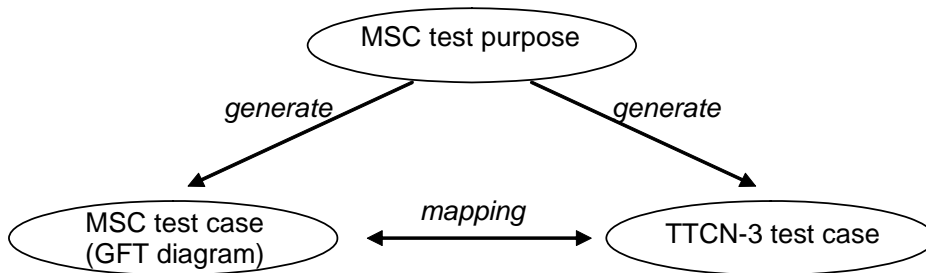
NOTE – The examples in Figures 2 and 3 are only simple examples to illustrate the different usages of MSC for testing. The diagrams will be more complicated in case of a distributed SUT that consists of several processes or a distributed test configuration with several test components.



**Figure 3 – MSC describing the interaction of an MTC with SUT interfaces**

In identifying these two categories of MSC usage two distinct areas of work can be identified (see Figure 4):

- a) *Generation of abstract test cases from MSC test purpose descriptions* – TTCN-3 core language or GFT may be used to represent the abstract test cases. However, it is perceived that test case generation from test purposes is non-trivial and involves the usage and development of Computer Aided Test Generation (CATG) techniques.
- b) *Development of a Graphical presentation format for TTCN-3 (GFT) and definition of the mapping between GFT and TTCN-3.*



**Figure 4 – Relations between MSC test purpose description, MSC test case descriptions and TTCN-3**

This Recommendation focuses on item b), i.e., it defines GFT and the mapping between GFT and the TTCN-3 core language.

## 5 GFT language concepts

GFT represents graphically the behavioural aspects of TTCN-3 like the behaviour of a test case or a function. It does not provide graphics for data aspects like declaration of types and templates.

GFT defines no graphical representation for the structure of a TTCN-3 module, but specifies the requirements for such a graphical representation (see also clause 7).

NOTE – The order and the grouping of definitions and declarations in the module definitions part define the structure of a TTCN-3 module.

GFT defines no graphical representation for:

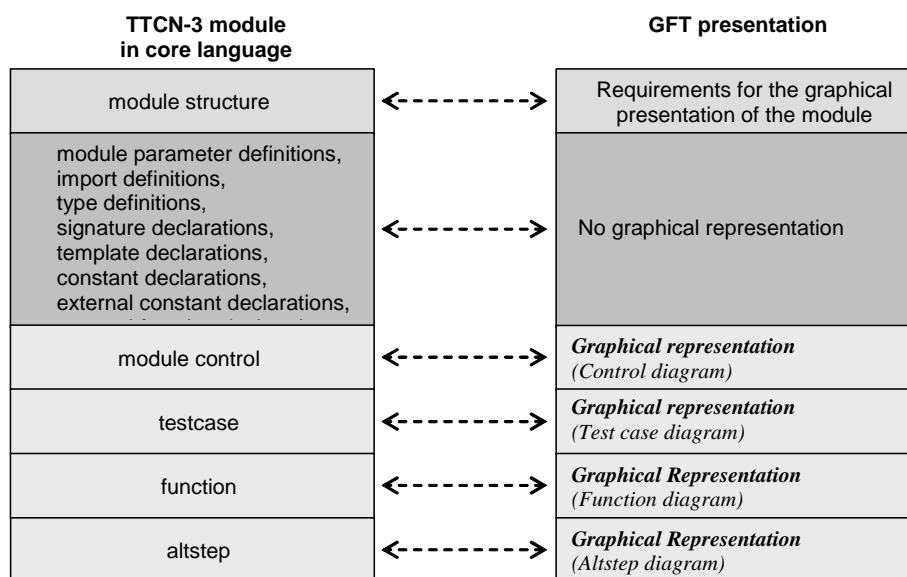
- module parameter definitions;
- import definitions;
- type definitions;
- signature declarations;
- template declarations;
- constant declarations;
- external constant declarations; and
- external function declarations.

TTCN-3 definitions and declarations without a corresponding GFT presentation may be presented in the TTCN-3 core language or in the tabular presentation format for TTCN-3 (TFT) ([ITU-T Z.162]).

GFT provides graphics for TTCN-3 behaviour descriptions. This means a GFT diagram provides a graphical presentation of either:

- the control part of a TTCN-3 module;
- a TTCN-3 test case;
- a TTCN-3 function; or
- a TTCN-3 altstep.

The relation between a TTCN-3 module and a corresponding GFT presentation is shown in Figure 5.



**Figure 5 – Relation between TTCN-3 core language and the corresponding GFT description**

GFT is based on MSC ([ITU-T Z.120]) and, thus, a GFT diagram maps onto an MSC diagram. Although GFT uses most of the graphical MSC symbols, the inscriptions of some MSC symbols have been adapted to the needs of testing and, in addition, some new symbols have been defined in order to emphasize test specific aspects. Though, the new symbols can be mapped onto valid MSC.

- the representation of port instances;

- the creation of test components;
- the start of test components;
- the return from a function call;
- the repetition of alternatives;
- the time supervision of a procedure-based call;
- the execution of test cases;
- the activation and deactivation of defaults;
- the labelling and goto; and
- the timers within call statements.

A complete list of all symbols used in GFT is presented in clause 8.

## 6 Mapping between GFT and TTCN-3 Core language

GFT provides graphical means for TTCN-3 behaviour definitions. The control part and each function, altstep and test case of a TTCN-3 core language module can be mapped onto a corresponding GFT diagram and vice versa. This means:

- the module control part can be mapped onto a control diagram (see clause 9.2) and vice versa;
- a test case can be mapped onto a test case diagram (see clause 9.3) and vice versa;
- a function in core language can be mapped onto a function diagram (see clause 9.4) and vice versa;
- an altstep can be mapped onto an altstep diagram (see clause 9.5) and vice versa.

NOTE 1 – GFT provides no graphical presentations for definitions of module parameters, types, constants, signatures, templates, external constants and external functions in the module definitions part. These definitions may be presented directly in core language or by using another presentation format, e.g., the tabular presentation format.

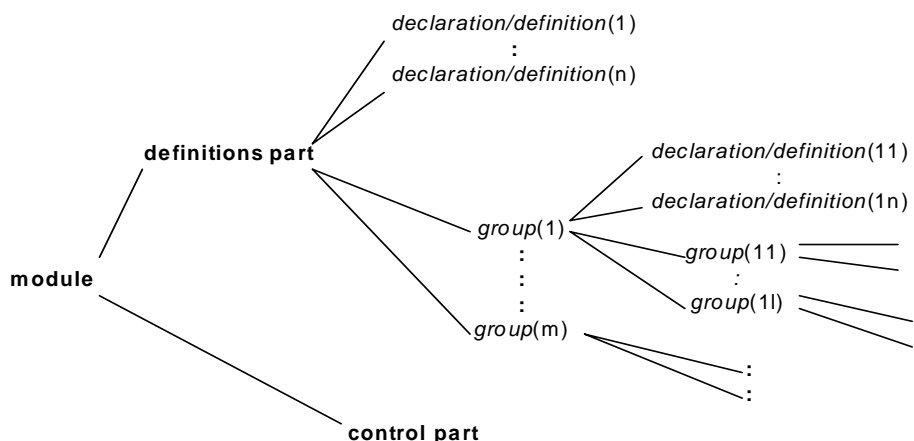
Each declaration, operation and statement in the module control and each test case, altstep or function can be mapped onto a corresponding GFT representation and vice versa.

The order of declarations, operations and statements within a module control, test case, altstep or function definition is identical to the order of the corresponding GFT representations within the related control, test case, altstep or function diagram.

NOTE 2 – The order of GFT constructs in a GFT diagram is defined by the order of the GFT constructs in the diagram header (declarations only) and the order of the GFT constructs along the control instance (control diagram) or component instance (test case diagram, altstep diagram or function diagram).

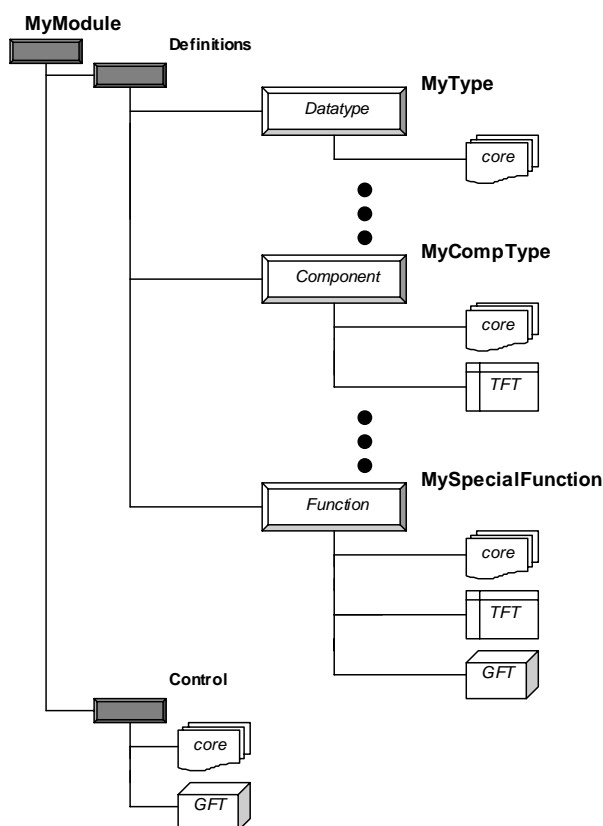
## 7 Module structure

As shown in Figure 6, a TTCN-3 module has a tree structure. A TTCN-3 module is structured into a module definitions part and a module control part. The module definitions part consists of definitions and declarations that may be structured further by means of groups. The module control part cannot be structured into sub-structures; it defines the execution order and the conditions for the execution of the test cases.

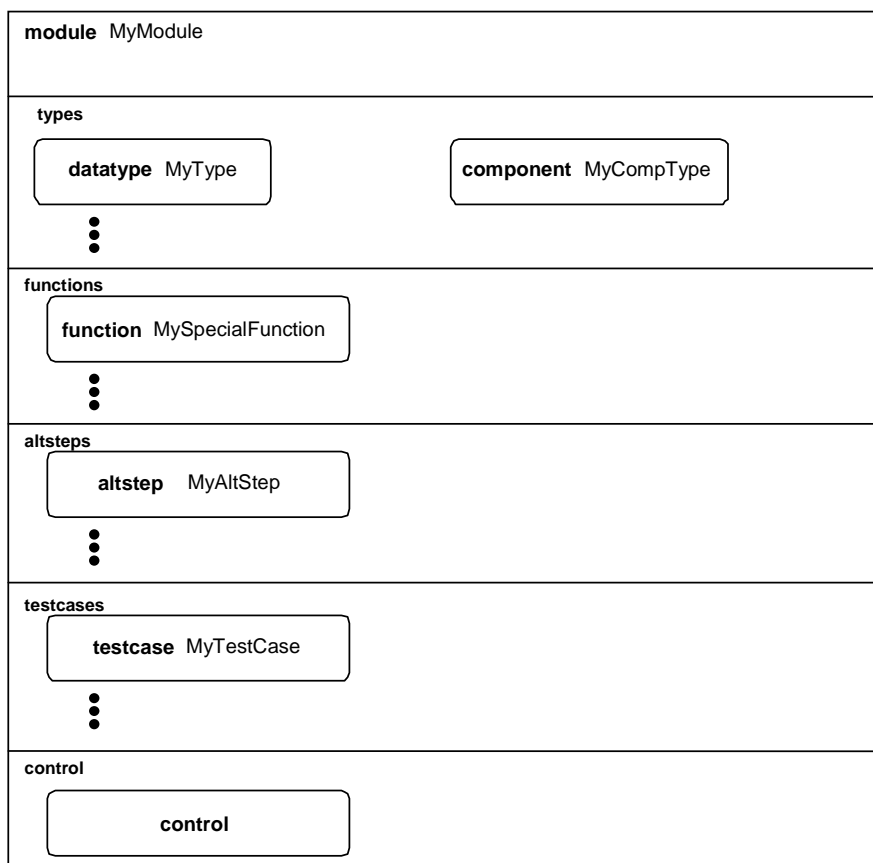


**Figure 6 – Structure of TTCN-3 modules**

GFT provides diagrams for all "behavioural" leaves of the module tree structure, i.e., for the module control part, for functions, for altsteps and for test cases. GFT defines no concrete graphics for the module tree-structure, however appropriate tool support for GFT requires a graphical presentation of the structure of a TTCN-3 module. The TTCN-3 module structure may be provided in form of an organizer view (Figure 7) or the MSC document-like presentation (Figure 8). An advanced tool may even support different presentations of the same object, e.g., the organizer view in Figure 7 indicates that some definitions are provided within several presentation formats, e.g., function MySpecialFunction is available in core language, in form of a TFT table and as GFT diagram.



**Figure 7 – Various presentation formats in an organizer view of a TTCN-3 module structure**



**Figure 8 – Graphical MSC document-like presentation of a TTCN-3 module structure**

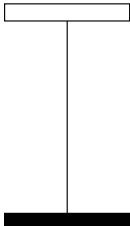

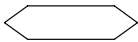


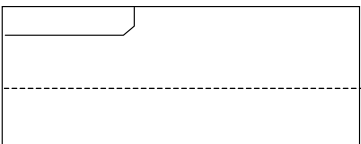
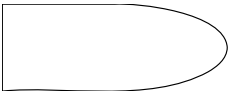



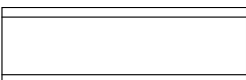



## 8 GFT symbols

This clause presents all graphical symbols used within GFT diagrams and comments their typical usage within GFT.

**Table 1 – GFT symbols**







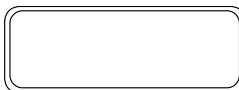
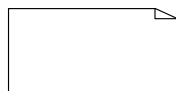
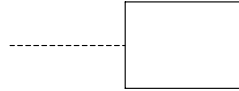
GFT element	Symbol	Description
Frame symbol		Used to frame GFT diagrams
Reference symbol		Used to represent the invocation of functions and altsteps
Port instance symbol		Used to represent port instances

**Table 1 – GFT symbols**

GFT element	Symbol	Description
Component instance symbol		Used to represent test components and the control instance
Action box symbol		Used for textual TTCN-3 declarations and statements, to be attached to a component symbol
Condition symbol		Used for textual TTCN-3 boolean expressions, verdict setting, port operations (start, stop and clear) and the done statement, to be attached to a component symbol
Labelling symbol		Used for TTCN-3 labelling and goto, to be attached to a component symbol
Goto symbol		Used for TTCN-3 labelling and goto, to be attached to a component symbol
Inline expression symbol		Used for TTCN-3 if-else, for, while, do-while, alt, call and interleave statement, to be attached to a component symbol
Default symbol		Used for TTCN-3 activate and deactivate statement, to be attached to a component symbol
Stop symbol		Used for TTCN-3 stop statement, to be attached to a component symbol
Return symbol		Used for TTCN-3 return statement, to be attached to a component symbol
Repeat symbol		Used for TTCN-3 repeat statement, to be attached to a component symbol
Create symbol		Used for TTCN-3 create statement, to be attached to a component symbol
Start symbol		Used for TTCN-3 start statement, to be attached to a component symbol
Message symbol		Used for TTCN-3 send, call, reply, raise, receive, getcall, getreply, catch, trigger and check statement, to be attached to a component symbol and a port symbol
Found symbol		Used for representing TTCN-3 receive, getcall, getreply, catch, trigger and check from any port, to be attached to a component symbol



**Table 1 – GFT symbols**

GFT element	Symbol	Description
Suspension region symbol		Used in combination with a blocking call, to be within a call inline expression and attached to a component symbol
Start timer symbol		Used for TTCN-3 start timer operation, to be attached to a component symbol
Timeout timer symbol		Used for TTCN-3 timeout operation, to be attached to a component symbol
Stop timer symbol		Used for TTCN-3 stop timer operation, to be attached to a component symbol
Start implicit timer symbol		Used for TTCN-3 implicit timer start in blocking call, to be within a call inline expression and attached to a component symbol
Timeout implicit timer symbol		Used for TTCN-3 timeout exception in blocking call, to be within a call inline expression and attached to a component symbol
Execute symbol		Used for TTCN-3 execute test case statement, to be attached to a component instance symbol
Text symbol		Used for TTCN-3 with statement and comments, to be placed within a GFT diagram
Event comment symbol		Used for TTCN-3 comments associated to events, to be attached to events on component instance or port instance symbols

## 9 GFT diagrams

GFT provides the following diagram types:

- control diagram* for the graphical presentation of a TTCN-3 module control part;
- test case diagram* for the graphical presentation of a TTCN-3 test case;
- altstep diagram* for the graphical presentation of a TTCN-3 altstep; and
- function diagram* for the graphical presentation of a TTCN-3 function.

The different diagram types have some common properties.

### 9.1 Common properties

Common properties of GFT diagrams are related to the diagram area, diagram heading and paging.

#### 9.1.1 Diagram area

Each GFT control, test case, altstep and function diagram shall have a frame symbol (also called diagram frame) to define the diagram area. All symbols and text needed to define a complete and syntactically correct GFT diagram shall be made inside the diagram area.

NOTE – GFT has no language constructs like the MSC gates, which are placed outside of, but connected to the diagram frame.

### 9.1.2 Diagram heading

Each GFT diagram shall have a diagram heading. The diagram heading shall be placed in the upper left-hand corner of the diagram frame.

The diagram heading shall uniquely identify each GFT diagram type. The general rule to achieve this is to construct the heading from the keywords `testcase`, `altstep` or `function` followed by the TTCN-3 signature of the test case, altstep or function that should be presented graphically. For a GFT control diagram, the unique heading is constructed from the keyword `module` followed by the module name.

NOTE – In MSC, the keyword `msc` always precedes the diagram name to identify MSC diagrams. GFT diagrams do not have such a common keyword to identify GFT diagrams.

### 9.1.3 Paging

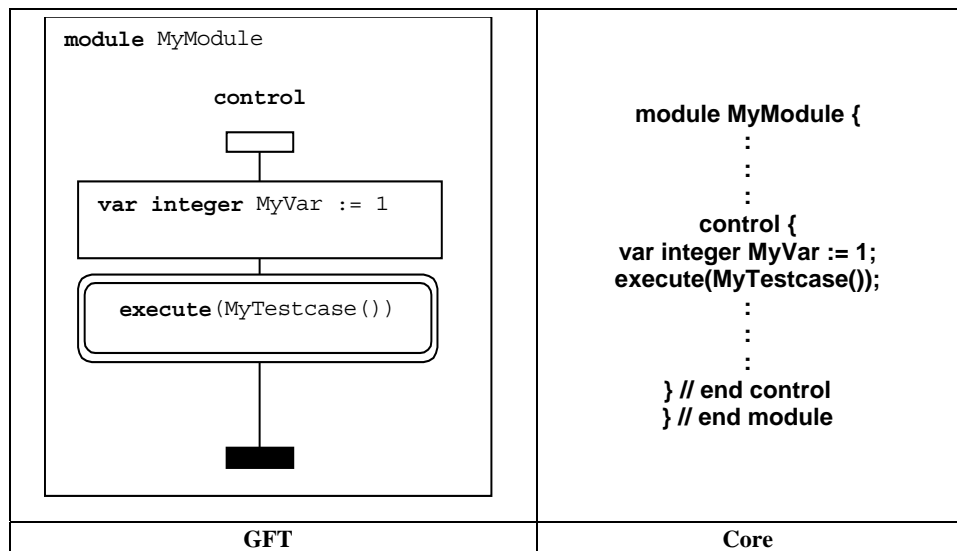
GFT diagrams may be organized in pages and a large GFT diagram may be split into several pages. Each page of a split diagram shall have a numbering in the upper right hand corner that identifies the page uniquely. The numbering is optional if the diagram is not split.

NOTE 1 – The concrete numbering scheme is considered to be a tools issue and is therefore outside the scope of this Recommendation. A simple numbering scheme may only assign a page number, whereas an advanced numbering scheme may support the reconstruction of a diagram only by using the numbering information on the different pages.

NOTE 2 – Paging requirements beyond the general numbering are considered to be tools issues and are therefore outside the scope of this Recommendation. For readability purposes, the diagram heading may be shown on each page, the instance line of an instance that will be continued on another page may be attached to the lower border of the page and the instance head of a continued instance may be repeated on the page that describes the continuation.

## 9.2 Control diagram

A GFT control diagram provides a graphical presentation of the control part of a TTCN-3 module. The heading of a control diagram shall be the keyword `module` followed by the module name. A GFT control diagram shall only include one component instance (also called control instance) with the instance name `control` without any type information. The control instance describes the behaviour of the TTCN-3 module control part. Attributes associated to the TTCN-3 module control part shall be specified within a text symbol in the control diagram. The principle shape of a GFT control diagram and the corresponding TTCN-3 core description are sketched in Figure 9.



**Figure 9 – Principle shape of a GFT control diagram and corresponding core language**

Within the control part, test cases can be selected or deselected for the test case execution with the use of Boolean expressions. Expressions, assignments, `log` statements, `label` and `goto` statements, `if-else` statements, `for` loop statements, `while` loop statements, `do while` loop statements, `stop` execution statements, and timer statements can be used to control the execution of test cases. Furthermore, functions can be used to group the test cases together with their preconditions for execution, which are invoked by the module control part.

The GFT representation of those language features is as described in the respective clauses below except that for the module control part the graphical symbols are attached to the control instance and not to a test component instance.

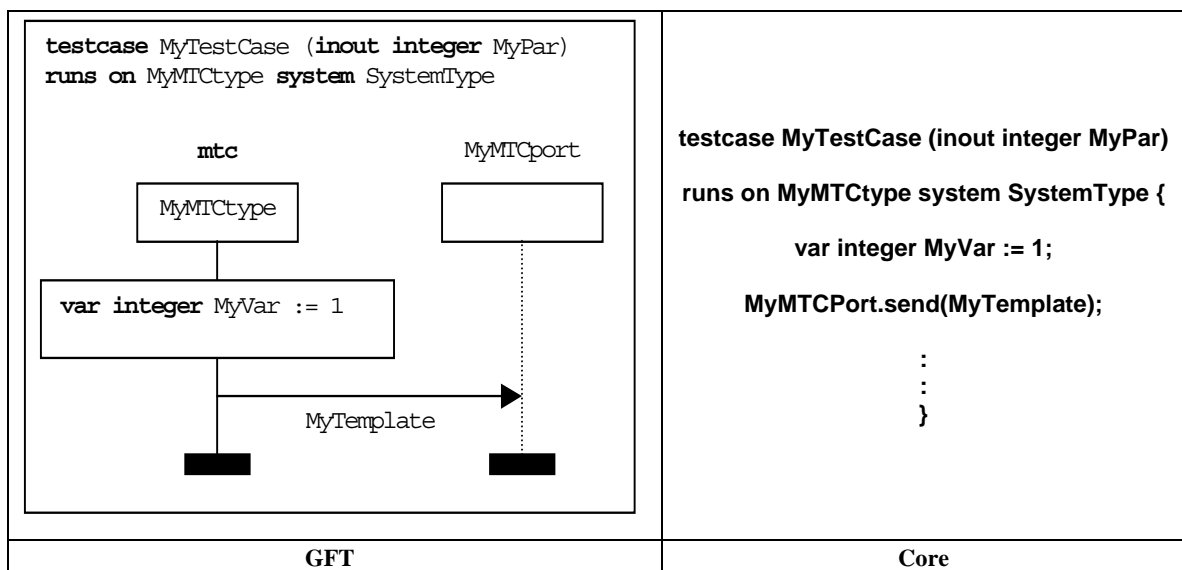
Please refer to clause 11.4 for the GFT representation of expressions, assignments, `log`, `label` and `goto`, `if-else`, `for` loop, `while` loop, `do while` loop, and `stop`, to clause 11.9 for timer operations and to clauses 9.4 and 11.2.2 for functions and their invocation.

### 9.3 Test case diagram

A GFT test case diagram provides a graphical presentation of a TTCN-3 test case. The heading of a test case diagram shall be the keyword `testcase` followed by the complete signature of the test case. Complete means that at least test case name and parameter list shall be present. The `runs on` clause is mandatory and the `system` clause is optional in the core language. If the system clause is specified in the corresponding core language, it shall also be present in the heading of the test case diagram.

A GFT test case diagram shall include one test component instance describing the behaviour of the `mtc` (also called `mtc` instance) and one port instance for each port owned by the `mtc`. The name associated with the `mtc` instance shall be `mtc`. The type associated with the `mtc` instance is optional, but if the type information is present, it shall be identical to the component type referred to in the `runs on` clause of the test case signature. The names associated with the port instances shall be identical to the port names defined in the component type definition of the `mtc`. The associated type information for port instances is optional. If the type information is present, port names and port types shall be consistent with the component type definition of the `mtc`. The `mtc` and port types are displayed in the component or port instance head symbol.

Attributes associated to the test case presented in GFT shall be specified within a text symbol in the test case diagram. The principle shape of a GFT test case diagram and the corresponding TTCN-3 core description are sketched in Figure 10.



**Figure 10 – Principle shape of a GFT test case diagram and corresponding core language**

A test case represents the dynamic test behaviour and can create test components. A test case may contain declarations, statements, communication and timer operations and invocation of functions or altsteps.

#### 9.4 Function diagram

GFT presents TTCN-3 functions by means of function diagrams. The heading of a function diagram shall be the keyword **function** followed by the complete signature of the function. Complete means that at least function name and parameter list shall be present. The **return** clause and the **runs on** clause are optional in the core language. If these clauses are specified in the corresponding core language, they shall also be present in the header of the function diagram.

A GFT function diagram shall include one test component instance describing the behaviour of the function and one port instance for each port usable by the function.

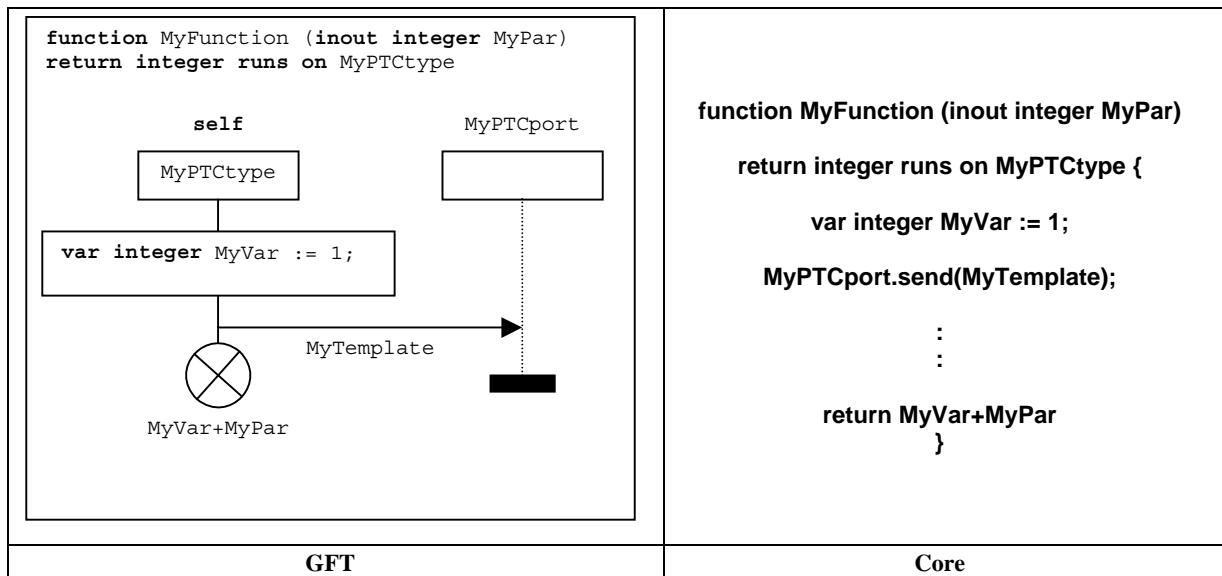
NOTE – The names and types of the ports that are usable by a function are passed in as parameters or are the port names and types that are defined in the component type definition referenced in the **runs on** clause.

The name associated with the test component instance shall be **self**. The type associated with the test component instance is optional, but if the type information is present, it shall be consistent with the component type in the **runs on** clause.

The names and types associated with the port instances shall be consistent with the port parameters (if the usable ports are passed in as parameters) or to the port declarations in the component type definition referenced in the **runs on** clause. The type information for port instances is optional.

**self** and port names are displayed on top of the component and resp. port instance head symbol. The component types and port types are displayed within the component and resp. port instance head symbol.

Attributes associated to the function presented in GFT shall be specified within a text symbol in the function diagram. The principle shape of a GFT function diagram and the corresponding TTCN-3 core description are sketched in Figure 11.



**Figure 11 – Principle shape of a GFT function diagram and corresponding core language**

A function is used to specify and structure test behaviour, define default behaviour or to structure computation in a module. A function may contain declarations, statements, communication and timer operations and invocation of function or altsteps and an optional return statement.

### 9.5 Altstep diagram

GFT presents TTCN-3 altsteps by means of altstep diagrams. The heading of an altstep diagram shall be the keyword **altstep** followed by the complete signature of the altstep. Complete means that at least altstep name and parameter list shall be present. The **runs on** clause is optional in the core language. If the **runs on** clause is specified in the corresponding core language, it shall also be present in the header of the altstep diagram.

A GFT altstep diagram shall include one test component instance describing the behaviour of the altstep and one port instance for each port usable by the altstep.

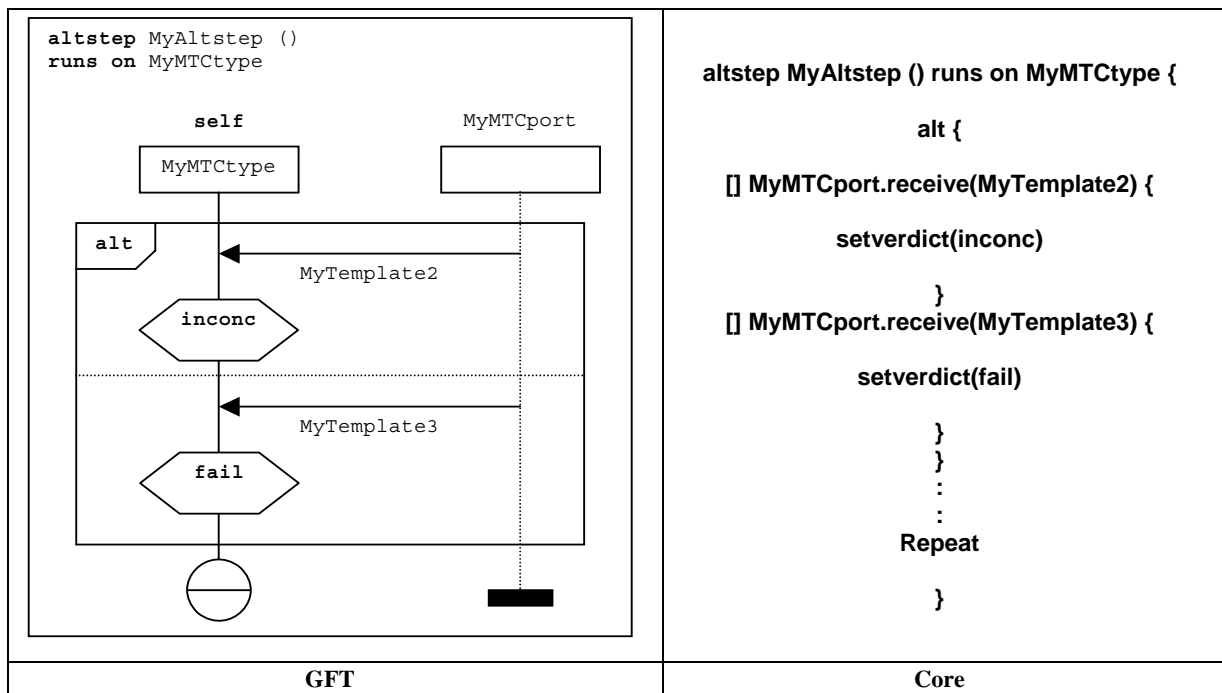
NOTE – The names and types of the ports that are usable by an altstep are passed in as parameters or are the port names and types that are defined in the component type definition referenced in the **runs on** clause.

The name associated with the test component instance shall be **self**. The type associated with the test component instance is optional, but if the type information is present, it shall be consistent with the component type in the **runs on** clause.

The names and types associated with the port instances shall be consistent with the port parameters (if the usable ports are passed in as parameters) or to the port declarations in the component type definition referenced in the **runs on** clause. The type information for port instances is optional.

**self** and port names are displayed on top of the component and resp. port instance head symbol. The component types and port types are displayed within the component and resp. port instance head symbol.

Attributes associated to the altstep shall be specified within a text symbol in the GFT altstep diagram. The principle shape of a GFT altstep diagram and the corresponding TTCN-3 core language are sketched in Figure 12.



**Figure 12 – Principle shape of a GFT altstep diagram and corresponding core language**

An altstep is used to specify default behaviour or to structure the alternatives of an `alt` statement. An altstep may contain statements, communication and timer operations and invocation of function or altsteps.

## 10 Instances in GFT diagrams

GFT diagrams include the following kinds of instances:

- *control instances* describing the flow of control for the module control part;
- *test component instances* describing the flow of control for the test component that executes a test case, function or altstep;
- *port instances* representing the ports used by the different test components.

### 10.1 Control instance

Only one control instance shall exist within a GFT control diagram (see clause 9.2). A control instance describes the flow of control of a module control part. A GFT control instance shall graphically be described by a component instance symbol with the mandatory name `control` placed on top of the instance head symbol. No instance type information is associated with a control instance. The principle shape of a control instance is shown in Figure 13 a).

### 10.2 Test component instances

Each GFT test case, function or altstep diagram includes one test component instance that describes the flow of control of that instance. A GFT test component instance shall graphically be described by an instance symbol with:

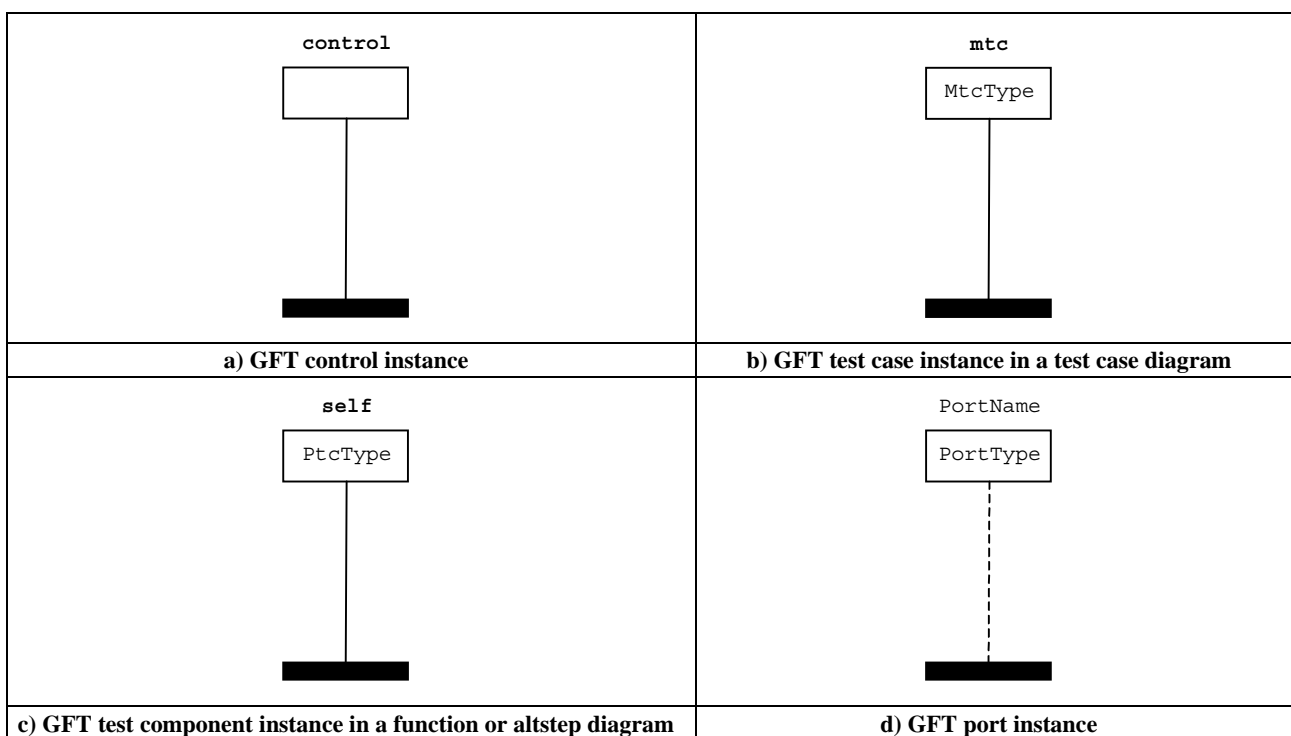
- the mandatory name `mtc` placed on top of the instance head symbol in the case of a test case diagram;
- the mandatory name `self` placed on top of the instance head symbol in the case of a function or altstep diagram.

The optional test component type may be provided within the instance head symbol. It has to be consistent with the test component type given after the `runs on` keyword in the heading of the GFT diagram.

The principle shape of a test component instance in a test case diagram is shown in Figure 13 b). The principle shape of a test component instance in a function or altstep diagram is shown in Figure 13 c).

### 10.3 Port instances

GFT port instances may be used within test case, altstep and function diagrams. A port instance represents a port that is usable by the test component that executes the specified test case, altstep or function. A GFT port instance is graphically described by a component instance symbol with a dashed instance line. The name of the represented port is mandatory information and shall be placed on top of the instance head symbol. The port type (optional) may be provided within the instance head symbol. The principle shape of a port instance is shown in Figure 13 d).



**Figure 13 – Principle shape of instance kinds in GFT diagrams**

## 11 Elements of GFT diagrams

This clause defines general drawing rules for the representation of specific TTCN-3 syntax elements (semicolons, comments). It describes how to display the execution of GFT diagrams and the graphical symbols associated with TTCN-3 language elements.

### 11.1 General drawing rules

General drawing rules in GFT are related to the usage of semicolons, TTCN-3 statements in action symbols and comments.

### 11.1.1 Usage of semicolons

All GFT symbols with the exception of the action symbol shall include only one statement in TTCN-3 core language. Only an action symbol may include a sequence of TTCN-3 statements (see clause 11.1.2).

The semicolon is optional if a GFT symbol includes only one statement in TTCN-3 core language (see Figure 14 a) and Figure 14 b)).

Semicolons shall separate the statements in a sequence of statements within an action symbol. The semicolon is optional for the last statement in the sequence (Figure 14 c)).

A sequence of variable, constant and timer declarations may also be specified in plain TTCN-3 core language following the heading of a GFT diagram. Semicolons shall also separate these declarations. The semicolon is optional for the last declaration in this sequence.

### 11.1.2 Usage of action symbols

The following TTCN-3 declarations, statements and operations are specified within action symbols: declarations (with the restrictions defined in clause 11.3), assignments, **log**, **connect**, **disconnect**, **map**, **unmap** and **action**.

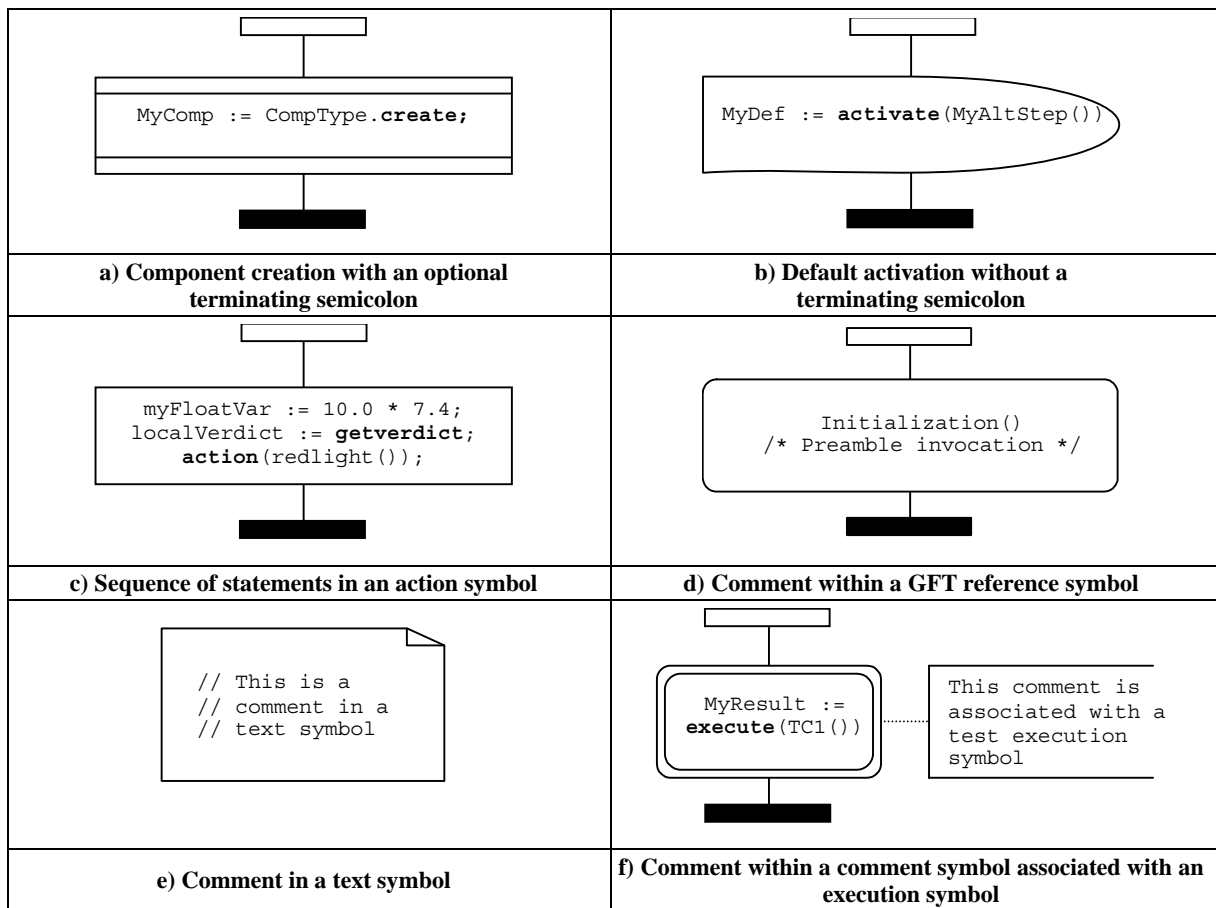
A sequence of declarations, statements and operations that shall be specified within action symbols variable can be specified in a single action symbol. It is not necessary to use a separate action symbol for each declaration, statement or operation.

### 11.1.3 Comments

GFT provide three possibilities to put comments into GFT diagrams:

- Comments may be put into GFT symbols following the symbol inscription and using the syntax for comments of the TTCN-3 core language (Figure 14 d)).
- Comments in the syntax for comments of the TTCN-3 core language can be put into text symbols and freely placed in the GFT diagram area (Figure 14 e)).
- The comment symbol can be used to associate comments to GFT symbols. A comment in a comment symbol can be provided in form of free text, i.e., the comment delimiter `/*`, `*/` and `/**` of the core language need not to be used (Figure 14 f)).





**Figure 14 – Examples for the effects of the general drawing rules**

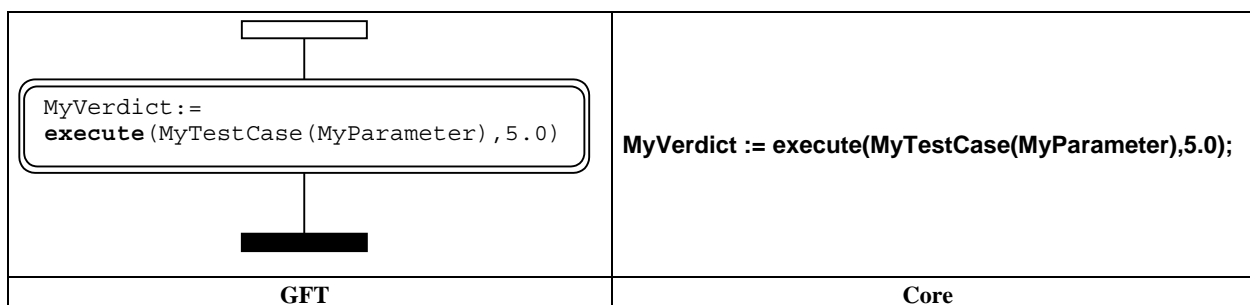
## 11.2 Invoking GFT diagrams

This clause describes how the individual kinds of GFT diagrams are invoked. Since there is no statement for executing the control part in TTCN-3 (as it is comparable to executing a program via main and out of the scope of TTCN-3), the clause discusses the execution of test cases, functions, and altsteps.

### 11.2.1 Execution of test cases

The execution of test cases is represented by use of the execute test case symbol (see Figure 15). The syntax of the **execute** statement is placed within that symbol. The symbol may contain:

- an **execute** statement for a test case with optional parameters and time supervision;
- optionally, the assignment of the returned verdict to a **verdicttype** variable; and
- optionally, the inline declaration of the **verdicttype** variable.



**Figure 15 – Test case execution**

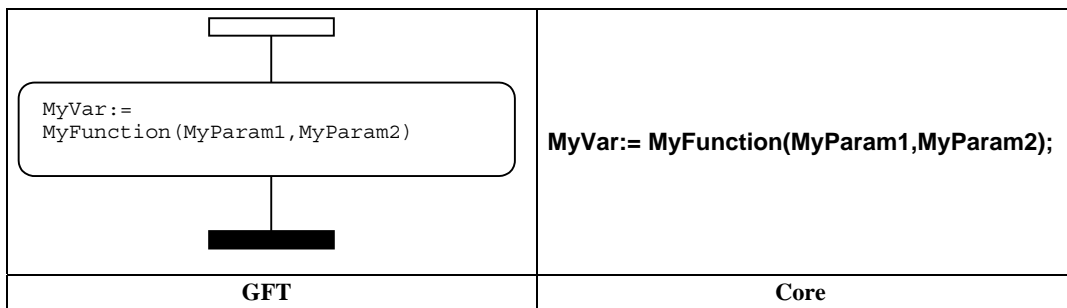
### 11.2.2 Invocation of functions

The invocation of functions is represented by the reference symbol (Figure 16), except of external and predefined functions (Figure 17) and except where a function is called inside a TTCN-3 language element that has a GFT representation (Figure 18).

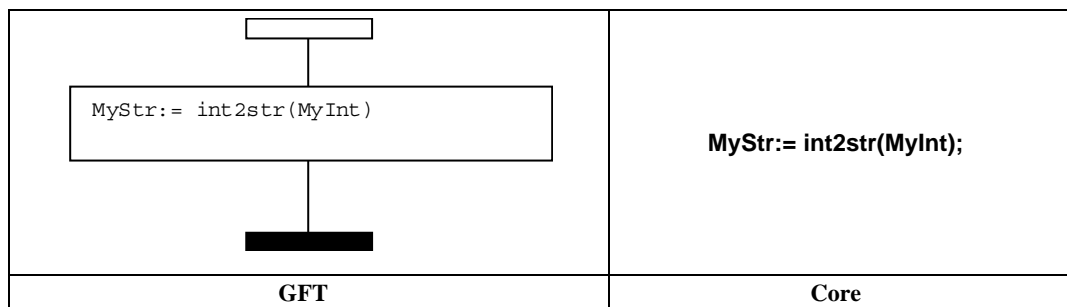
The syntax of the function invocation is placed within the reference symbol. The symbol may contain:

- the invocation of a function with optional parameters;
- an optional assignment of the returned value to a variable; and
- an optional inline declaration of the variable.

The reference symbol is only used for user defined functions defined within the current module. It shall not be used for external functions or predefined TTCN-3 functions, which shall be represented in their text form within an action form (Figure 17) or other GFT symbols (see example in Figure 18).

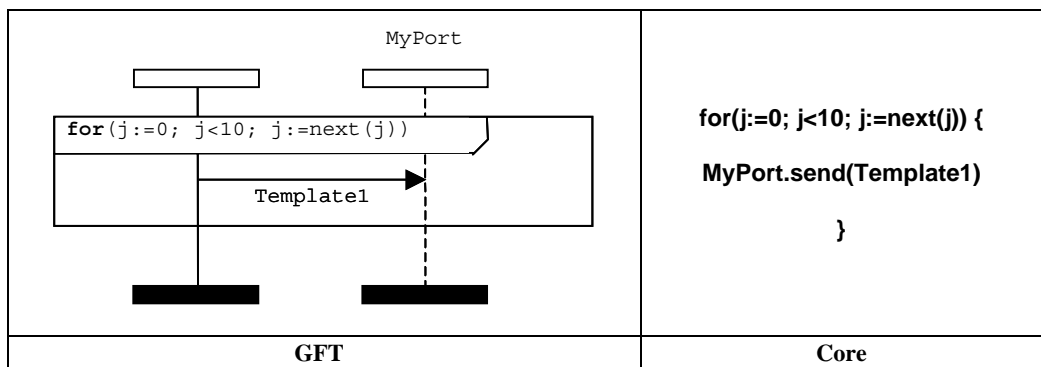


**Figure 16 – Invocation of user defined function**



**Figure 17 – Invocation of predefined/external function**

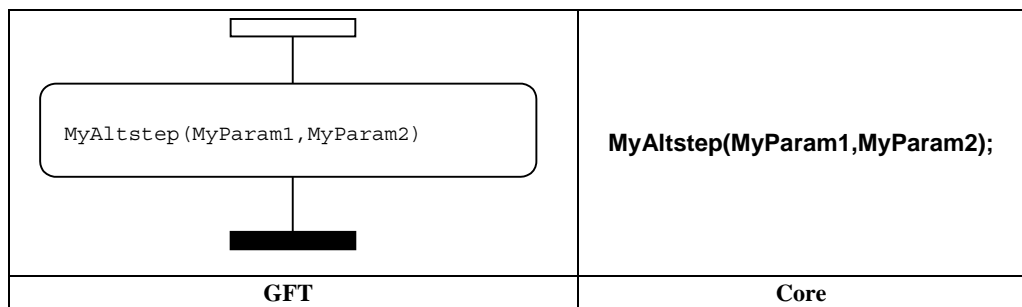
Functions called inside a TTCN-3 construct with an associated GFT symbol are represented as text within that symbol.



**Figure 18 – Invocation of user defined function within GFT symbol**

### 11.2.3 Invocation of altsteps

The invocation of altsteps is represented by use of the reference symbol (see Figure 19). The syntax of the altstep invocation is placed within that symbol. The symbol may contain the invocation of an altstep with optional parameters. It shall be used within alternative behaviour only, where the altstep invocation shall be one of the operands of the alternative statements (see also Figure 32 in clause 11.5.2).



**Figure 19 – Altstep invocation**

Another possibility is the implicit invocation of altsteps via activated defaults. Please refer to clause 11.6.2 for further details.

## 11.3 Declarations

TTCN-3 allows the declaration and initialization of timers, constants and variables at the beginning of statement blocks. GFT uses the syntax of the TTCN-3 core language for declarations in several symbols. The type of a symbol depends on the specification of the initialization, e.g., a variable of type `default` that is initialized by means of an `activate` operation shall be specified within a default symbol (see clause 11.6).

### 11.3.1 Declaration of timers, constants and variables in action symbols

The following declarations shall be made within action symbols:

- timer declarations;
- declarations of variables without initialization;
- declarations of variables and constants with initialization;

- if the initialization is not made by means of functions that include communication functions; or
- if a declaration is:
  - of a component type that is not initialized by means of a `create` operation;
  - of type `default` that is not initialized by means of an `activate` operation;
  - of type `verdicttype` that is not initialized by means of an `execute` statement;
  - of a simple basic type;
  - of a basic string type;
  - of the type `anytype`;
  - of a port type;
  - of the type `address`; or
  - of a user-defined structured type with fields that fulfil all restrictions mentioned in this bullet for "declarations of variables and constants with initialization".

NOTE – Please refer to Table 3 of [ITU-T Z.161] for an overview on TTCN-3 types.

A sequence of declarations that shall be made within action symbols can be put into one action symbol and need not to be made in separate action symbols. Examples for declarations within action symbols can be found in Figures 20 a) and 20 b).

### **11.3.2 Declaration of constants and variables within inline expression symbols**

Constants and variable declarations of a component type that are initialized within an `if-else`, `for`, `while`, `do-while`, `alt` or `interleave` statement shall be presented within the same inline expression symbol.

### **11.3.3 Declaration of constants and variables within create symbols**

Constants and variable declarations of a component type that are initialized by means of `create` operations shall be made within a create symbol. In contrast to declarations within action symbols, each declaration that is initialized by means of a `create` operation shall be presented in a separate create symbol. An example for a variable declaration within a create symbol is shown in Figure 20 c).

### **11.3.4 Declaration of constants and variables within default symbols**

Constants and variable declarations of type `default` that are initialized by means of `activate` operations shall be made within a default symbol. In contrast to declarations within action symbols, each declaration that is initialized by means of an `activate` operation shall be presented in a separate default symbol. An example for a variable declaration within a default symbol is shown in Figure 20 d).

### **11.3.5 Declaration of constants and variables within reference symbols**

Constants and variable declarations that are initialized by means of a function, which includes communication operations, shall be made within reference symbols. In contrast to declarations within action symbols, each declaration that is initialized by means of a function, which includes communication functions, shall be presented in a separate reference symbol. An example for a variable declaration within a reference symbol is shown in Figure 20 e).

### 11.3.6 Declaration of constants and variables within execute test case symbols

Constants and variable declarations of type `verdicttype` that are initialized by means of `execute` statements shall be made within execute test case symbols. In contrast to declarations within action symbols, each declaration that is initialized by means of an `execute` statement shall be presented in a separate execute test case symbol. An example for a variable declaration within an execute test case symbol is shown in Figure 20 f).

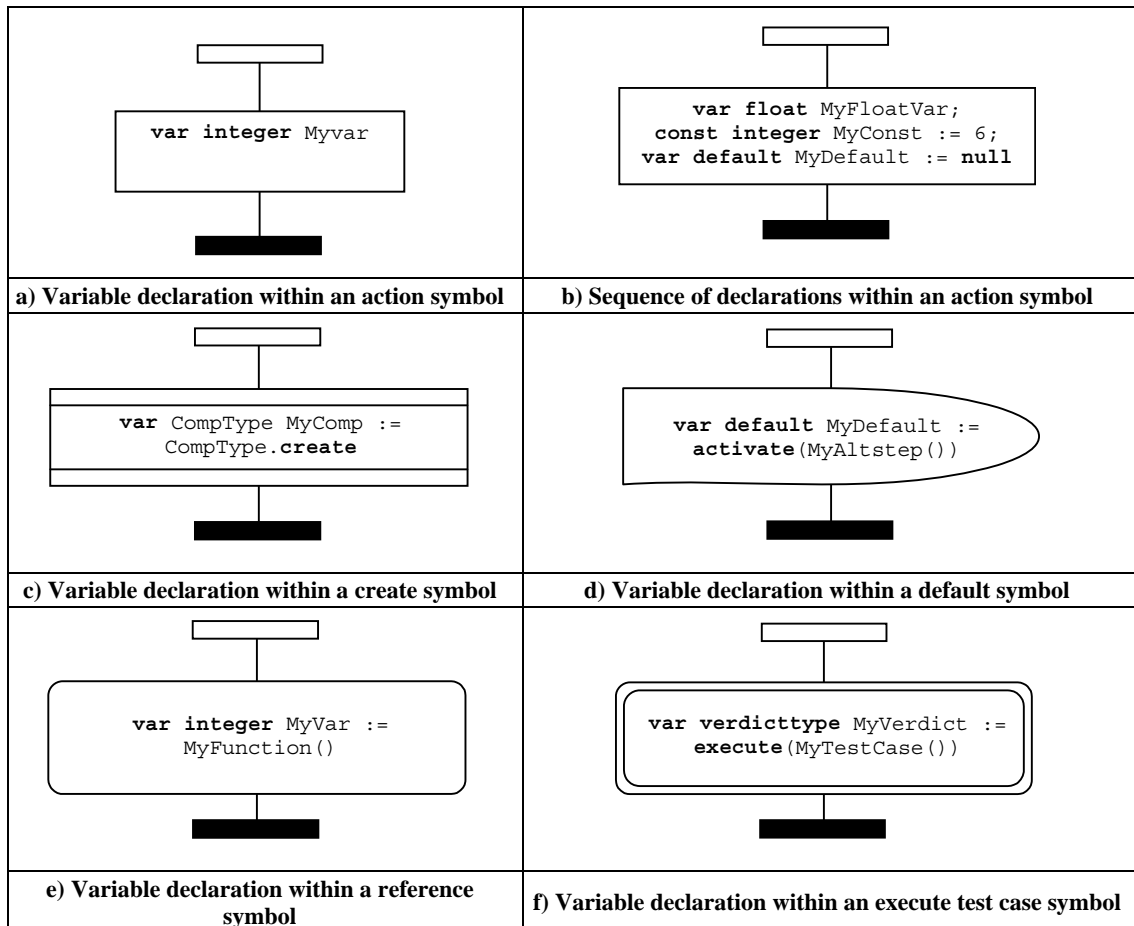


Figure 20 – Examples for declarations in GFT

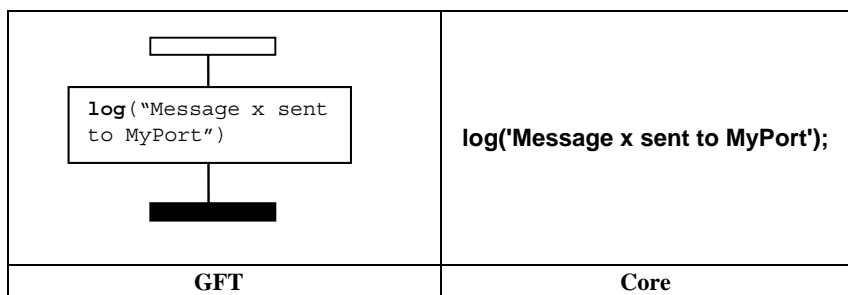
### 11.4 Basic program statements

Basic program statements are expressions, assignments, operations, loop constructs, etc. All basic program statements can be used within GFT diagrams for the control part, test cases, functions and altsteps.

GFT does not provide any graphical representation for expressions and assignments. They are textually denoted at the places of their use. Graphics is provided for the `log`, `label`, `goto`, `if-else`, `for`, `while` and `do-while` statement.

### 11.4.1 The Log statement

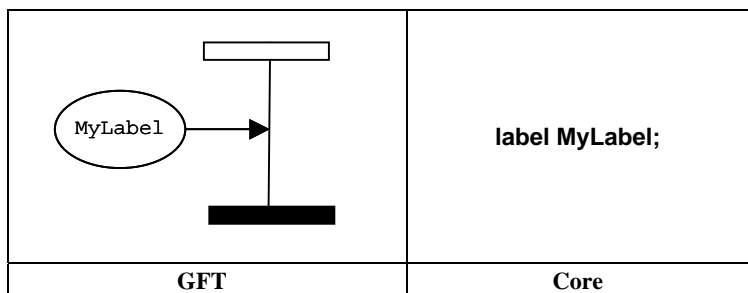
The `log` statement shall be represented within an action symbol (see Figure 21).



**Figure 21 – Log statement**

### 11.4.2 The Label statement

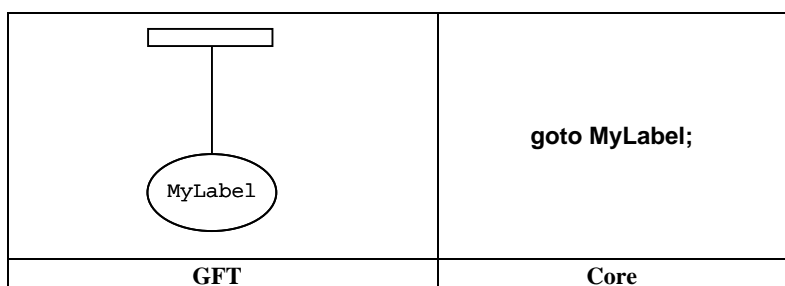
The `label` statement shall be represented with a label symbol, which is connected to a component instance. Figure 22 illustrates a simple example of a `label` named `MyLabel`.



**Figure 22 – Label statement**

### 11.4.3 The Goto statement

The `goto` statement shall be represented with a goto symbol. It shall be placed at the end of a component instance or at the end of an operand in an inline expression symbol. Figure 23 illustrates a simple example of a `goto`.



**Figure 23 – Goto statement**

### 11.4.4 The If-else statement

The `if-else` statement shall be represented by an inline expression symbol labelled with the `if` keyword and a Boolean expression as defined in clause 19.6 of [ITU-T Z.161]. The if-else inline expression symbol may contain one or two operands, separated by a dashed line. Figure 24 illustrates an `if` statement with a single operand, which is executed when the Boolean expression

$x > 1$  evaluates to true. Figure 25 illustrates an **if-else** statement in which the top operand is executed when the Boolean expression  $x > 1$  evaluates to true, and the bottom operand is executed if the Boolean expression evaluates to false.

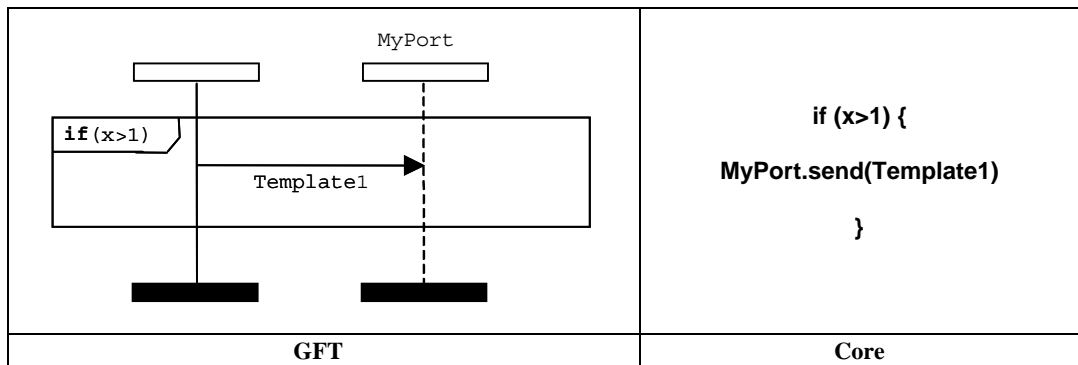


Figure 24 – If-statement

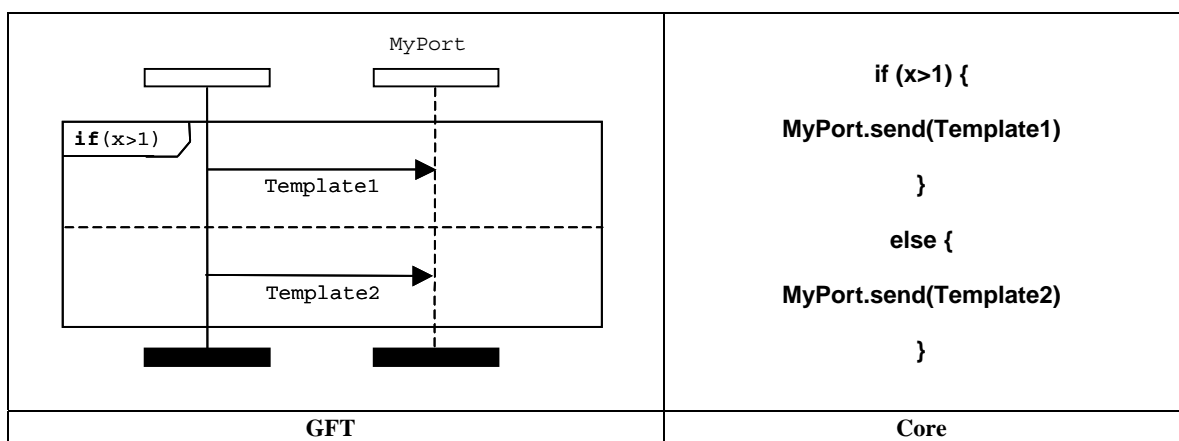


Figure 25 – If-else statement

#### 11.4.5 The For statement

The **for** statement shall be represented by an inline expression symbol labelled with a **for** definition as defined in clause 19.7 of [ITU-T Z.161]. The **for** body shall be represented as the operand of the for inline expression symbol. Figure 26 represents a simple **for** loop in which the loop variable is declared and initialized within the **for** statement.

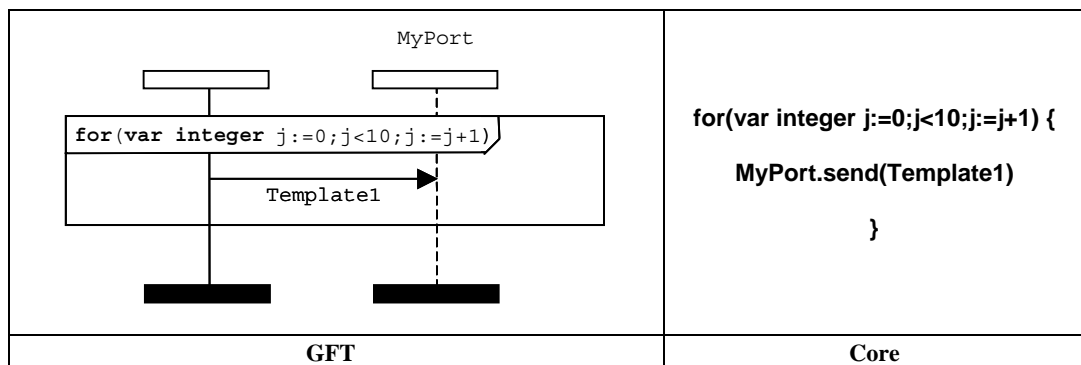


Figure 26 – For statement

### 11.4.6 The While statement

The `while` symbol shall be represented by an inline expression symbol labelled with a `while` definition as defined in clause 19.8 of [ITU-T Z.161]. The `while` body shall be represented as the operand of the while inline expression symbol. Figure 27 represents an example of a `while` statement.

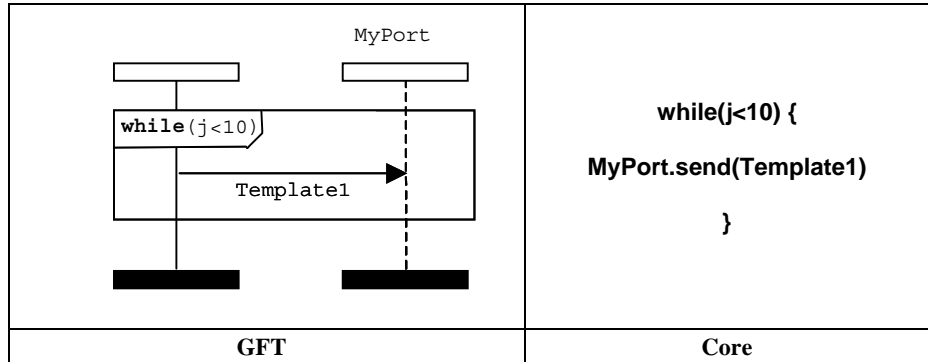


Figure 27 – While statement

### 11.4.7 The Do-while statement

The `do-while` statement shall be represented by an inline expression symbol labelled with a `do-while` definition as defined in clause 19.9 of [ITU-T Z.161]. The `do-while` body shall be represented as the operand of the do-while inline expression symbol. Figure 28 represents an example of a `do-while` statement.

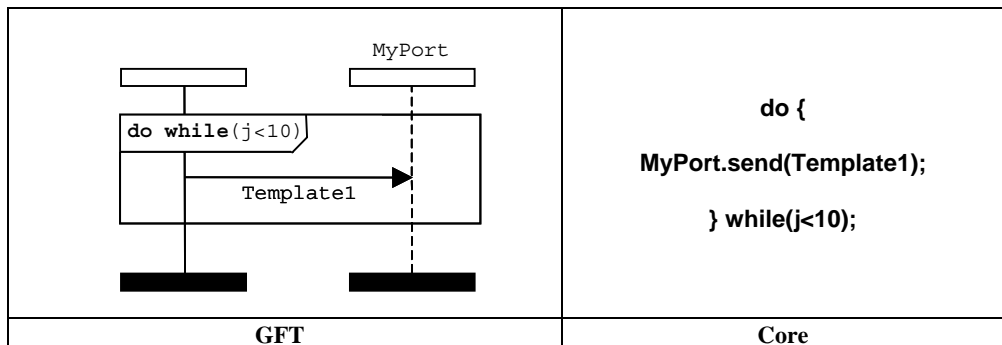


Figure 28 – Do-while statement

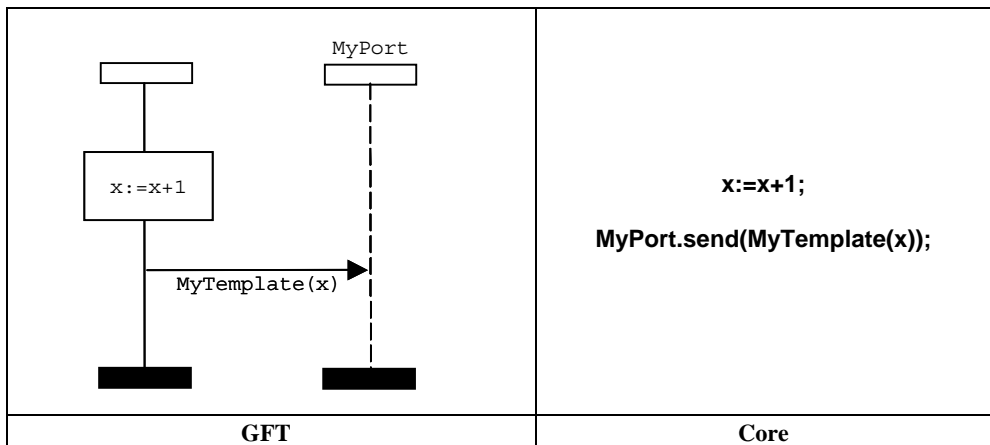
## 11.5 Behavioural program statements

Behavioural statements may be used within test cases, functions and altsteps, the only exception being the return statement, which can only be used within functions. Test behaviour can be expressed sequentially, as a set of alternatives or using an interleaving statement. Return and repeat are used to control the flow of behaviour.

### 11.5.1 Sequential behaviour

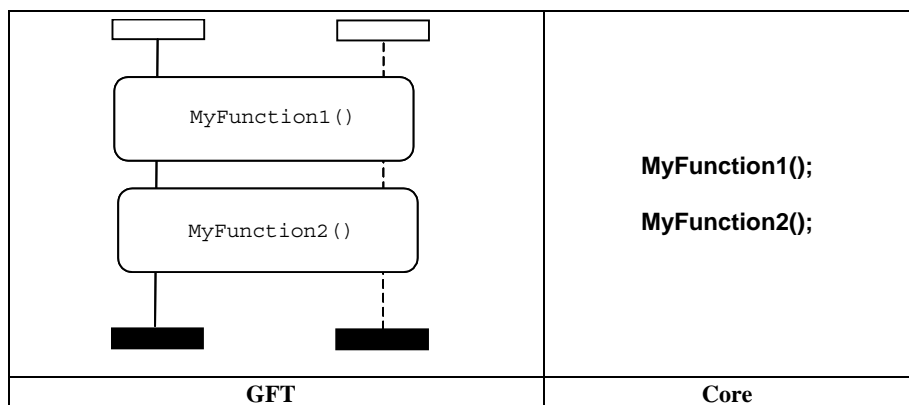
Sequential behaviour is represented by the order of events placed upon a test component instance. The ordering of events is taken in a top-down manner, with events placed nearest the top of the component instance symbol being evaluated first. Figure 29 illustrates a simple case in which the test component firstly evaluates the expression contained within the action symbol and then sends a message to a port `MyPort`.





**Figure 29 – Sequential behaviour**

Sequencing can also be described using references to test cases, functions, and altsteps. In this case, the order in which references are placed upon a component instance axis determines the order in which they are evaluated. Figure 30 represents a simple GFT diagram in which `MyFunction1` is called, followed by `MyFunction2`.

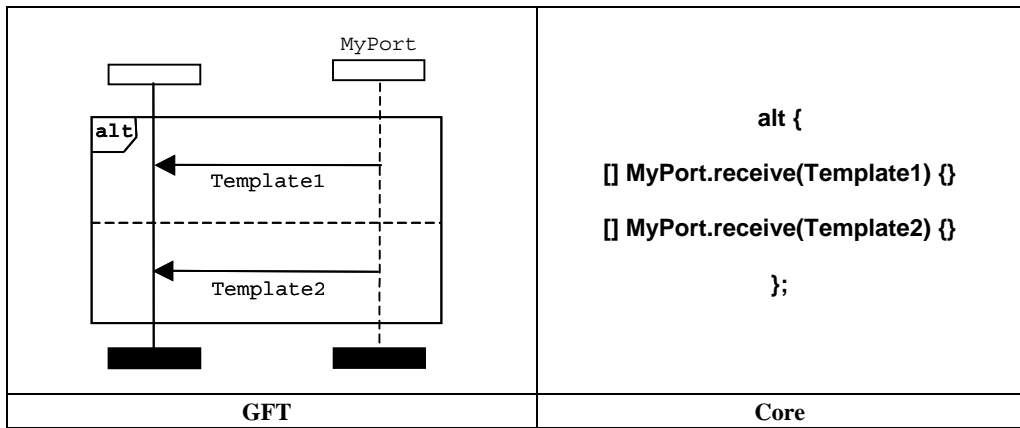


**Figure 30 – Sequencing using references**

### 11.5.2 Alternative behaviour

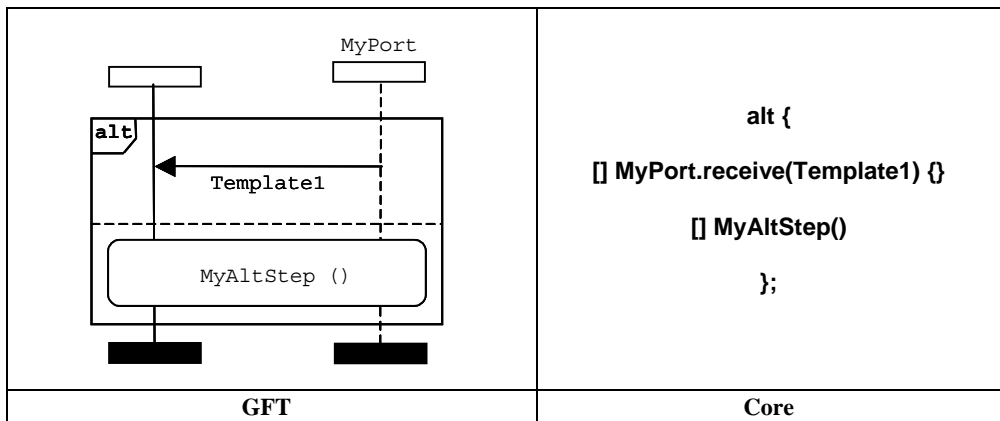
Alternative behaviour shall be represented using inline expression symbol with the `alt` keyword placed in the top left hand corner. Each operand of the alternative behaviour shall be separated using a dashed line. Operands are evaluated top-down.

Note that an alternative inline expression should always cover all port instances, if communication operators are involved. Figure 31 illustrates an alternative behaviour in which either a message event is received with the value defined by `Template1`, or a message event is received with the value defined by `Template2`. The invocation of an altstep in an alternative inline expression is shown in Figure 32.



**Figure 31 – Alternative behaviour statement**

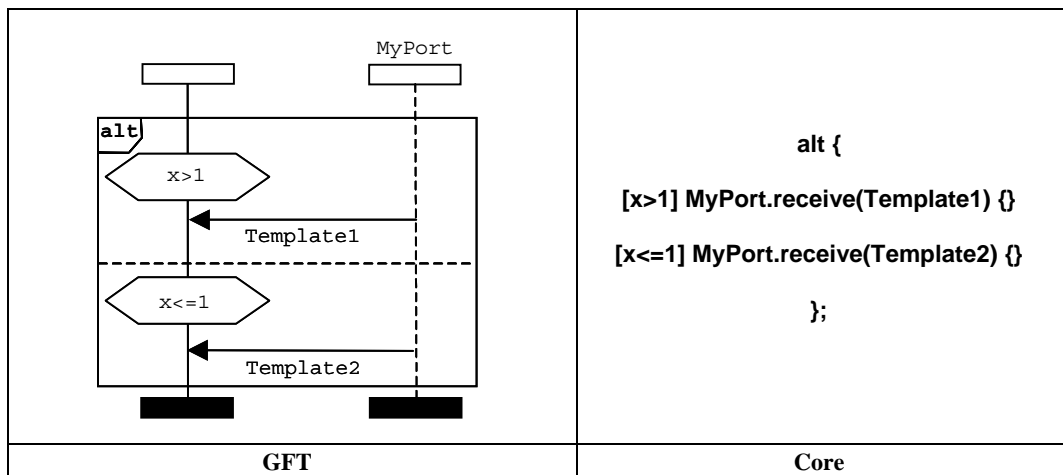
In addition, it is possible to call an altstep as the only event within an alternative operand. This shall be drawn using a reference symbol (see clause 11.2.3).



**Figure 32 – Alternative behaviour with altstep invocation**

### 11.5.2.1 Selecting/deselecting an alternative

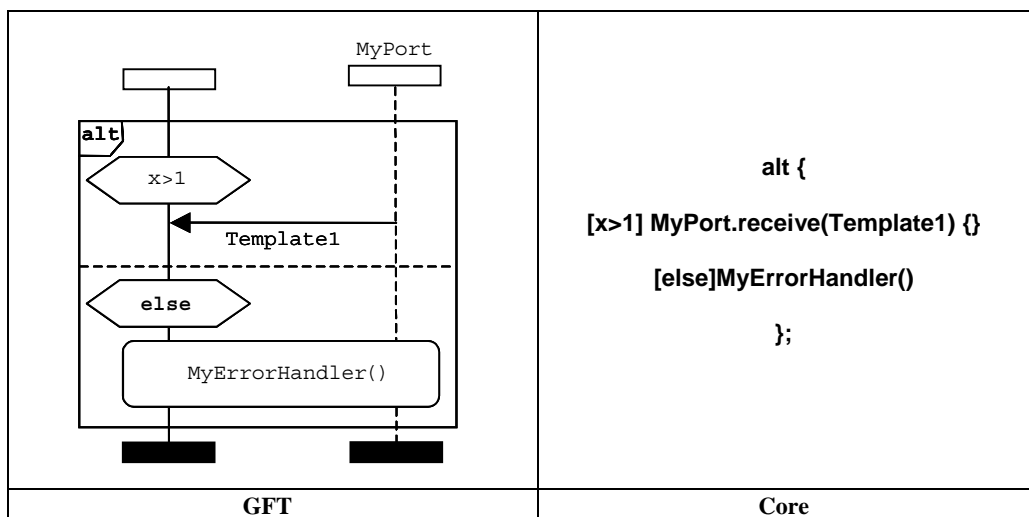
It is possible to disable/enable an alternative operand by means of a Boolean expression contained within a condition symbol placed upon the test component instance. Figure 33 illustrates a simple alternative statement in which the first operand is guarded with the expression  $x > 1$ , and the second with the expression  $x \leq 1$ .



**Figure 33 – Selecting/deselecting an alternative**

### 11.5.2.2 Else branch in alternatives

The `else` branch shall be denoted using a condition symbol placed upon the test component instance axis labelled with the `else` keyword. Figure 34 illustrates a simple alternative statement where the second operand represents an `else` branch.



**Figure 34 – Else within an alternative**

Note that the reference symbol within an `else` branch should always cover all port instances, if communication operations are involved.

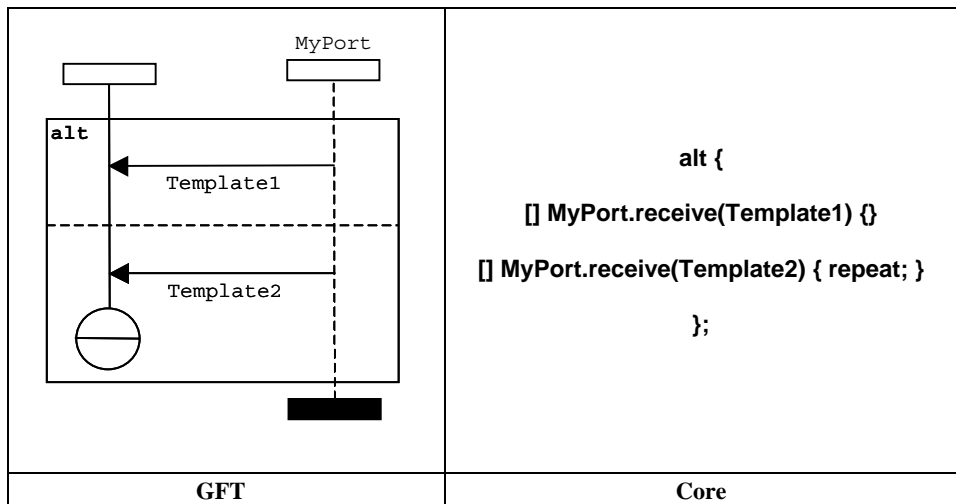
The re-evaluation of an `alt` statement can be specified using a repeat statement, which is represented by the repeat symbol (see clause 11.5.3).

The invocation of `altsteps` within alternatives is represented using the reference symbol (see clause 11.2.3).

### 11.5.3 The Repeat statement

The `repeat` statement shall be represented by a repeat symbol. This symbol shall only be used as last event of an alternative operand in an `alt` statement or as last event of an operand of the top alternative in an `altstep` definition. Figure 35 illustrates an alternative statement in which the second

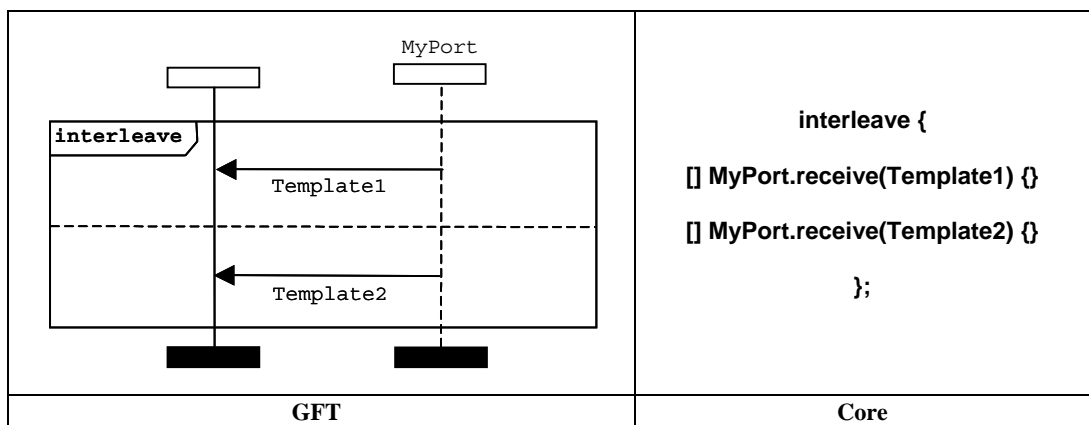
operand, having successfully received a message with a value matching `Template2`, causes the alternative to be repeated.



**Figure 35 – Repeat within an alternative**

#### 11.5.4 Interleaved behaviour

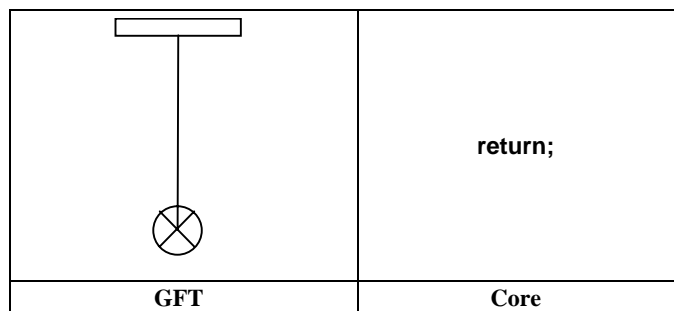
Interleave behaviour shall be represented using an inline expression symbol with the `interleave` keyword placed in the top left hand corner (see Figure 36). Each operand shall be separated using a dashed line. Operands are evaluated in a top-down order.



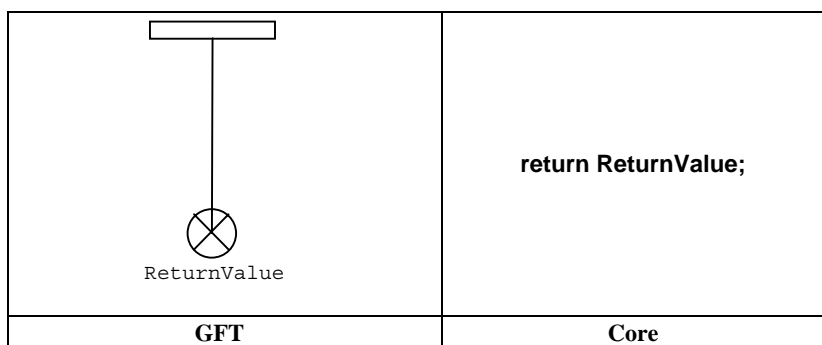
**Figure 36 – Interleave statement**

#### 11.5.5 The Return statement

The `return` statement shall be represented by a return symbol. This may be optionally associated with a return value. A return symbol shall only be used in a GFT function diagram. It shall only be used as last event of a component instance or as last event of an operand in an inline expression symbol. Figure 37 illustrates a simple function using a return statement without a returning a value, and Figure 38 illustrates a function that returns a value.



**Figure 37 – Return symbol without a return value**



**Figure 38 – Return symbol with a return value**

## 11.6 Default handling

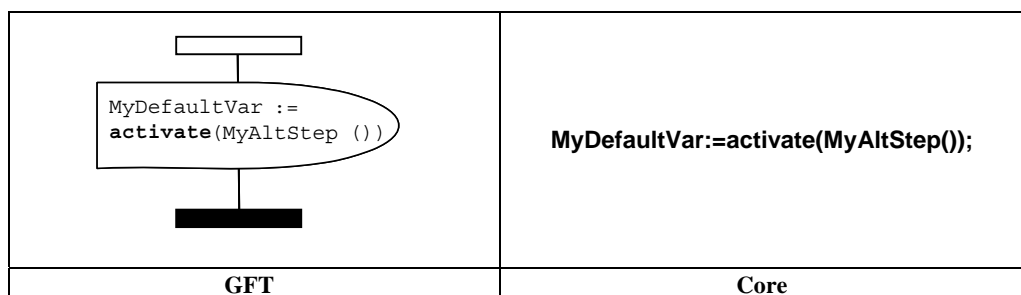
GFT provides graphical representation for the activation and deactivation of defaults (see clause 21 of [ITU-T Z.161]).

### 11.6.1 Default references

Variables of type `default` can either be declared within an action symbol or within a default symbol as part of an activate statement. Clauses 11.3.1 and 11.3.4 illustrate how a variable called `MyDefaultType` is declared within GFT.

### 11.6.2 The activate operation

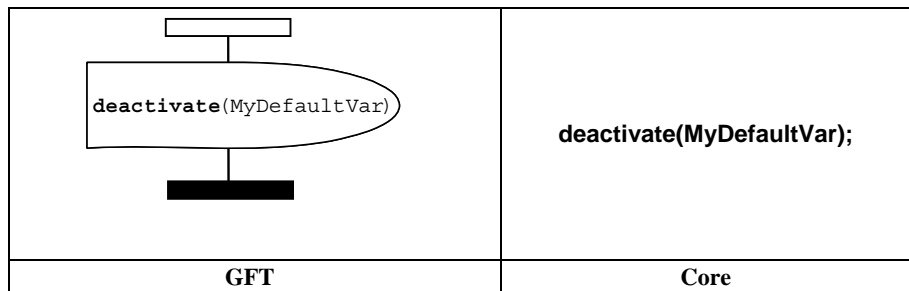
The activation of defaults shall be represented by the placement of the `activate` statement within a default symbol (see Figure 39).



**Figure 39 – Default activation**

### 11.6.3 The deactivate operation

The deactivation of defaults shall be represented by the placement of the `deactivate` statement within a default symbol (see Figure 40). If no operands are given to the `deactivate` statement then all defaults are deactivated.



**Figure 40 – Deactivation of defaults**

## 11.7 Configuration operations

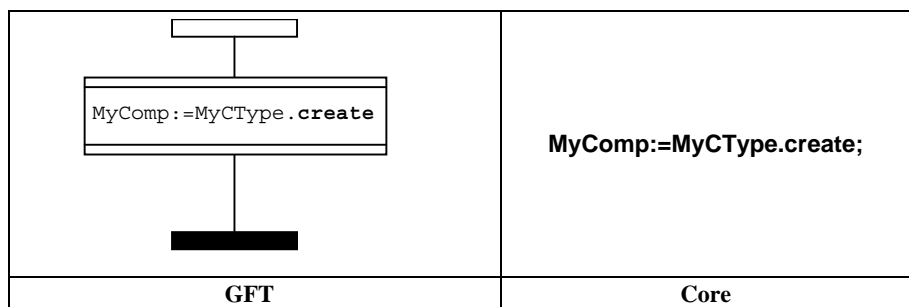
Configuration operations are used to set up and control test components. These operations shall only be used in GFT test case, function, and altstep diagrams.

The `mtc`, `self`, and `system` operations have no graphical representation; they are textually denoted at the places of their use.

GFT does not provide any graphical representation for the running operation (being a Boolean expression). It is textually denoted at the place where it is used.

### 11.7.1 The Create operation

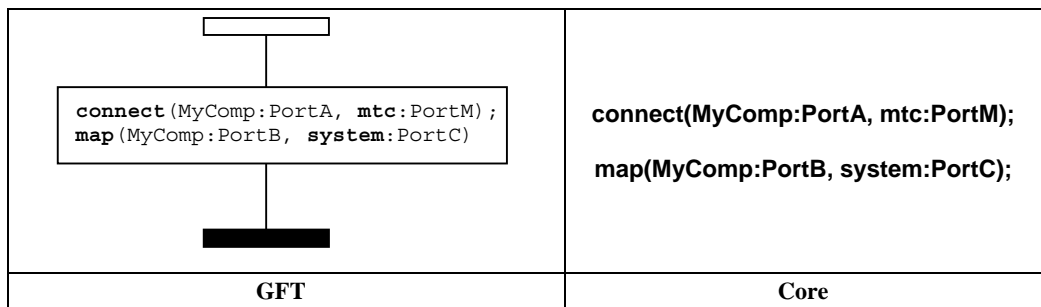
The `create` operation shall be represented within the create symbol, which is attached to the test component instance which performs the create operation (see Figure 41). The create symbol contains the create statement.



**Figure 41 – Create operation**

### 11.7.2 The Connect and Map operations

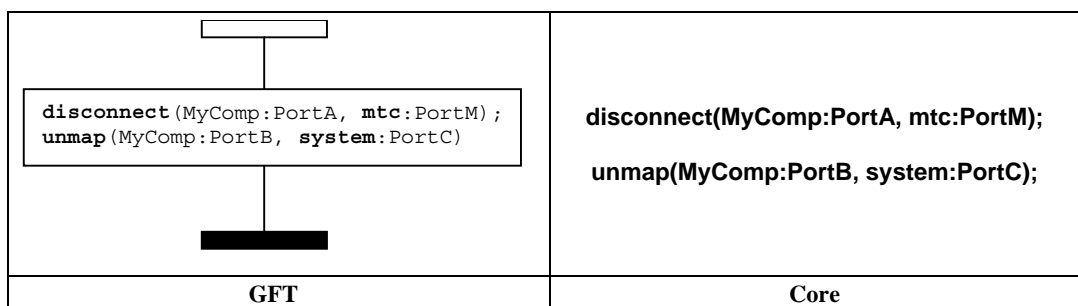
The `connect` and `map` operations shall be represented within an action box symbol, which is attached to the test component instance which performs the `connect` or `map` operation (see Figure 42). The action box symbol contains the `connect` or `map` statement.



**Figure 42 – Connect and map operation**

### 11.7.3 The Disconnect and Unmap operations

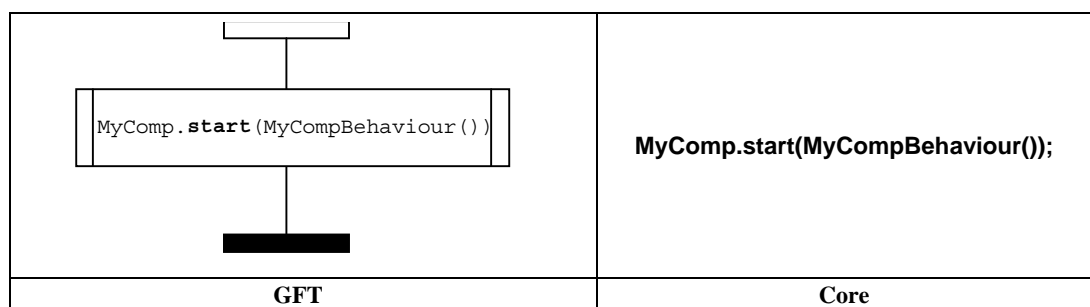
The `disconnect` and `unmap` operations shall be represented within an action box symbol, which is attached to the test component instance which performs the `disconnect` or `unmap` operation (see Figure 43). The action box symbol contains the `disconnect` or `unmap` statement.



**Figure 43 – Disconnect and unmap operation**

### 11.7.4 The Start test component operation

The `start` test component operation shall be represented within the `start` symbol, which is attached to the test component instance that performs the `start` operation (see Figure 44). The start symbol contains the `start` statement.

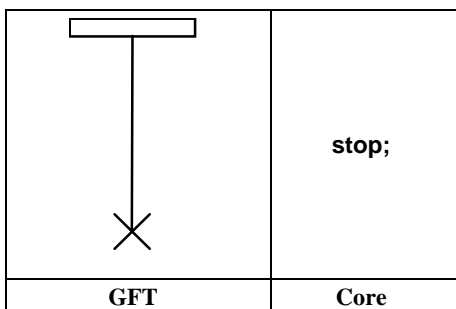


**Figure 44 – Start operation**

### 11.7.5 The Stop execution and Stop test component operations

TTCN-3 has two stop operations: The module control and test components may stop themselves by using *stop execution operations*, or a test component can stop other test components by using *stop test component operations*.

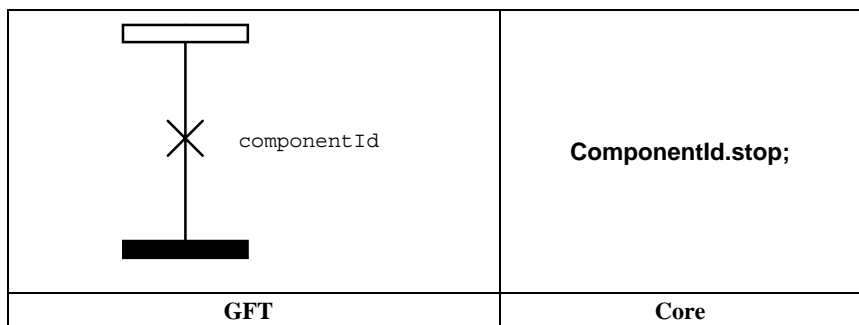
The **stop** execution operation shall be represented by a stop symbol, which is attached to the test component instance, which performs the **stop** execution operation (see Figure 45). It shall only be used as last event of a component instance or as last event of an operand in an inline expression symbol.



**Figure 45 – Stop execution operation**

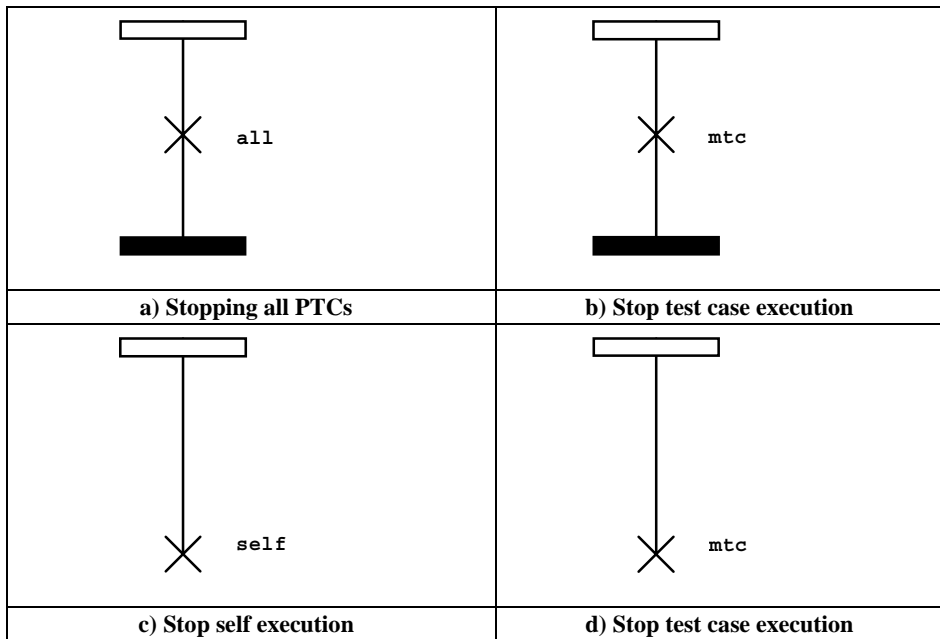
The **stop** test component operation shall be represented by a stop symbol, which is attached to the test component instance, which performs the **stop** test component operation. It shall have an associated expression that identifies the component to be stopped (see Figure 46). The MTC may stop all PTCs in one step by using the stop component operation with the keyword **all** (see Figure 47 a)). A PTC can stop the test execution by stopping the MTC (see Figure 47 b)). The **stop** test component operation shall be used as last event of a component instance or 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**) (see Figures 47 c) and d)).

NOTE – The stop symbol has an associated expression. It is not always possible to determine statically, if a stop component operation stops the instance that executes the stop operation or stops the test execution.



**Figure 46 – Stop test component operation**

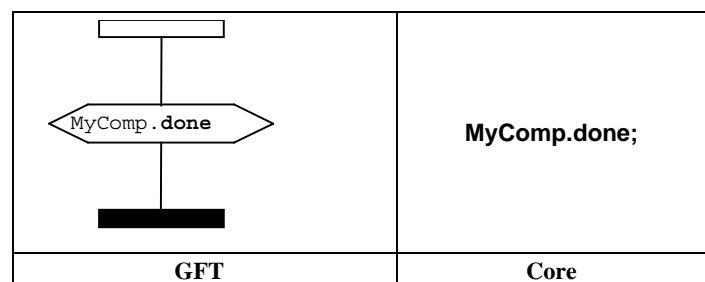




**Figure 47 – Special usages of the stop test component operation**

### 11.7.6 The Done operation

The `done` operation shall be represented within a condition symbol, which is attached to the test component instance, which performs the `done` operation (see Figure 48). The condition symbol contains the `done` statement.



**Figure 48 – Done operation**

The `any` and `all` keywords can be used for the `running` and `done` operations but from the MTC instance only. They have no graphical representation, but are textually denoted at the places of their use.

### 11.8 Communication operations

Communication operations are structured into two groups:

- a) *Sending operations*: a test component sends a message (`send` operation), calls a procedure (`call` operation), replies to an accepted call (`reply` operation) or raises an exception (`raise` operation).
- b) *Receiving operations*: a component receives a message (`receive` operation), accepts a procedure call (`getcall` operation), receives a reply for a previously called procedure (`getreply` operation) or catches an exception (`catch` operation).

### 11.8.1 General format of the sending operations

All sending operations use a message symbol that is drawn from the test component instance performing the sending operation to the port instance to which the information is transmitted (see Figure 49).

Sending operations consist of a *send* part and, in the case of a blocking procedure-based `call` operation, a *response* and *exception handling* part.

The send part:

- specifies the port at which the specified operation shall take place;
- defines the optional type and value of the information to be transmitted;
- gives an optional address expression that uniquely identifies the communication partner in the case of a one-to-many connection.

The port shall be represented by a port instance. The operation name for the `call`, `reply`, and `raise` operations shall be denoted on top of the message symbol in front of the optional type information. The `send` operation is implicit, i.e., the `send` keyword shall not be denoted. The value of the information to be transmitted shall be placed underneath the message symbol. The optional address expression (denoted by the `to` keyword) shall be placed underneath the value of the information to be transmitted.

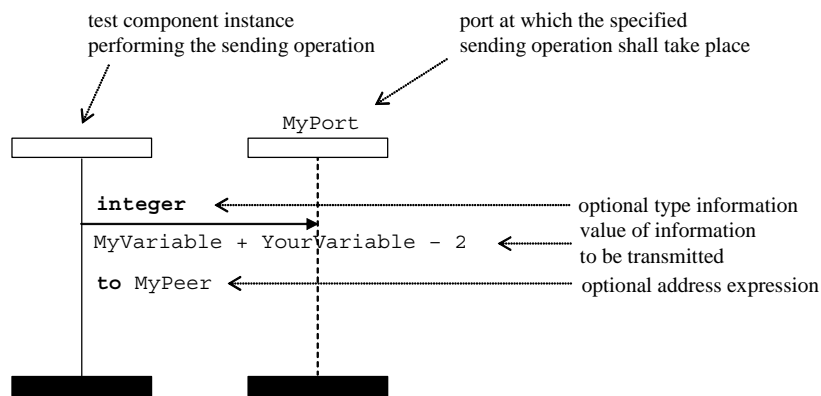


Figure 49 – General format of sending operations

The structure of the `call` operation is more specific. Please refer to clause 11.8.4.1 for further details.

### 11.8.2 General format of the receiving operations

All receiving operations use a message symbol drawn from the port instance to the test component instance receiving the information (see Figure 50).

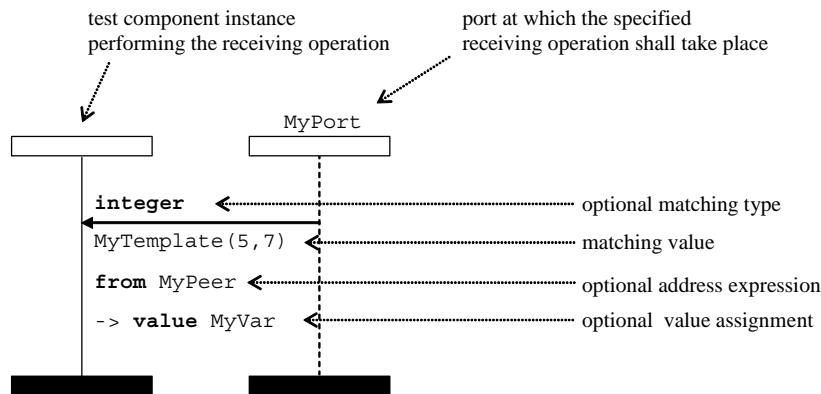
A receiving operation consists of a *receive* part and an optional *assignment* part.

The receive part:

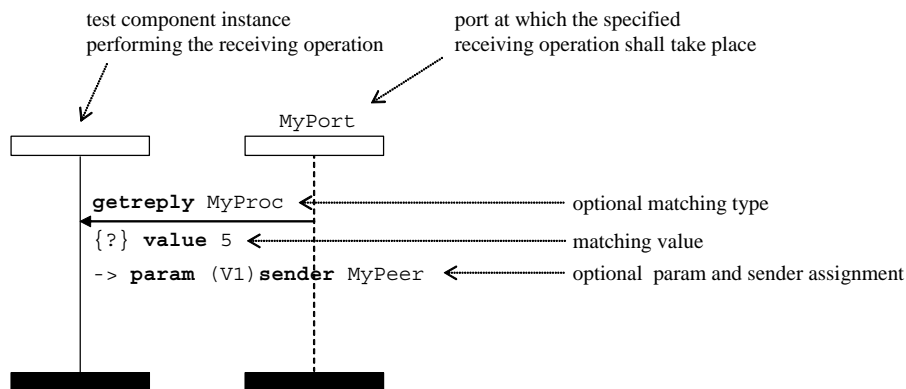
- a) specifies the port at which the operation shall take place;
- b) defines a matching part consisting of an optional type information and the matching value which specifies the acceptable input which will match the statement;
- c) gives an (optional) address expression that uniquely identifies the communication partner (in case of one-to-many connections).

The port shall be represented by a port instance. The operation name for the `getcall`, `getreply`, and `catch` operations shall be denoted on top of the message symbol in front of (optional) type information. The `receive` operation is given implicitly, i.e., the `receive` keyword shall not be denoted. The matching value for the acceptable input shall be placed underneath the message symbol. The (optional) address expression (denoted by the `from` keyword) shall be placed underneath the value of the information to be transmitted.

The (optional) assignment part (denoted by the `'->'`) shall be placed underneath the value of the information to be transmitted or if present underneath the address expression. It may be split over several lines, for example to have the value, parameter and sender assignment each on individual lines (see Figure 51).



**Figure 50 – General format of receiving operations with address and value assignment**

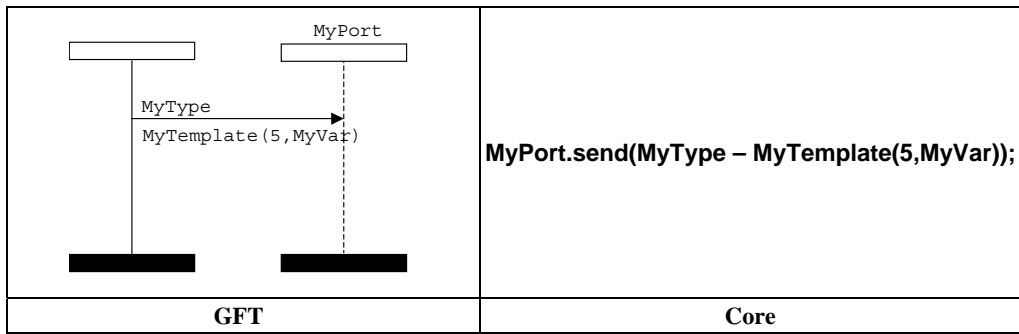


**Figure 51 – General format of receiving operations with param and sender assignment**

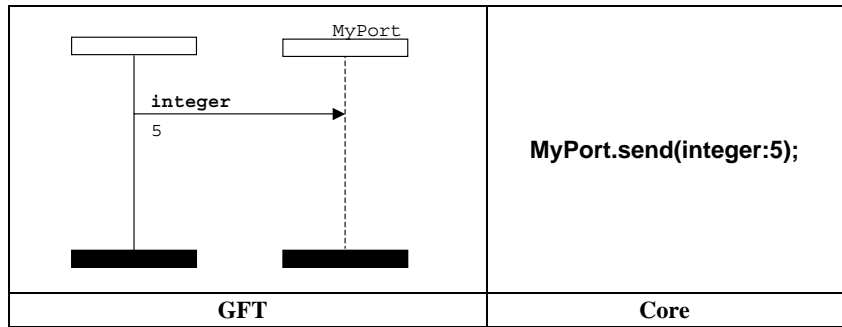
### 11.8.3 Message-based communication

#### 11.8.3.1 The Send operation

The send operation shall be represented by an outgoing message symbol from the test component to the port instance. The optional type information shall be placed above the message arrow. The (inline) template shall be placed underneath the message arrow (see Figures 52 and 53).



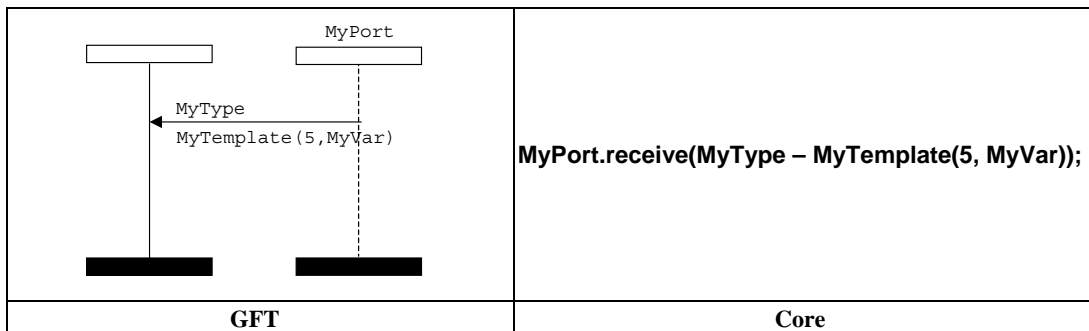
**Figure 52 – Send operation with template reference**



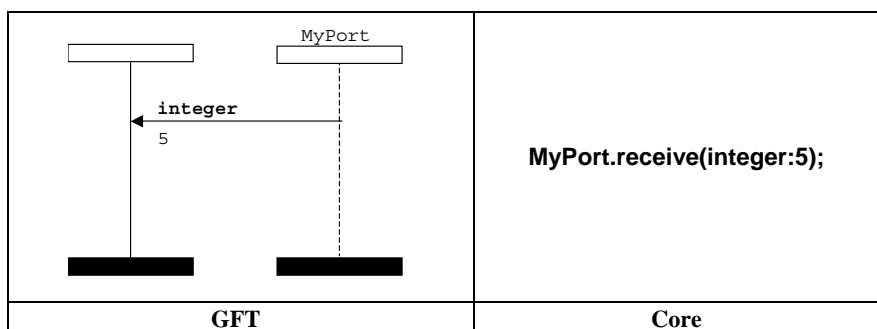
**Figure 53 – Send operation with inline template**

### 11.8.3.2 The Receive operation

The receive operation shall be represented by an incoming message arrow from the port instance to the test component. The optional type information shall be placed above the message arrow. The (inline) template shall be placed underneath the message arrow (see Figures 54 and 55).



**Figure 54 – Receive operation with template reference**



**Figure 55 – Receive operation with inline template**

### 11.8.3.2.1 Receive any message

The receive any message operation shall be represented by an incoming message arrow from the port instance to the test component without any further information attached to it (see Figure 56).

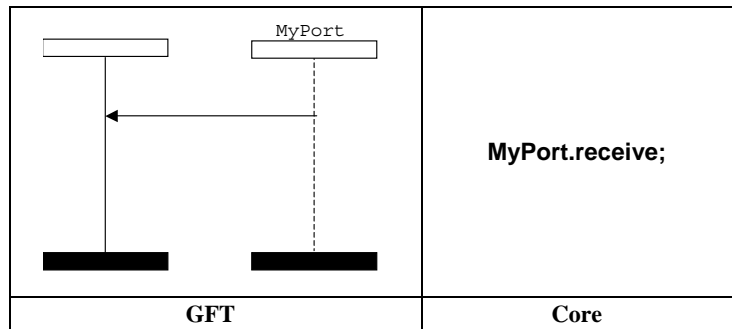


Figure 56 – Receive any message

### 11.8.3.2.2 Receive on any port

The receive on any port operation shall be represented by a found symbol representing any port to the test component (see Figure 57).

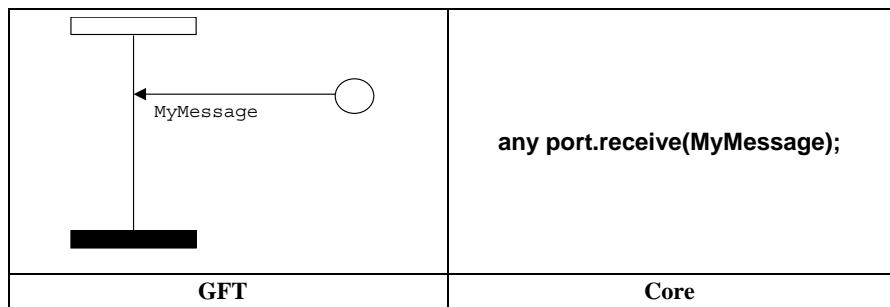


Figure 57 – Receive on any port

### 11.8.3.3 The Trigger operation

The trigger operation shall be represented by an incoming message arrow from the port instance to the test component and the keyword `trigger` above the message arrow preceding the type information if present. The optional type information is placed above the message arrow subsequent to the keyword `trigger`. The (inline) template is placed underneath the message arrow (see Figures 58 and 59).

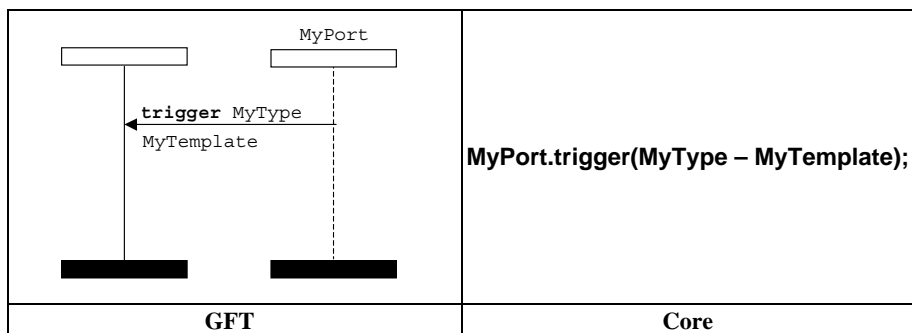
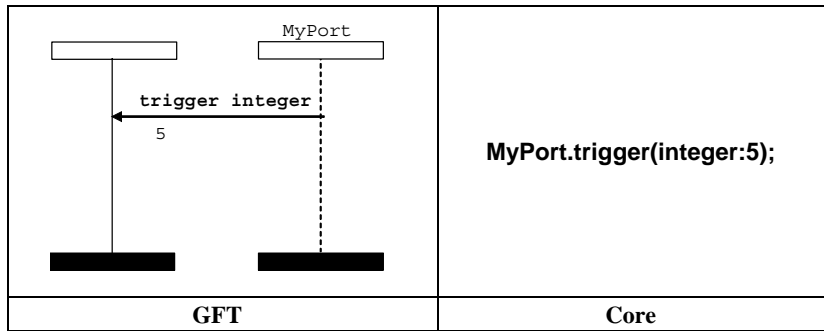


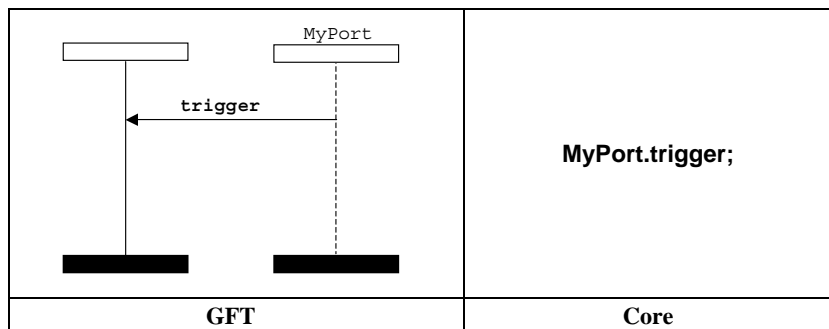
Figure 58 – Trigger operation with template reference



**Figure 59 – Trigger operation with inline template**

### 11.8.3.3.1 Trigger on any message

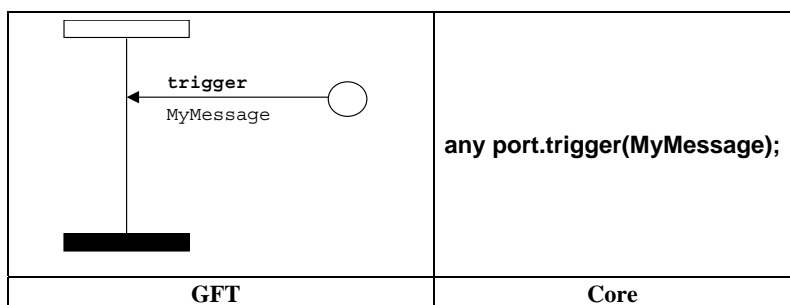
The trigger on any message operation shall be represented by an incoming message arrow from the port instance to the test component and the keyword `trigger` above the message arrow without any further information attached to it (see Figure 60).



**Figure 60 – Trigger on any message operation**

### 11.8.3.3.2 Trigger on any port

The trigger on any port operation shall be represented by a found symbol representing any port to the test component (see Figure 61).



**Figure 61 – Trigger on any port operation**

## 11.8.4 Procedure-based communication

### 11.8.4.1 The Call operation

#### 11.8.4.1.1 Calling blocking procedures

The blocking `call` operation is represented by an outgoing message symbol from the test component to the port instance with a subsequent suspension region on the test component and the keyword `call` above the message arrow preceding the signature if present. The (inline) template is placed underneath the message arrow (see Figures 62 and 63).

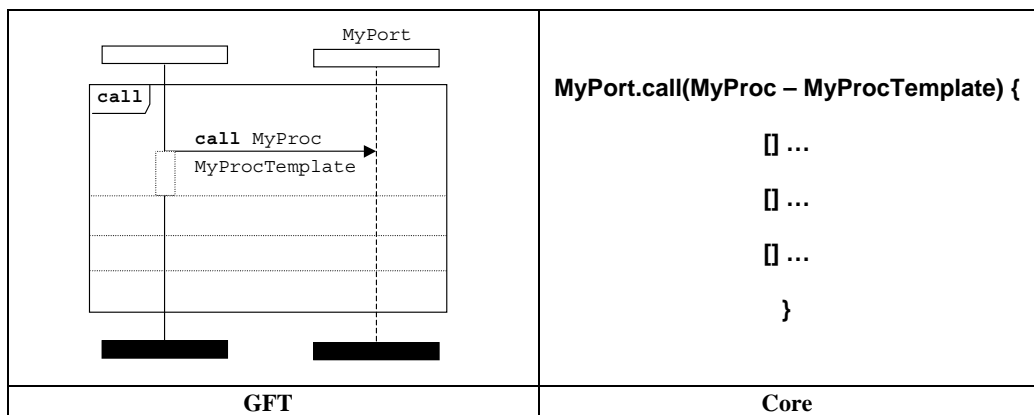


Figure 62 – Blocking call operation with template reference

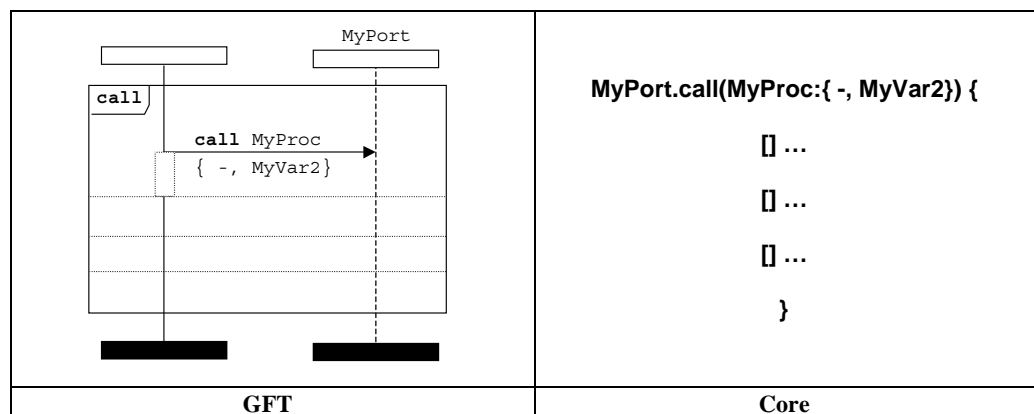
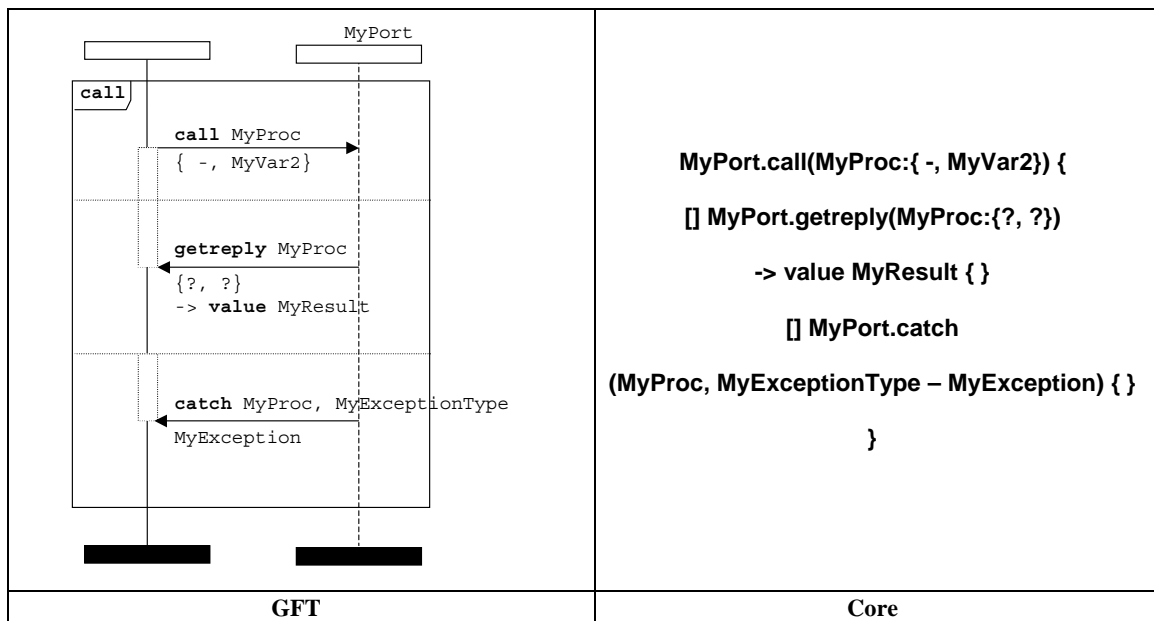


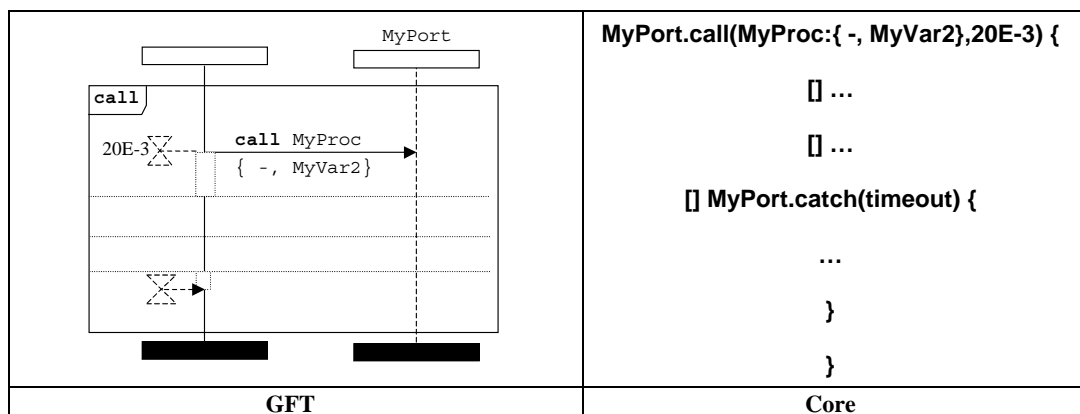
Figure 63 – Blocking call operation with inline template

The call inline expression is introduced in order to facilitate the specification of the alternatives of the possible responses to the blocking call operation. The call operation may be followed by alternatives of `getreply`, `catch` and `timeout`. The responses to a call are specified within the call inline expression following the call operation separated by dashed lines (see Figure 64).



**Figure 64 – Blocking call operation followed by alternatives of getreply and catch**

The call operation may optionally include a timeout. For that, the start implicit timer symbol is used to start this timing period. The timeout implicit timer symbol is used to represent the timeout exception (see Figure 65).

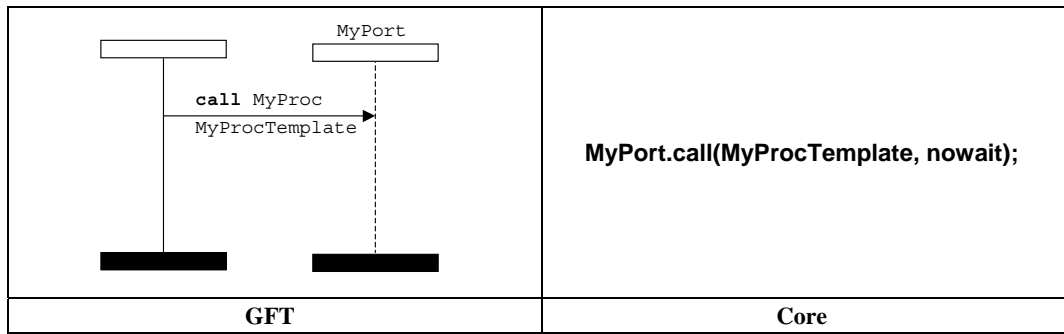


**Figure 65 – Blocking call operation followed by timeout exception**

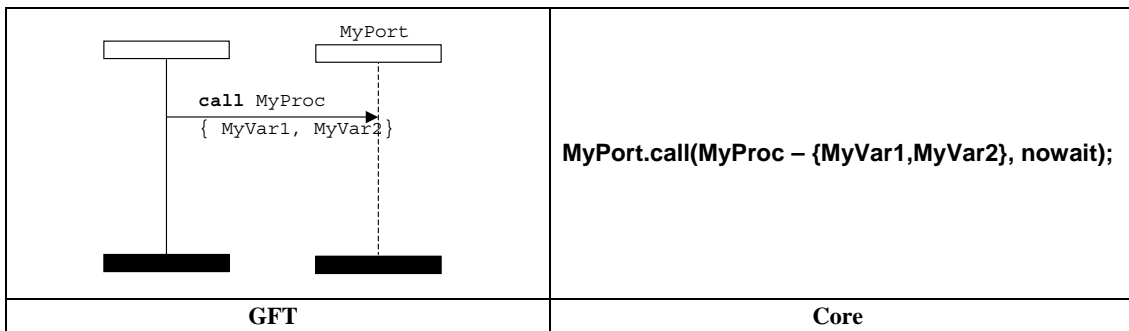
#### 11.8.4.1.2 Calling non-blocking procedures

The non-blocking call operation shall be represented by an outgoing message symbol from the test component to the port and the keyword `call` above the message arrow preceding the signature. There shall be no suspension region symbol attached to the message symbol. The optional signature is represented above the message arrow. The (inline) template is placed underneath the message arrow (see Figures 66 and 67).





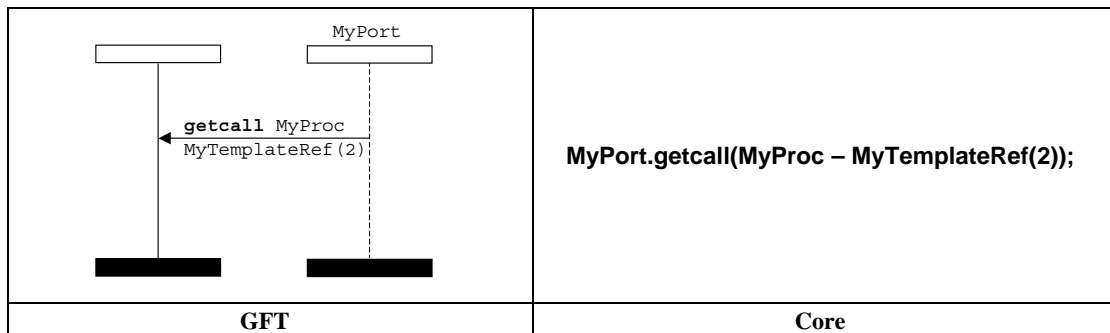
**Figure 66 – Non-blocking call operation with template reference**



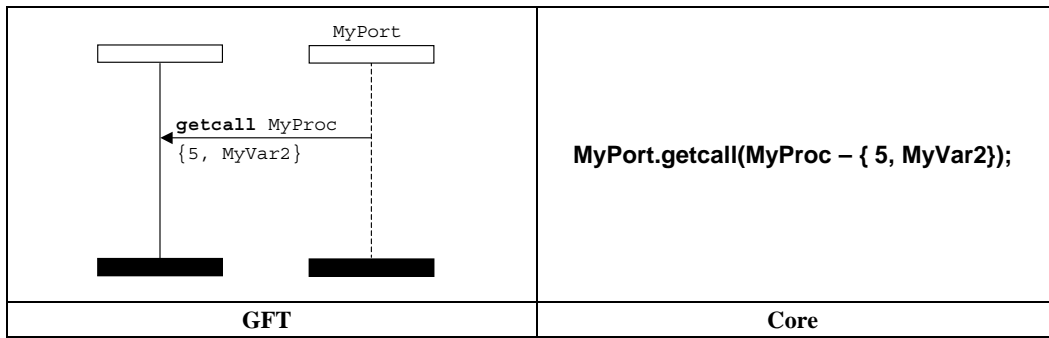
**Figure 67 – Non-blocking call operation with inline template**

#### 11.8.4.2 The Getcall operation

The getcall operation shall be represented by an incoming message arrow from the port instance to the test component and the keyword `getcall` above the message arrow preceding the signature. The signature is placed above the message arrow subsequent to the keyword `getcall`. The (inline) template is placed underneath the message arrow (see Figures 68 and 69).



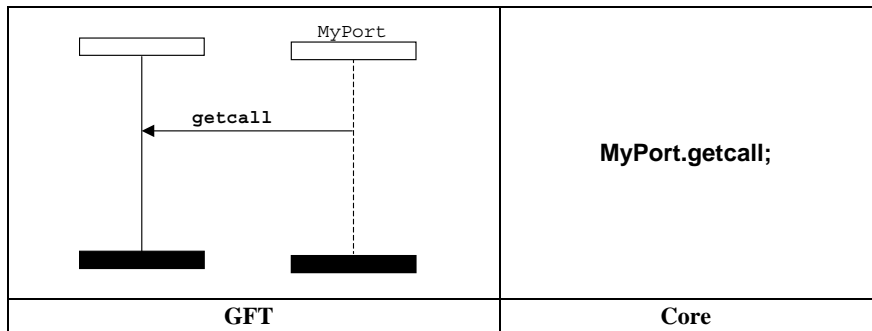
**Figure 68 – Getcall operation with template reference**



**Figure 69 – Getcall operation with inline template**

**11.8.4.2.1 Accepting any call**

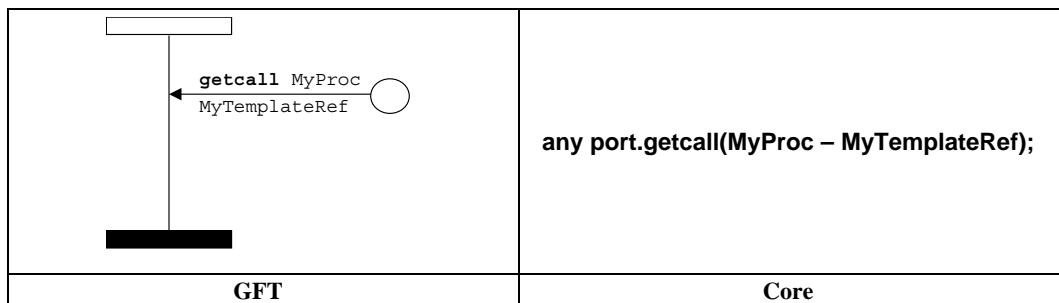
The accepting any call operation shall be represented by an incoming message arrow from the port instance to the test component and the keyword `getcall` above the message arrow. No further information shall be attached to the message symbol (see Figure 70).



**Figure 70 – Getcall on any call operation**

**11.8.4.2.2 Getcall on any port**

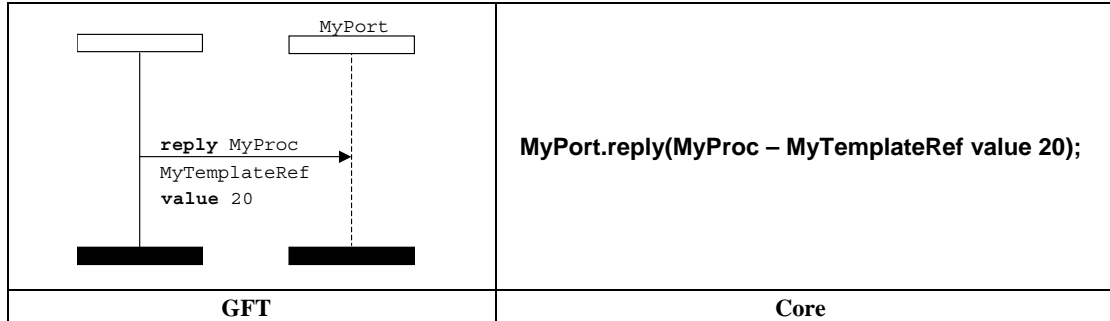
The getcall on any port operation is represented by a found symbol representing any port to the test component and the keyword `getcall` above the message arrow followed by the signature if present. The (inline) template if present shall be placed underneath the message arrow (see Figure 71).



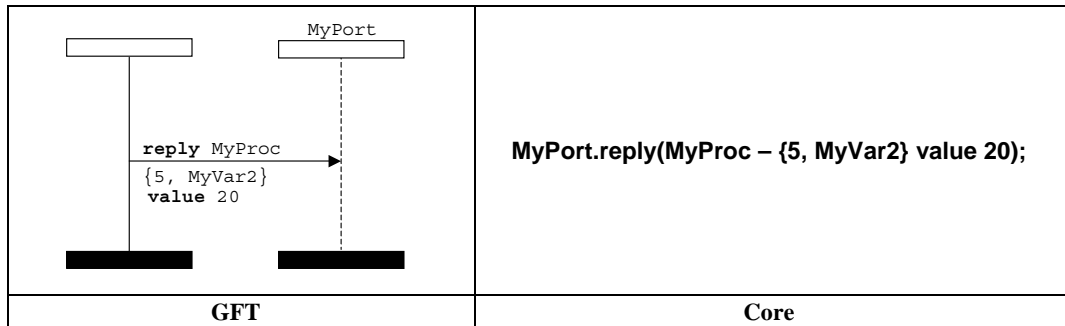
**Figure 71 – Getcall on any port operation with template reference**

### 11.8.4.3 The Reply operation

The reply operation shall be represented by an outgoing message arrow from the test component to the port instance and the keyword **reply** above the message arrow preceding the signature. The signature shall be placed above the message arrow subsequent to the keyword **reply**. The (inline) template shall be placed underneath the message arrow (see Figures 72 and 73).



**Figure 72 – Reply operation with template reference**

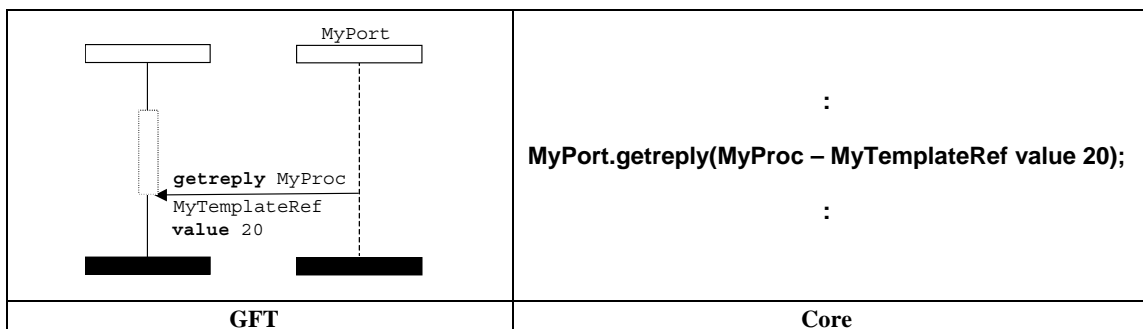


**Figure 73 – Reply operation with inline template**

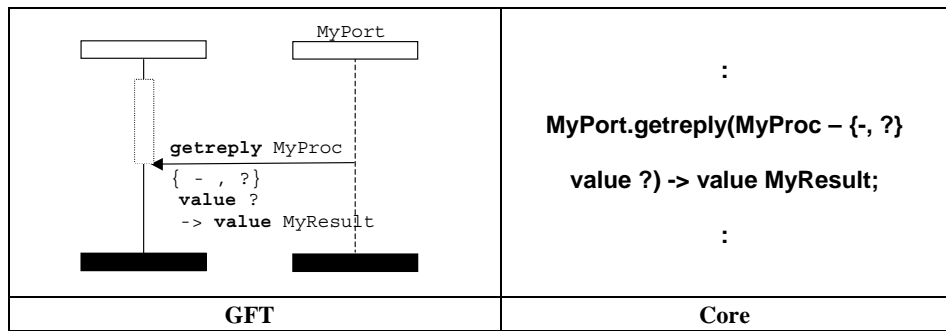
### 11.8.4.4 The Getreply operation

The getreply operation shall be represented by an incoming message arrow from the port instance to the test component and the keyword **getreply** above the message arrow preceding the signature. Within a call symbol, the message arrow head shall be attached to a preceding suspension region on the test component (see Figures 74 and 75). Outside a call symbol, the message arrow head shall not be attached to a preceding suspension region on the test component (see Figures 76 and 77).

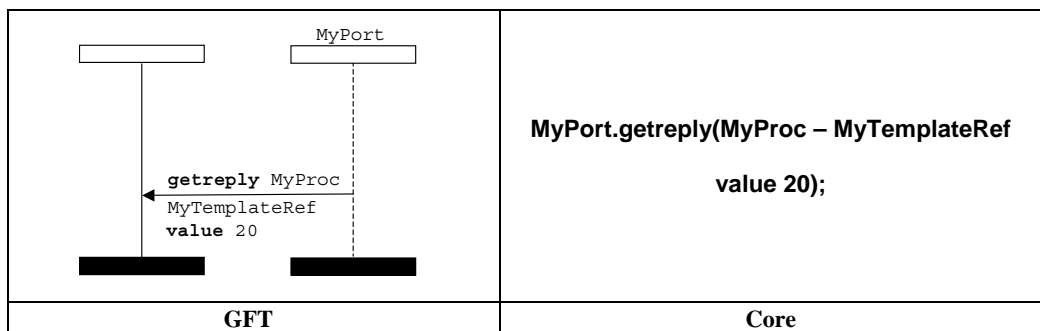
The signature shall be placed above the message arrow subsequent to the keyword **getreply**. The (inline) template shall be placed underneath the message arrow.



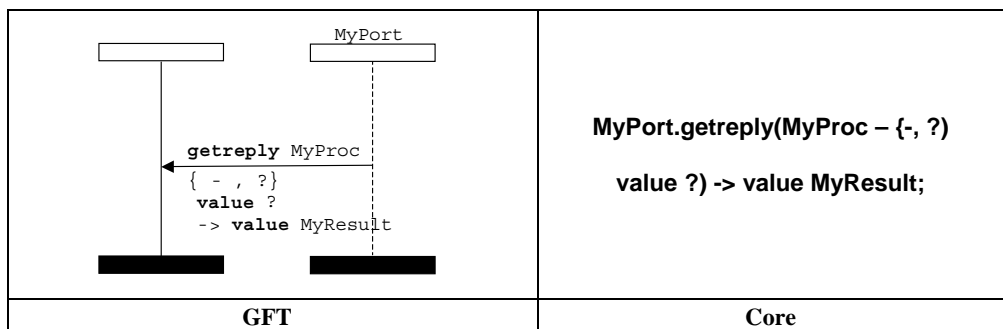
**Figure 74 – Getreply operation with template reference (within a call symbol)**



**Figure 75 – Getreply operation with inline template (within a call symbol)**



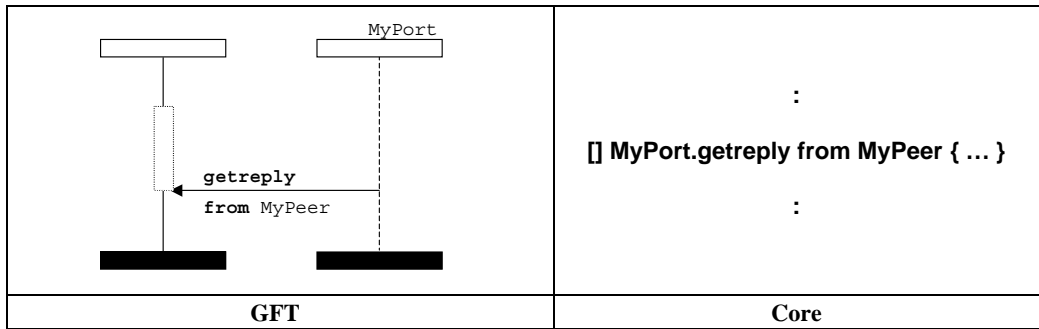
**Figure 76 – Getreply operation with template reference (outside a call symbol)**



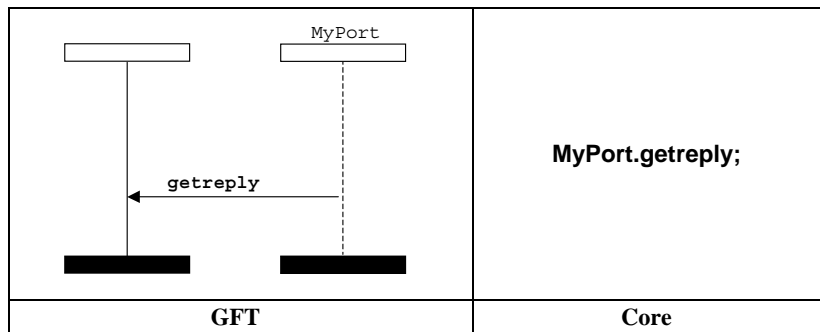
**Figure 77 – Getreply operation with inline template (outside a call symbol)**

#### 11.8.4.4.1 Get any reply from any call

The get any reply from any call operation shall be represented by an incoming message arrow from the port instance to the test component and the keyword `getreply` above the message. No signature shall follow the `getreply` keyword. Within a call symbol, the message arrow head shall be attached to a preceding suspension region on the test component (see Figure 78). Outside a call symbol, the message arrow head shall not be attached to a preceding suspension region on the test component (see Figure 79).



**Figure 78 – Get any reply from any call (within a call symbol)**

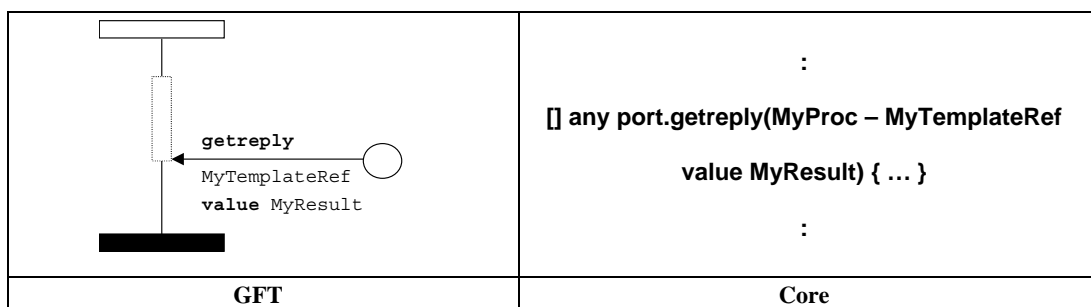


**Figure 79 – Getreply from any call (outside a call symbol)**

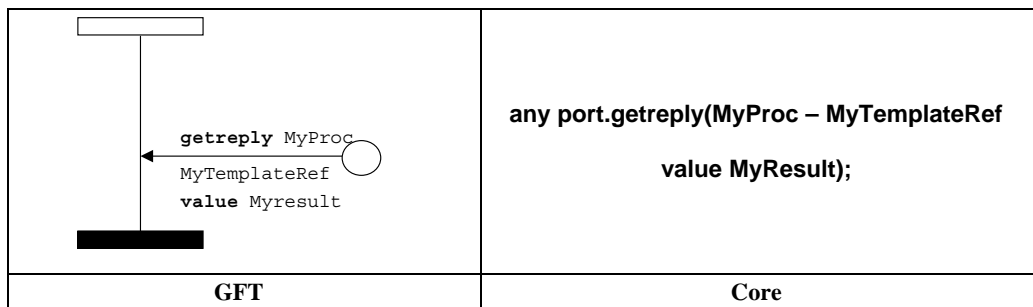
#### 11.8.4.4.2 Get a reply on any port

The get a reply on any port operation is represented by a found symbol representing any port to the test component. The keyword `getreply` shall be placed above the message arrow followed by the signature if present. Within a call symbol, the message arrow head shall be attached to a preceding suspension region on the test component (see Figure 80). Outside a call symbol, the message arrow head shall not be attached to a preceding suspension region on the test component (see Figure 81).

The signature if present shall be placed above the message arrow subsequent to the keyword `getreply`. The optional (inline) template is placed underneath the message arrow.



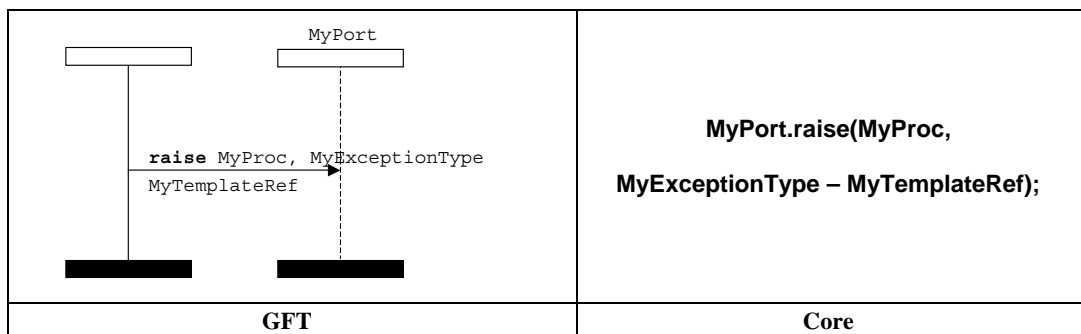
**Figure 80 – Get a reply on any port (within a call symbol)**



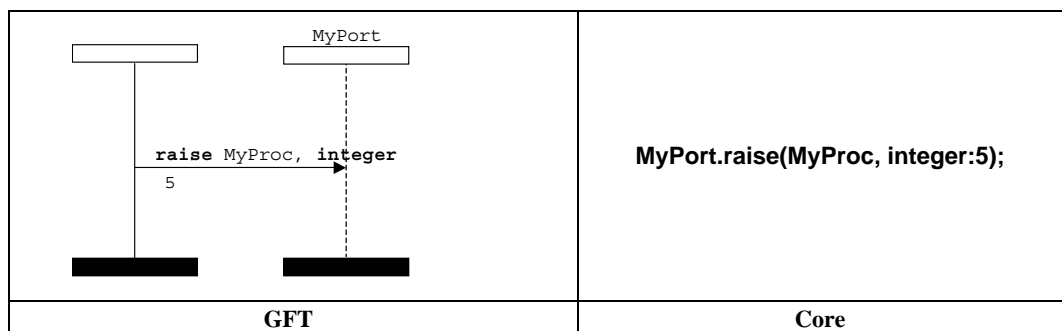
**Figure 81 – Get a reply on any port (outside a call symbol)**

#### 11.8.4.5 The Raise operation

The raise operation shall be represented by an outgoing message symbol from the test component to the port instance. The keyword **raise** shall be placed above the message arrow preceding the signature and the exception type, which are separated by a comma. The (inline) template shall be placed underneath the message arrow (see Figures 82 and 83).



**Figure 82 – Raise operation with template reference**

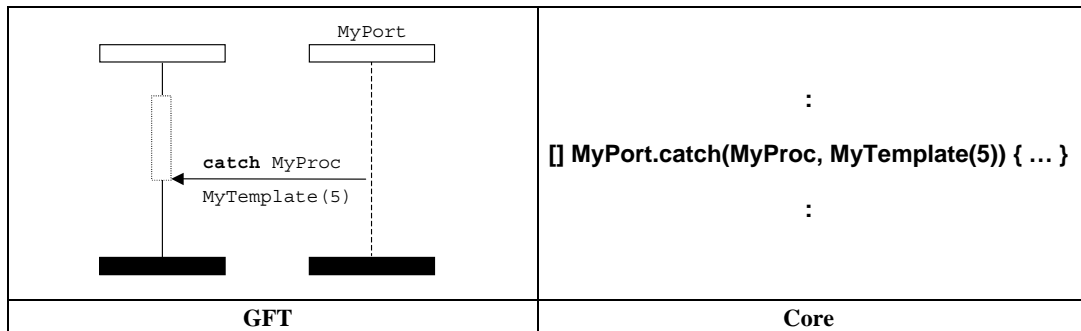


**Figure 83 – Raise operation with inline template**

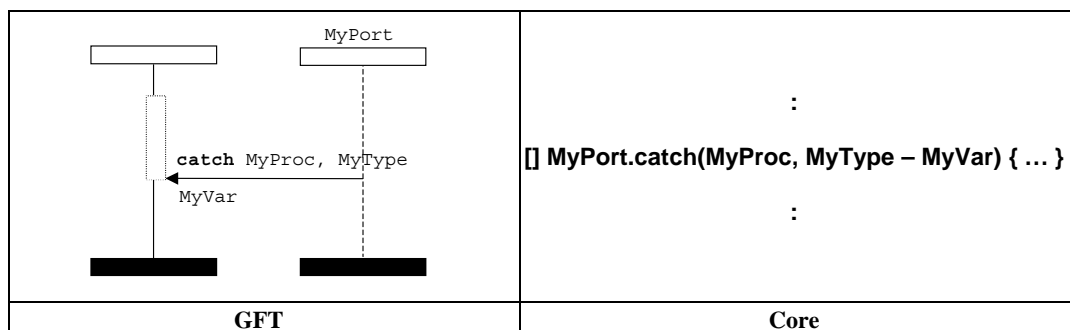
#### 11.8.4.6 The Catch operation

The catch operation shall be represented by an incoming message arrow from the port instance to the test component and the keyword **catch** above the message arrow preceding the signature and the exception type (if present). Within a call symbol, the message arrow head shall be attached to a preceding suspension region on the test component (see Figures 84 and 85). Outside a call symbol, the message arrow head shall not be attached to a preceding suspension region on the test component (see Figures 86 and 87).

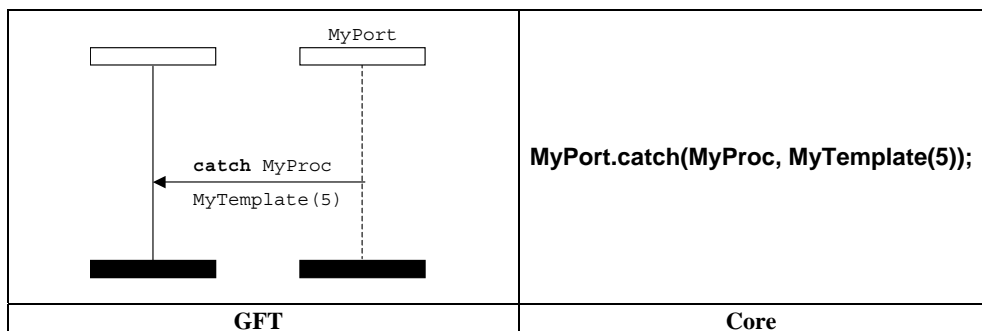
The signature and optional exception type information are placed above the message arrow subsequent to the keyword `catch` and are separated by a comma if the exception type is present. The (inline) template is placed underneath the message arrow.



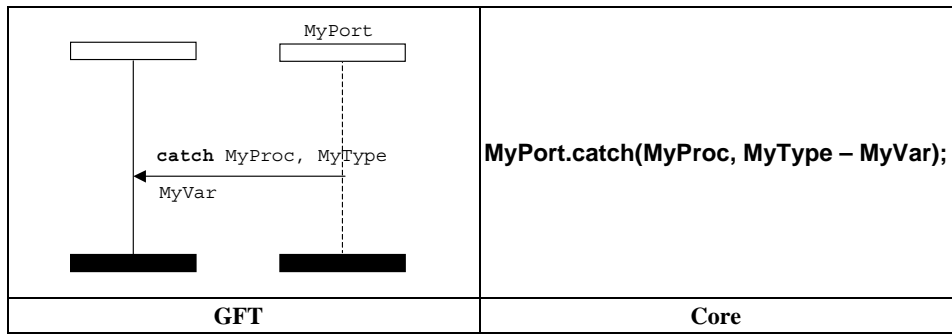
**Figure 84 – Catch operation with template reference (within a call symbol)**



**Figure 85 – Catch operation with inline template (within a call symbol)**



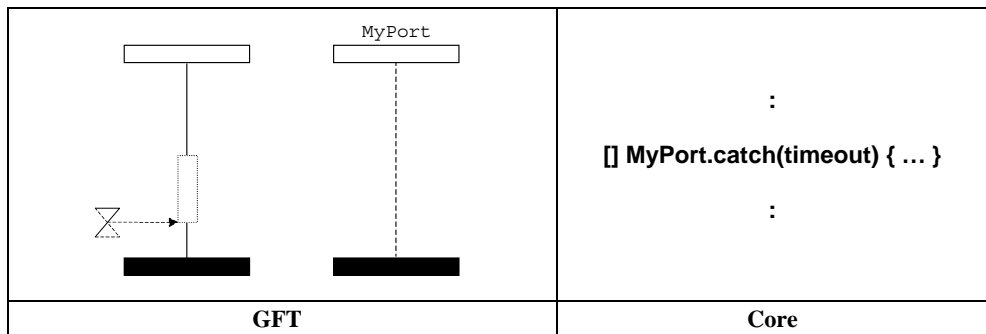
**Figure 86 – Catch operation with template reference (outside a call symbol)**



**Figure 87 – Catch operation with inline template (outside a call symbol)**

#### 11.8.4.6.1 The Timeout exception

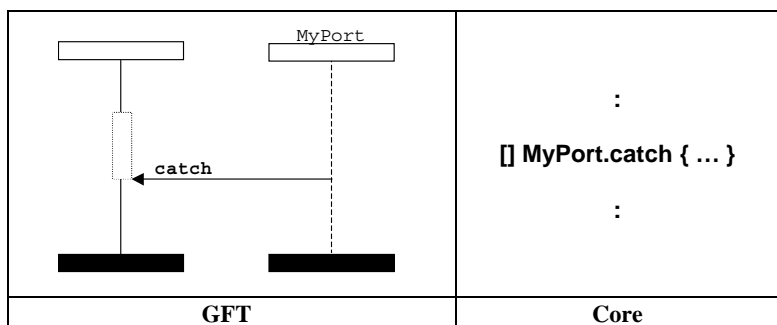
The timeout exception operation shall be represented by a timeout symbol with the arrow connected to the test component (see Figure 88). No further information shall be attached to the timeout symbol. It shall be used within a call symbol only. The message arrow head shall be attached to a preceding suspension region on the test component.



**Figure 88 – Timeout exception (within a call symbol)**

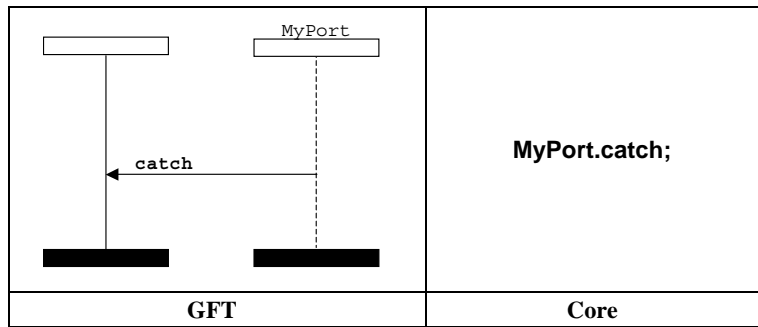
#### 11.8.4.6.2 Catch any exception

The catch any exception operation shall be represented by an incoming message arrow from the port instance to the test component and the keyword `catch` above the message arrow. Within a call symbol, the message arrow head shall be attached to a preceding suspension region on the test component (see Figure 89). Outside a call symbol, the message arrow head shall not be attached to a preceding suspension region on the test component (see Figure 90). The catch any exception shall have no template and no exception type.



**Figure 89 – Catch any exception (within a call symbol)**

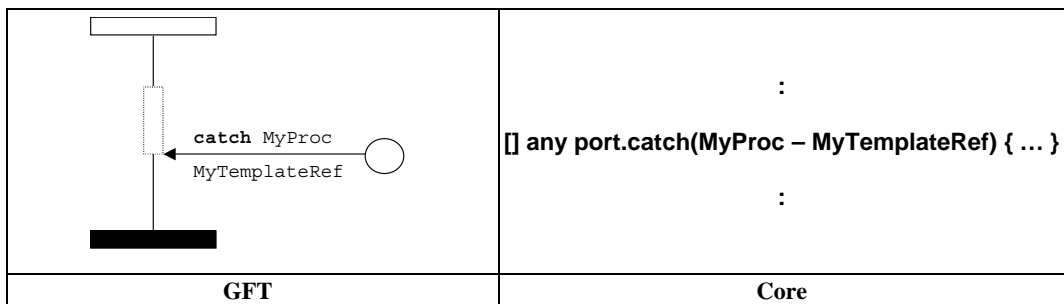




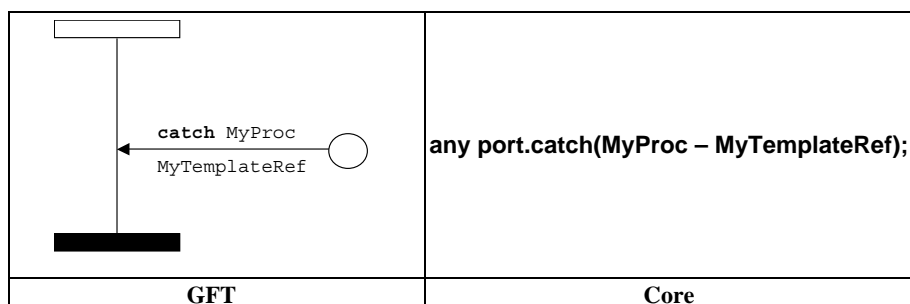
**Figure 90 – Catch any exception (outside a call symbol)**

### 11.8.4.6.3 Catch on any port

The catch on any port operation is represented by a found symbol representing any port to the test component and the keyword `catch` above the message arrow. Within a call symbol, the message arrow head shall be attached to a preceding suspension region on the test component (see Figure 91). Outside a call symbol, the message arrow head shall not be attached to a preceding suspension region on the test component (see Figure 92). The template if present is placed underneath the message arrow.



**Figure 91 – Catch on any port (within a call symbol)**



**Figure 92 – Catch on any port (outside a call symbol)**

### 11.8.5 The Check operation

The check operation shall be represented by an incoming message arrow from the port instance to the test component. The keyword `check` shall be placed above the message arrow. The attachment of the information related to the `receive` (see Figure 93), `getcall`, `getreply` (see Figures 94 and 95) and `catch` follows the check keyword and is according to the rules for representing those operations.

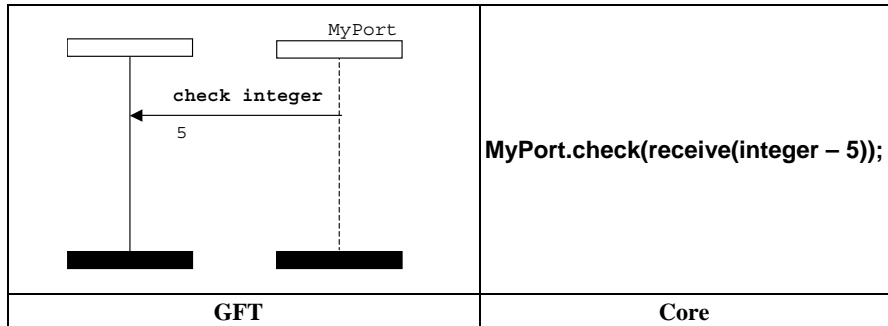


Figure 93 – Check a receive with inline template

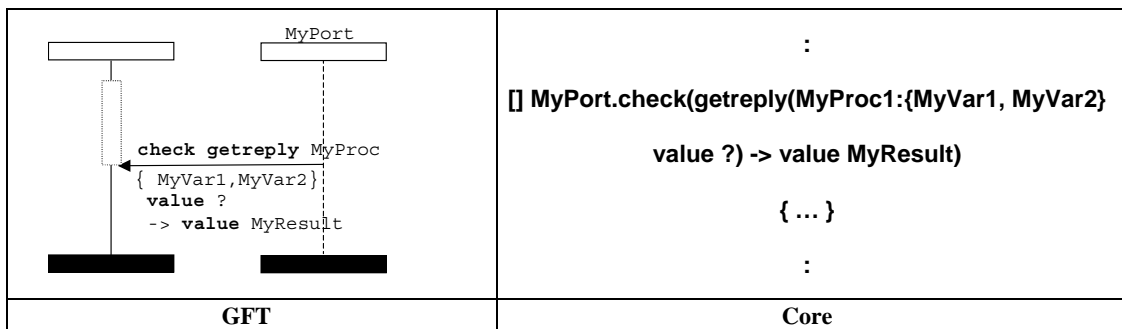


Figure 94 – Check a getreply (within a call symbol)

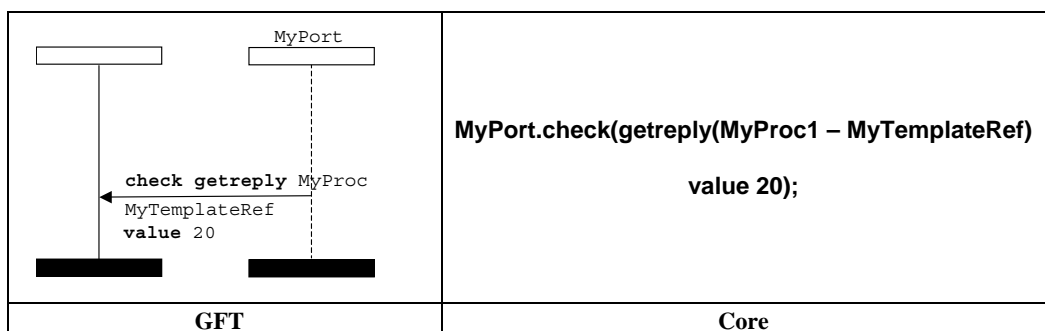
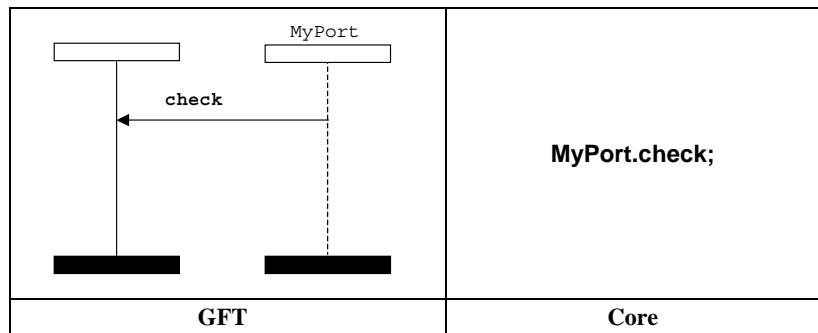


Figure 95 – Check a getreply (outside a call symbol)

#### 11.8.5.1 The Check any operation

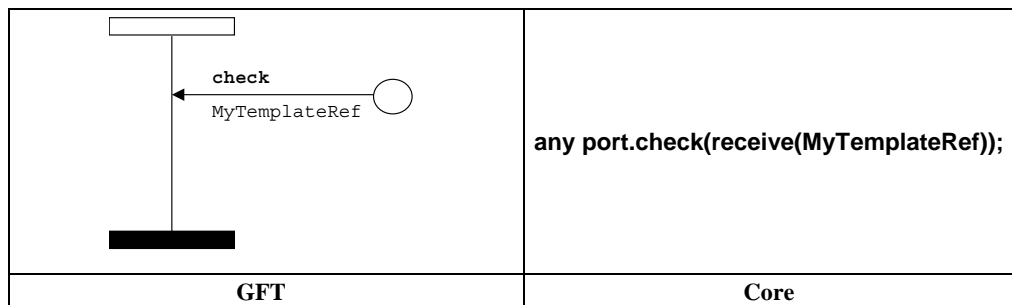
The check any operation shall be represented by an incoming message arrow from the port instance to the test component and the keyword `check` above the message arrow (see Figure 96). It shall have no receiving operation keyword, type and template attached to it. Optionally, an address information and storing the sender can be attached.



**Figure 96 – Check any operation**

### 11.8.5.2 Check on any port

The check on any port operation is represented by a found symbol representing any port to the test component and the keyword **check** above the message arrow (see Figure 97). The attachment of the information related to the **receive**, **getcall**, **getreply** and **catch** follows the check keyword and is according to the rules for representing those operations.

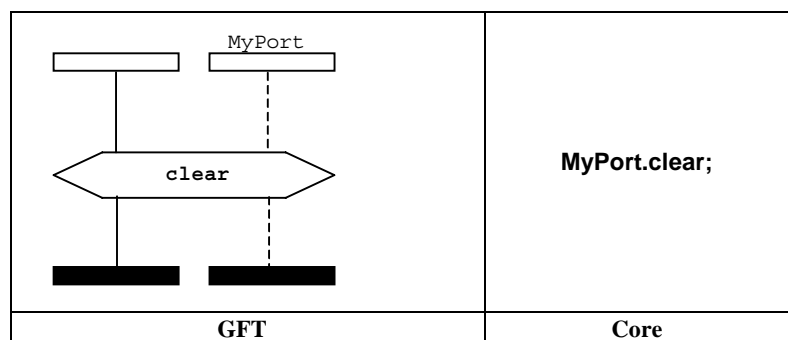


**Figure 97 – Check a receive on any port**

## 11.8.6 Controlling communication ports

### 11.8.6.1 The Clear port operation

The clear port operation shall be represented by a condition symbol with the keyword **clear**. It is attached to the test component instance, which performs the clear port operation, and to the port that is cleared (see Figure 98).



**Figure 98 – Clear port operation**

### 11.8.6.2 The Start port operation

The start port operation shall be represented by a condition symbol with the keyword `start`. It is attached to the test component instance, which performs the start port operation, and to the port that is started (see Figure 99).

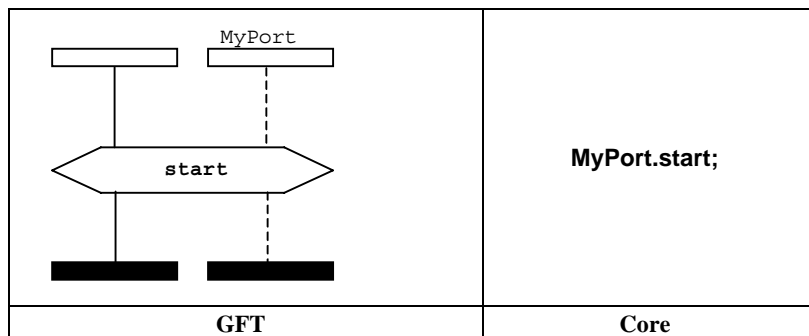


Figure 99 – Start port operation

### 11.8.6.3 The Stop port operation

The stop port operation shall be represented by a condition symbol with the keyword `stop`. It is attached to the test component instance, which performs the stop port operation, and to the port that is stopped (see Figure 100).

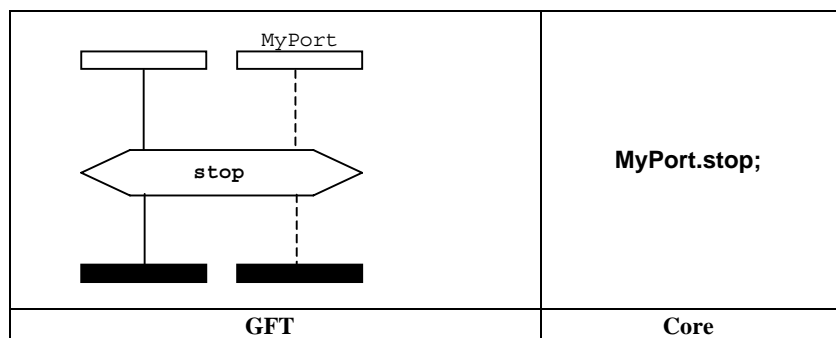


Figure 100 – Stop port operation

### 11.8.6.4 Use of any and all with ports

The GFT representation of the `any` keyword for ports together with the `receive`, `trigger`, `getcall`, `getreply`, `catch`, and `check` operations is explained in clauses 11.8.1 to 11.8.6.3.

The `all` keyword for ports together with the `clear`, `start` and `stop` operation is represented by attaching the condition symbol containing the `clear`, `start` or `stop` operation to all port instances represented in the GFT diagram for a testcase, function or altstep.

## 11.9 Timer operations

In GFT, there are two different timer symbols: one for identified timers and one for call timers (see Figure 101). They differ in appearance as solid line timer symbols are used for identified timers and dashed timer symbols for call timers. An identified timer shall have its name attached to its symbol, whereas a call timer does not have a name. Identified timers are described in this clause. The call timer is dealt with in clause 11.8.4.

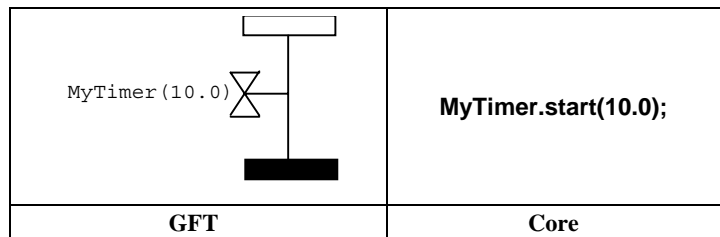


**Figure 101 – Identified timer and call timers**

GFT does not provide any graphical representation for the `running` timer operation (being a Boolean expression). It is textually denoted at the places of its use.

### 11.9.1 The Start timer operation

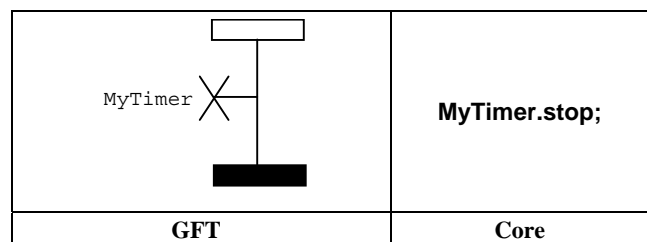
For the start timer operation, the start timer symbol shall be attached to the component instance. A timer name and an optional duration value (within parentheses) may be associated (see Figure 102).



**Figure 102 – The start timer operation**

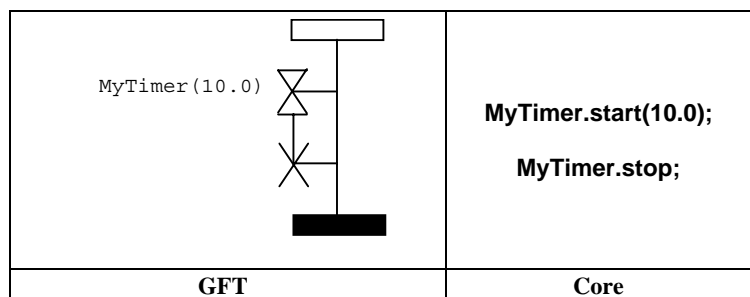
### 11.9.2 The Stop timer operation

For the stop timer operation, the stop timer symbol shall be attached to the component instance. An optional timer name may be associated (see Figure 103).



**Figure 103 – The stop timer operation**

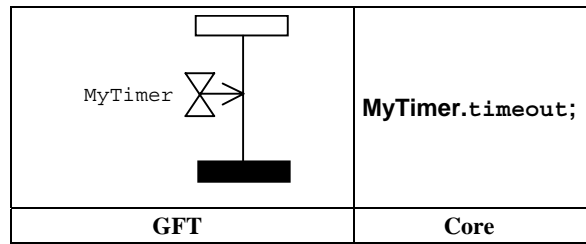
The symbols for a start timer and a stop timer operation may be connected with a vertical line. In this case, the timer identifier needs only be specified next to the start timer symbol (see Figure 104).



**Figure 104 – Connected start and stop timer symbols**

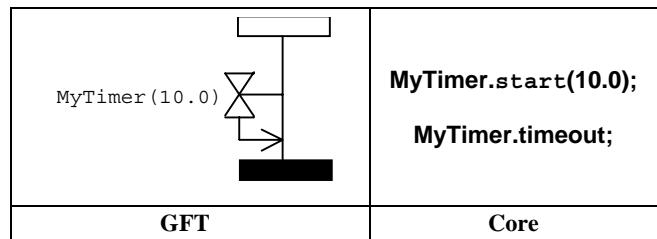
### 11.9.3 The Timeout operation

For the timeout operation, the timeout symbol shall be attached to the component instance. An optional timer name may be associated (see Figure 105).



**Figure 105 – The timeout operation**

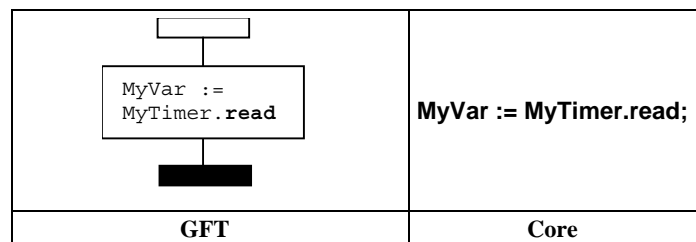
The symbols for a start timer and a timeout operation may be connected with a vertical line. In this case, the timer identifier needs only be specified next to the start timer symbol (see Figure 106).



**Figure 106 – Connected start and timeout timer symbols**

### 11.9.4 The Read timer operation

The read timer operation shall be put into an action box (see Figure 107).



**Figure 107 – The read timer operation**

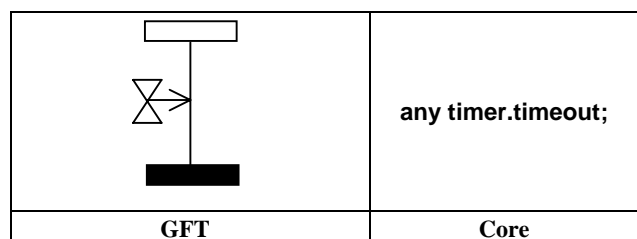
### 11.9.5 Use of any and all with timers

The stop timer operation can be applied to **all** timers (see Figure 108).



**Figure 108 – Stopping all timers**

The timeout operation can be applied to **any** timer (see Figure 109).

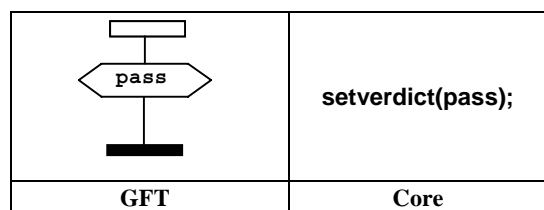


**Figure 109 – Timeout from any timer**

### 11.10 Test verdict operations

The verdict set operation **setverdict** is represented in GFT with a condition symbol within which the values **pass**, **fail**, **inconc** or **none** are denoted (see Figure 110).

NOTE – The rules for setting a new verdict follow the normal TTCN-3 overwriting rules for test verdicts.

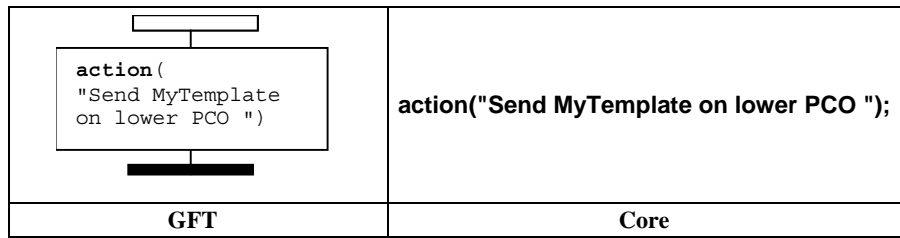


**Figure 110 – Set local verdict**

GFT does not provide any graphical representation for the **getverdict** operation (being an expression). It is textually denoted at the places of its use.

### 11.11 External actions

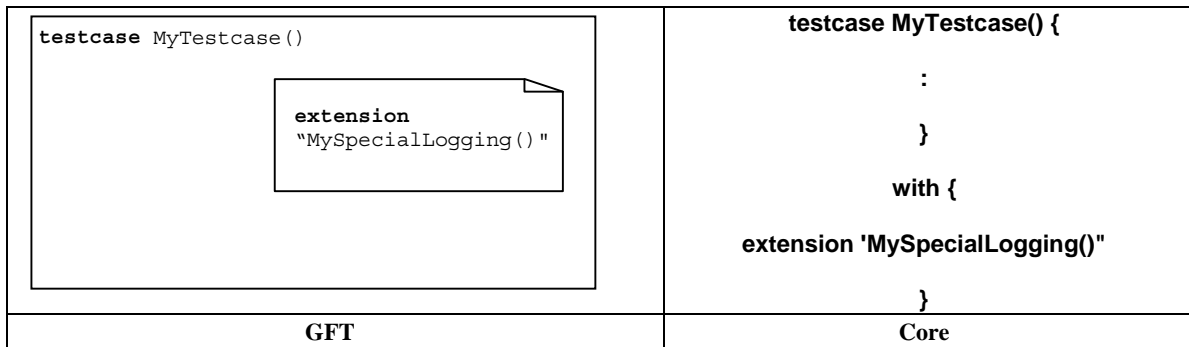
External actions are represented within action box symbols (see Figure 111). The syntax of the external action is placed within that symbol.



**Figure 111 – External actions**

### 11.12 Specifying attributes

The attributes defined for the module control part, testcases, functions and altsteps are represented within the text symbol. The syntax of the `with` statement is placed within that symbol. An example is given in Figure 112.



**Figure 112 – Specifying attributes**



## Annex A

### GFT BNF

(This annex forms an integral part of this Recommendation)

#### A.1 Meta-language for GFT

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

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

See [ITU-T Z.120] 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 – The syntactic meta-notation

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

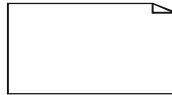
### A.3.1.5 Comments

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

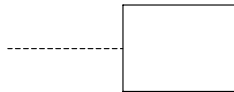
Note that there is no explicit rule for TTCN3 comments, they are explained in clause A.1.4 of [ITU-T Z.161]

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

```
TextSymbol ::=
```



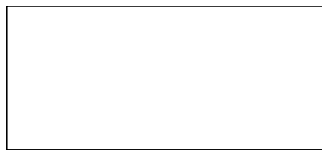
```
CommentArea ::=
    EventCommentSymbol contains TTCN3Comments
EventCommentSymbol ::=
```



/\* 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
```

```

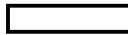
ComponentHeadArea ::=
  ( MTCOp | SelfOp )
  is followed by ( InstanceHeadSymbol [ contains ComponentType ] )

```

```

InstanceHeadSymbol ::=

```



```

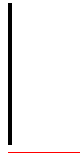
ComponentBodyArea ::=
  InstanceAxisSymbol
  is attached to { InstanceEventArea } set
  is followed by ComponentEndArea

```

```

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

```

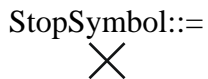
### A.3.2.4 Instance end

InstanceEndSymbol ::=

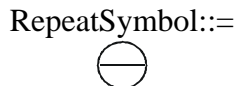


StopArea ::=  
 StopSymbol  
*is associated with* ( Expression )

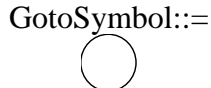
/\* STATIC SEMANTICS – the expression shall refer to either the mtc or to self \*/



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



GotoArea ::=  
 GotoSymbol  
*contains* LabelIdentifier



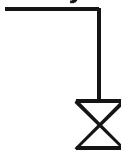
### A.3.3 Timer

InstanceTimerEventArea ::=  
 InstanceTimerStartArea  
 | InstanceTimerStopArea  
 | InstanceTimeoutArea

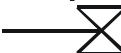
InstanceTimerStartArea ::=  
 TimerStartSymbol  
*is associated with* ( TimerRef [ "(" TimerValue " ] )  
*is attached to* InstanceAxisSymbol  
 [ *is attached to* { TimerStopSymbol2 | TimeoutSymbol3 } ]

TimerStartSymbol ::=  
 TimerStartSymbol1 | TimerStartSymbol2

TimerStartSymbol1 ::=



TimerStartSymbol2 ::=



InstanceTimerStopArea ::=  
 TimerStopArea1 | TimerStopArea2

```

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

```

```

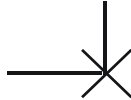
TimerStopArea2 ::=
  TimerStopSymbol2
  is attached to InstanceAxisSymbol
  is attached to TimerStartSymbol

```

**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

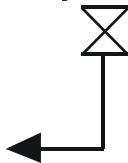
```

```

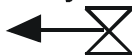
TimeoutSymbol ::=
  TimeoutSymbol1 | TimeoutSymbol2

```

**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

```

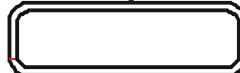
```

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

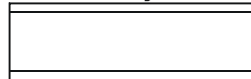
### A.3.7.1 Creation of test components

```
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 \*/

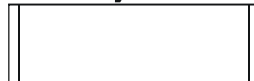
#### CreateSymbol ::=



### 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 \*/

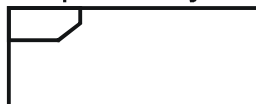
```

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 an 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 an 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 \*/

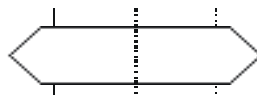
/\* STATIC SEMANTICS - the SetLocalVerdict shall not be used to assign the value error \*/

/\* 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, the port axis symbol is drawn through:



```
PortOperationText ::=
    ClearOpKeyword
    | StartKeyword
    | StopKeyword
```

```
ElseConditionArea ::=
    ConditionSymbol
    contains ElseKeyword
    is attached to InstanceAxisSymbol
```

**ConditionSymbol ::=**



### A.3.9.1 Condition on component instances

```
InstanceConditionArea ::=
  InstanceDoneArea
  | InstanceSetVerdictArea
  | InstancePortOperationArea

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

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

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

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

### A.3.9.2 Condition on ports

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

PortConditionBeginSymbol ::=
  VoidSymbol

PortConditionEndSymbol ::=
  VoidSymbol
```

### A.3.10 Message-based communication

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

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

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

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

/\* STATIC SEMANTICS – a from clause, if existent, shall be put underneath the message symbol \*/  
/\* STATIC SEMANTICS – a port redirect, if existent, shall be put underneath the message symbol \*/

**MessageSymbol ::=**



### A.3.10.1 Message-based communication on component instances

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

```
MessageOutSymbol ::=
    VoidSymbol
```

The VoidSymbol is a geometric point without spatial extension.

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

```
MessageInSymbol ::=
    VoidSymbol
```

### A.3.10.2 Message-based communication on port instances

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

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

### A.3.11 Signature-based communication

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

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

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

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

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

```

is attached to InstanceReplyEventArea
is attached to PortReplyInEventArea

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

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

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

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

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

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

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

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

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

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

```

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

### A.3.11.1 Signature-based communication on component instances

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

```
InstanceCallTimerStartArea ::=
  CallTimerStartSymbol
  is associated with TimerValue
  is attached to InstanceAxisSymbol
  is attached to SuspensionRegionSymbol
  [is attached to CallTimeoutSymbol3 ]
```

#### CallTimerStartSymbol ::=



```
InstanceNonBlockingCallEventArea ::=
  InstanceSendEventArea
```

```
InstanceGetcallEventArea ::=
  InstanceReceiveEventArea
```

```
InstanceReplyEventArea ::=
  InstanceSendEventArea
```

```
InstanceGetreplyWithinCallEventArea ::=
  InstanceReceiveEventArea
  is attached to SuspensionRegionSymbol
```

```
InstanceGetreplyOutsideCallEventArea ::=
  InstanceReceiveEventArea
```

```
InstanceRaiseEventArea ::=
  InstanceSendEventArea
```

```
InstanceCatchWithinCallEventArea ::=
  InstanceReceiveEventArea
  is attached to SuspensionRegionSymbol
```

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

#### CallTimeoutSymbol ::=



```
InstanceCatchOutsideCallEventArea ::=
  InstanceReceiveEventArea
```

### A.3.11.2 Signature-based communication on ports

```
PortGetcallOutEventArea ::=
  PortOutMsgEventArea
```

```
PortGetreplyOutEventArea ::=
  PortOutMsgEventArea
```

```
PortCatchOutEventArea ::=
  PortOutMsgEventArea
```

```
PortCallInEventArea ::=
  PortInMsgEventArea
```

```

PortReplyInEventArea ::=
    PortInMsgEventArea

PortRaiseInEventArea ::=
    PortInMsgEventArea

```

## A.3.12 Trigger and check

### A.3.12.1 Trigger and check on component instances

```

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

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

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

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

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

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

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

InstanceTriggerEventArea ::=
    InstanceReceiveEventArea

InstanceCheckEventArea ::=
    InstanceReceiveEventArea

```

### A.3.12.2 Trigger and check on port instances

```

PortTriggerOutEventArea ::=
    PortOutMsgEventArea

PortCheckOutEventArea ::=
    PortOutMsgEventArea

```

## A.3.13 Handling of communication from any port

```

InstanceFoundEventArea ::=
    FoundSymbol
    contains FoundEvent
    is attached to InstanceAxisSymbol

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

```

```

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

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

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

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

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

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

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

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

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

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

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

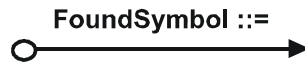
```

```

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

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

```



### 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 */

```





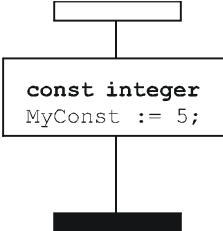
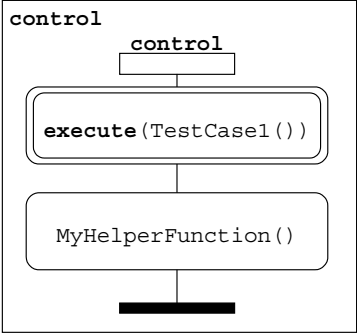
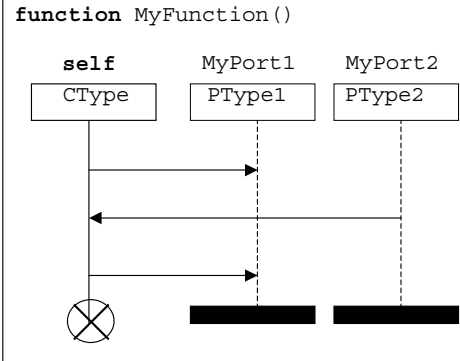
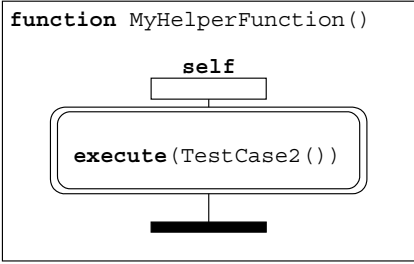
## Annex B

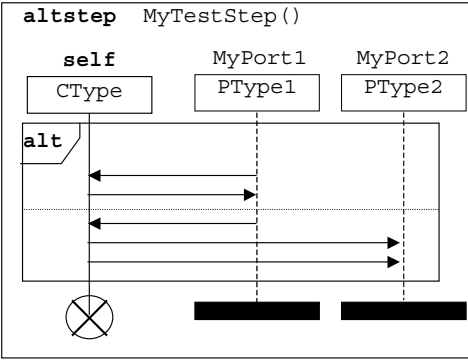
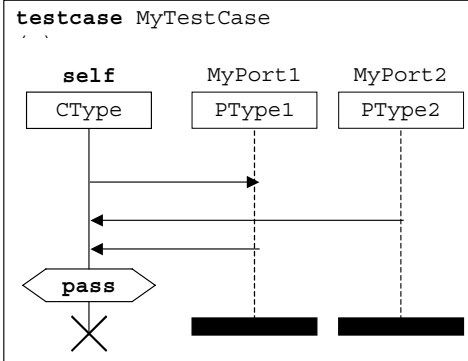
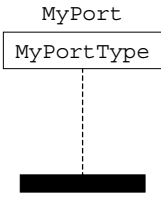
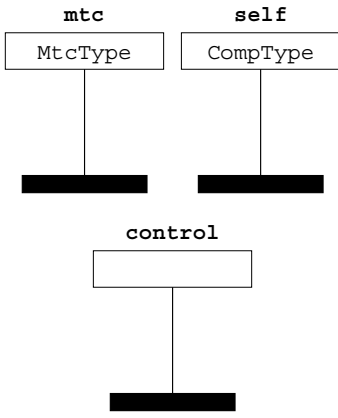
### Reference guide for GFT

(This annex forms an integral part of this Recommendation)

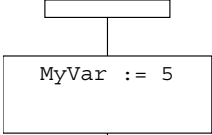
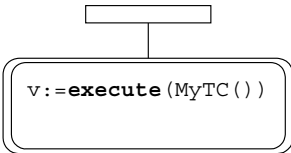
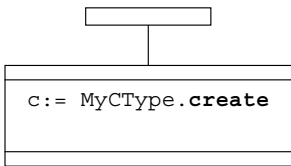
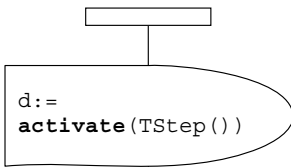
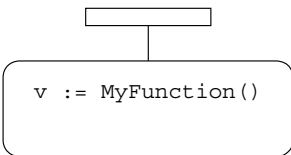
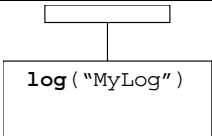
This annex lists the main TTCN-3 language elements and their representation in GFT. For a complete description of the GFT symbols and their use please refer to the main text.

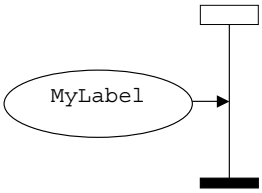
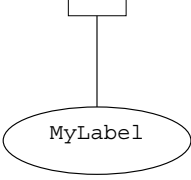
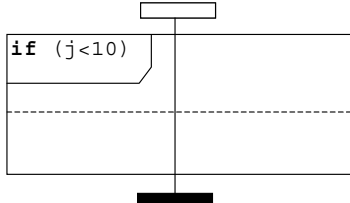
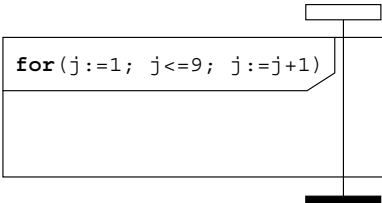
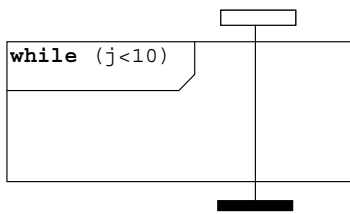
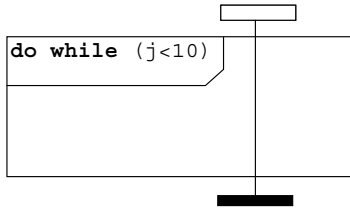
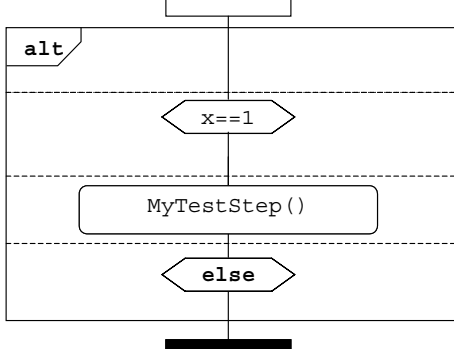
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
<b>Module definitions</b>			
TTCN-3 module definition	<b>module</b>		No special GFT symbol, i.e., the core language or another presentation format may be used.
Import of definitions from other module	<b>import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


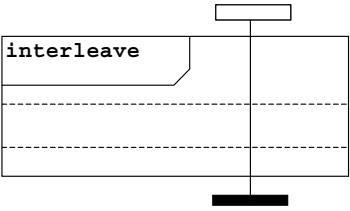
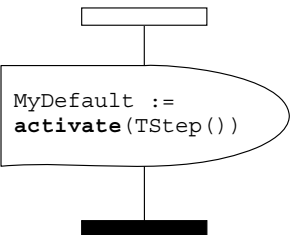
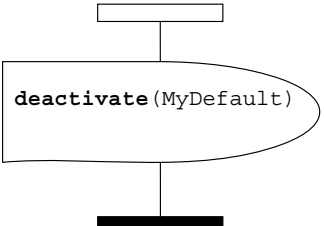
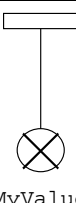
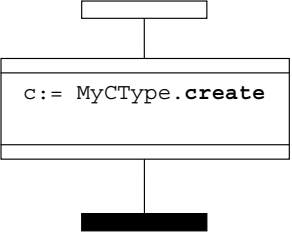
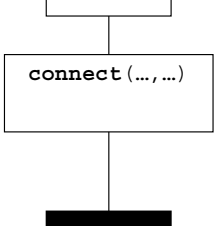
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
			Local constant declaration in an action box.
Data/signature template definitions	<b>template</b>		No special GFT symbol, i.e., the core language or another presentation format may be used.
Control definitions	<b>control</b>		GFT control diagram represents the control part of a TTCN-3 module.
Function definitions	<b>function</b>		GFT function diagrams are used to represent functions.
			GFT function diagrams may be defined to structure the behaviour of the control part of a TTCN-3 module.

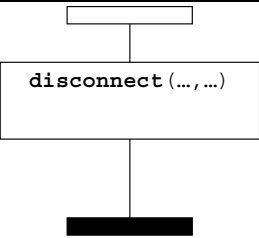
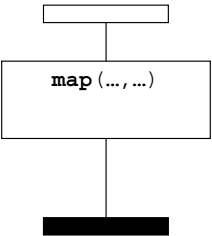
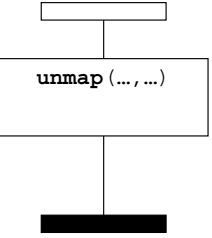
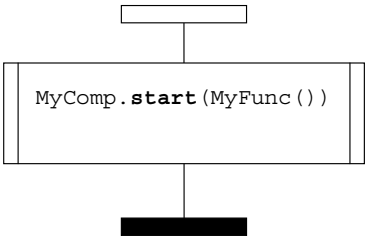
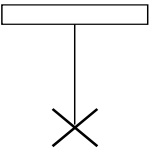
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
Altstep definitions	<b>altstep</b>	<pre>altstep MyTestStep ()</pre> 	GFT altstep diagrams are used to represent altsteps.
Test case definitions	<b>testcase</b>	<pre>testcase MyTestCase</pre> 	GFT test case diagrams are used to represent test cases.
<b>Usage of component instances and ports</b>			
Port instance			A Port in a test case, test step and function diagram is represented by an instance with a dashed instance line. The port name is specified above and the (optional) port type is described within the instance header.
Test component instance			<p>An <b>mtc</b> instance represents the main test component in a test case diagram.</p> <p>A <b>self</b> instance represents a test component in a test step or function diagram.</p> <p>A <b>control</b> instance represents the instance that executes the module control part in a control diagram.</p>

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
<b>Declarations</b>			
Variable declarations	<b>var</b>	<b>var integer</b> 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.
			Variable declaration within a test component creation symbol.
			Variable declaration within a default activation symbol.
			Variable declaration within a reference symbol.
Timer declarations	<b>timer</b>	<b>timer</b> MyTimer	Textual timer declaration in the header of a control, test case, test step or function diagram.
			Timer declaration in an action box.

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
<b>Basic program statements</b>			
Expressions	(…)		No special GFT symbol, i.e., the core language or another presentation format may be used.
Assignments	:=		Assignment in an action box.
			Assignment within a test case execution symbol.
			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.

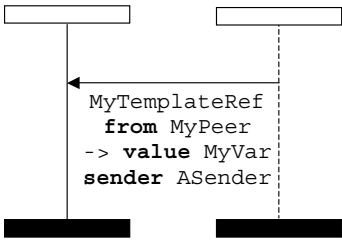
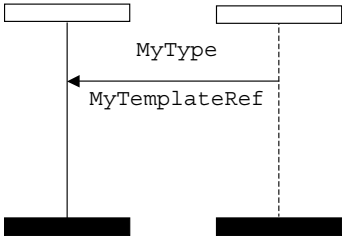
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
Label and Goto	<b>label</b>		Definition of a label.
	<b>goto</b>		Go to label.
If-else	<b>if (...)</b> {...} <b>else</b> {...}		
For loop	<b>for (...)</b> {...}		
While loop	<b>while (...)</b> {...}		
Do while loop	<b>do</b> {...} <b>while (...)</b>		
<b>Behavioural program statements</b>			
Alternative behaviour	<b>alt</b> {...}		

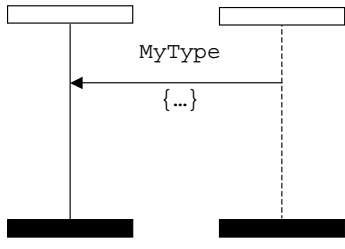
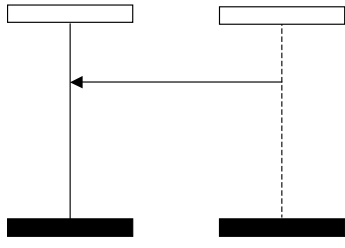
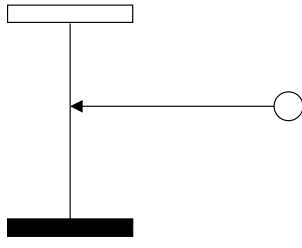
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
Repeat	<b>repeat</b>		To be used within alternative behaviour and test steps.
Interleaved behaviour	<b>interleave</b> {...}		
Activate a default	<b>activate</b>		The activate statement is put into a default symbol.
Deactivate a default	<b>deactivate</b>		The deactivate statement is put into a default symbol.
Returning control	<b>return</b>	 MyValue	The optional return value is attached to the return symbol.
<b>Configuration operations</b>			
Create parallel test component	<b>create</b>		The create statement is put into a test component creation symbol.
Connect component to component	<b>connect</b>		The connect statement is put into an action box.

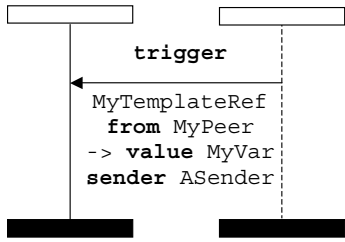
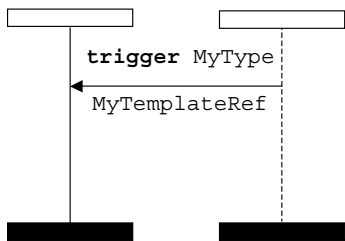
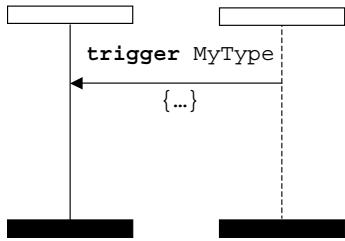
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
Disconnect two components	<b>disconnect</b>		The disconnect statement is put into an action box.
Map port to test system interface	<b>map</b>		The map statement is put into an action box.
Unmap port from test system interface	<b>unmap</b>		The unmap statement is put into an action box.
Get MTC address	<b>mtc</b>		No special GFT symbol, used within statements, expressions or as test component identifier.
Get test system interface address	<b>system</b>		No special GFT symbol, used within statements or expressions.
Get own address	<b>self</b>		No special GFT symbol, used within statements, expressions or as test component identifier.
Start execution of test component	<b>start</b>		The start statement is put into a start symbol.
Stop execution of a test component by itself	<b>stop</b>		The termination of mtc terminates also all the other test components. Port instances cannot be stopped.

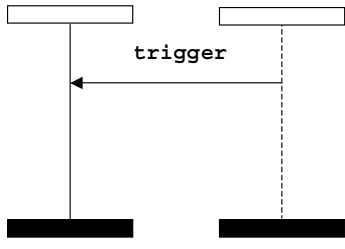
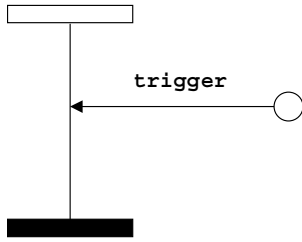
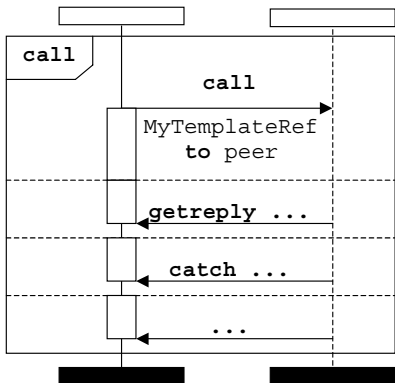


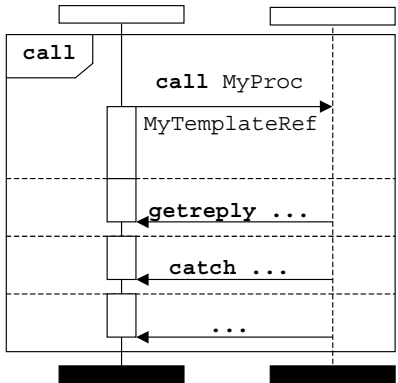
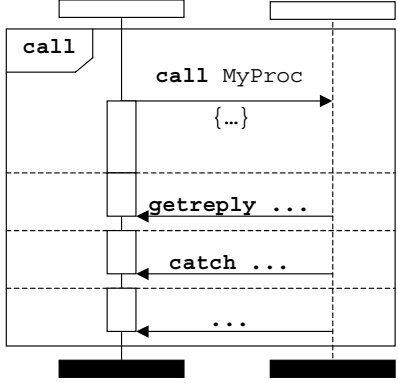
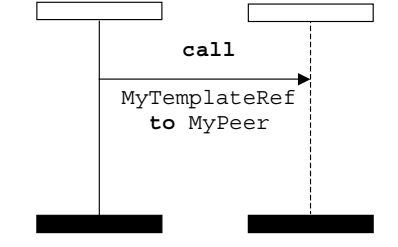
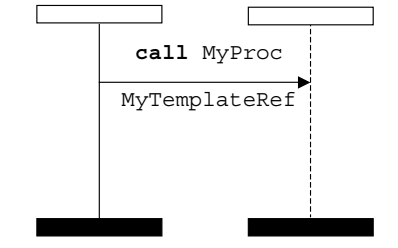
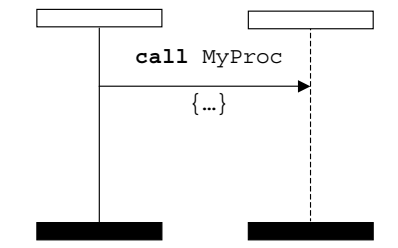
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
Of another test component			The component identifier is put near to the stop symbol.
Check termination of a PTC	<b>running</b>		No special GFT symbol, used within expressions.
Wait for termination of a PTC	<b>done</b>		The done statement is put into a condition symbol.
<b>Communication operations</b>			
Send message	<b>send</b>		Send a message defined by a template reference but without type information. The receiver is identified uniquely by the (optional) <b>to</b> -directive.
			Send a message defined by a template reference and with type information. An (optional) <b>to</b> -directive may be present to identify the peer entity uniquely.
			Send a message defined by an inline template definition. An (optional) <b>to</b> -directive may be present to identify the peer entity uniquely.

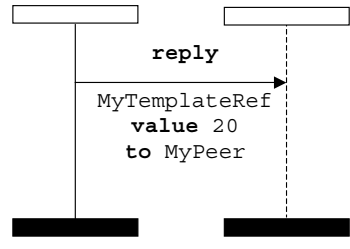
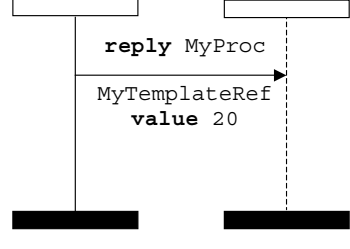
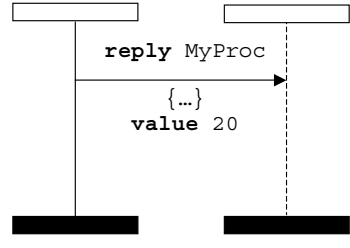
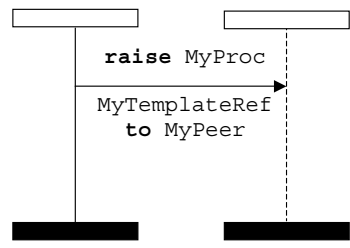
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
Receive message	<b>receive</b>	 <pre> sequenceDiagram     participant A     participant B     B--&gt;&gt;A: MyTemplateRef from MyPeer -&gt; value MyVar sender ASender   </pre>	<p>Receive a message with a value defined by a template reference but without type information.</p> <p>The (optional) <b>from</b>-directive denotes that the sender of the message shall be identified by variable <code>MyPeer</code>.</p> <p>The (optional) <b>value</b>-directive assigns received message to variable <code>MyVar</code>.</p> <p>The (optional) <b>sender</b>-directive retrieves the identifier of the sender and stores it in variable <code>ASender</code>.</p>
		 <pre> sequenceDiagram     participant A     participant B     B--&gt;&gt;A: MyType MyTemplateRef   </pre>	<p>Receive a message with a value defined by a template reference and with type information.</p> <p>Optional <b>from</b>-, <b>value</b>- and <b>sender</b>-directives may be present to identify the sender of the message, to assign the message to a variable or to retrieve the identifier of the peer entity.</p>

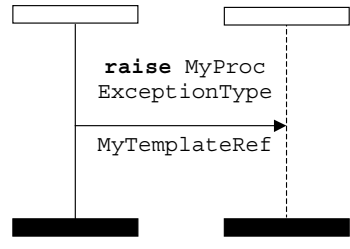
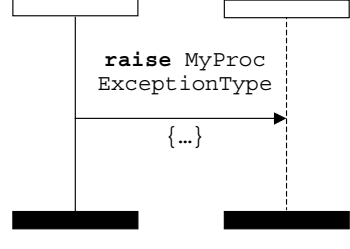
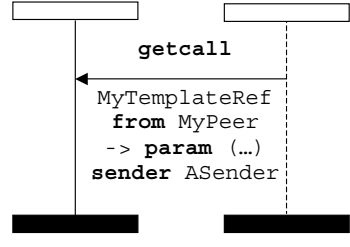
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
			<p>Receive a message with a value defined by an inline template definition.</p> <p>Optional <b>from-</b>, <b>value-</b> and <b>sender-</b> directives may be present to identify the sender of the message, to assign the message to a variable or to retrieve the identifier of the peer entity.</p>
			<p>Receive any message (no value and no type is specified).</p> <p>Optional <b>from-</b>, <b>value-</b> and <b>sender-</b> directives may be present to identify the sender of the message, to assign the message to a variable or to retrieve the identifier of the peer entity.</p>
			<p>Receive any message (no value and no type is specified) from any port.</p> <p>The message value to be received from any port may be restricted by means referring to templates or by using inline templates.</p> <p>Optional <b>from-</b>, <b>value-</b> and <b>sender-</b> directives may be present to identify the sender of the message, to assign the message to a variable or to retrieve the identifier of the peer entity.</p>

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
Trigger message	<b>trigger</b>	 <pre> sequenceDiagram     participant A     participant B     B--&gt;&gt;A: trigger     Note over A: MyTemplateRef     Note over A: from MyPeer     Note over A: -&gt; value MyVar     Note over A: sender ASender </pre>	<p>Trigger on a message with a value defined by a template reference but without type information.</p> <p>The (optional) <b>from</b>-directive denotes that the sender of the message shall be identified by variable <i>MyPeer</i>.</p> <p>The (optional) <b>value</b>-directive assigns received message to variable <i>MyVar</i>.</p> <p>The (optional) <b>sender</b>-directive retrieves the identifier of the sender and stores it in variable <i>ASender</i>.</p>
		 <pre> sequenceDiagram     participant A     participant B     B--&gt;&gt;A: trigger MyType     Note over A: MyTemplateRef </pre>	<p>Trigger on a message with a value defined by a template reference and with type information.</p> <p>Optional <b>from</b>-, <b>value</b>- and <b>sender</b>-directives may be present to identify the sender of the message, to assign the message to a variable or to retrieve the identifier of the peer entity.</p>
		 <pre> sequenceDiagram     participant A     participant B     B--&gt;&gt;A: trigger MyType     Note over A: {...} </pre>	<p>Trigger on a message with a value defined by an inline template definition.</p> <p>Optional <b>from</b>-, <b>value</b>- and <b>sender</b>-directives may be present to identify the sender of the message, to assign the message to a variable or to retrieve the identifier of the peer entity.</p>

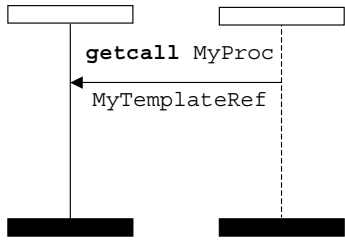
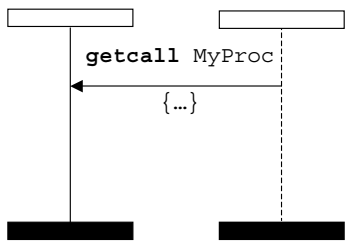
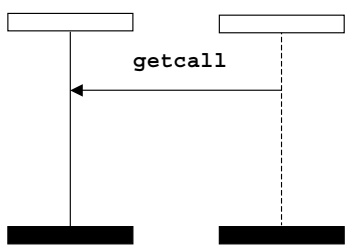
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
		 <p>A sequence diagram with two lifelines. The left lifeline is solid, and the right is dashed. A message arrow labeled 'trigger' points from the right lifeline to the left. Both lifelines have thick black bars at their base.</p>	<p>Trigger on any message (no value and no type is specified).</p> <p>Optional <b>from-</b>, <b>value-</b> and <b>sender-</b> directives may be present to identify the sender of the message, to assign the message to a variable (of type <b>anytype</b>) and to retrieve the identifier of the peer entity.</p>
		 <p>A sequence diagram with one solid lifeline. A message arrow labeled 'trigger' points from a small circle (representing a port) to the lifeline. The lifeline has a thick black bar at its base.</p>	<p>Trigger on any message (no value and no type is specified) from any port.</p> <p>The value of the message that shall cause the trigger from any port may be restricted by means referring to templates or by using inline templates.</p> <p>Optional <b>from-</b>, <b>value-</b> and <b>sender-</b> directives may be present to identify the sender of the message, to assign the message to a variable (of type <b>anytype</b>) and to retrieve the identifier of the peer entity.</p>
Invoke blocking procedure call	<b>call</b>	 <p>A sequence diagram with two lifelines. The left lifeline has a thick black bar at its base. A 'call' block is shown on the left lifeline, containing a nested 'call' block. The inner 'call' block sends a message 'MyTemplateRef to peer' to the right lifeline. The right lifeline returns 'getreply ...' to the inner 'call' block. The inner 'call' block returns 'catch ...' to the outer 'call' block. The outer 'call' block returns '...' to the left lifeline. Dashed lines separate the different levels of the call structure.</p>	<p>Invoking a blocking procedure by using a signature template.</p> <p>The receiver is identified uniquely by the (optional) <b>to-</b>directive.</p> <p>The call body, i.e., possible <b>getreply</b> and <b>catch</b> operations, is shown schematically only.</p>

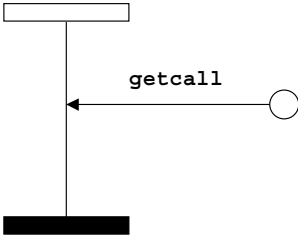
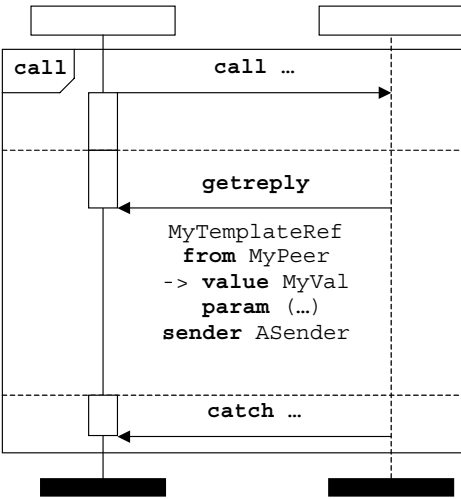
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
		 <p>A sequence diagram with two lifelines. A callout box labeled 'call' points to the start of the first lifeline. An arrow labeled 'call MyProc' points from the first lifeline to the second, with 'MyTemplateRef' written below it. A return arrow labeled 'getreply ...' points from the second lifeline back to the first. A return arrow labeled 'catch ...' points from the second lifeline back to the first. A return arrow labeled '...' points from the second lifeline back to the first. The lifelines end with thick black bars.</p>	<p>Invoking a blocking procedure by using a signature template and signature information.</p> <p>An (optional) <b>to</b>-directive may be present to identify the peer entity uniquely.</p> <p>The call body, i.e., possible <b>getreply</b> and <b>catch</b> operations, is shown schematically only.</p>
		 <p>A sequence diagram with two lifelines. A callout box labeled 'call' points to the start of the first lifeline. An arrow labeled 'call MyProc' points from the first lifeline to the second, with '{...}' written below it. A return arrow labeled 'getreply ...' points from the second lifeline back to the first. A return arrow labeled 'catch ...' points from the second lifeline back to the first. A return arrow labeled '...' points from the second lifeline back to the first. The lifelines end with thick black bars.</p>	<p>Invoking a blocking procedure by using an inline template.</p> <p>An (optional) <b>to</b>-directive may be present to identify the peer entity uniquely.</p> <p>The call body, i.e., possible <b>getreply</b> and <b>catch</b> operations, is shown schematically only.</p>
Invoke non-blocking procedure call	<b>call</b>	 <p>A sequence diagram with two lifelines. An arrow labeled 'call' points from the first lifeline to the second, with 'MyTemplateRef to MyPeer' written below it. The lifelines end with thick black bars.</p>	<p>Call a remote procedure, the call is defined by a template reference but without signature information.</p> <p>The receiver is identified uniquely by the (optional) <b>to</b>-directive.</p>
		 <p>A sequence diagram with two lifelines. An arrow labeled 'call MyProc' points from the first lifeline to the second, with 'MyTemplateRef' written below it. The lifelines end with thick black bars.</p>	<p>Call the remote procedure MyProc. The call is defined by a template reference.</p> <p>An (optional) <b>to</b>-directive may be present to identify the peer entity uniquely.</p>
		 <p>A sequence diagram with two lifelines. An arrow labeled 'call MyProc' points from the first lifeline to the second, with '{...}' written below it. The lifelines end with thick black bars.</p>	<p>Call the remote procedure MyProc. The call is defined by an inline template.</p> <p>An (optional) <b>to</b>-directive may be present to identify the peer entity uniquely.</p>

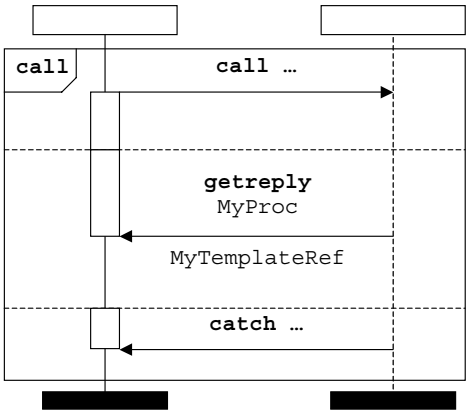
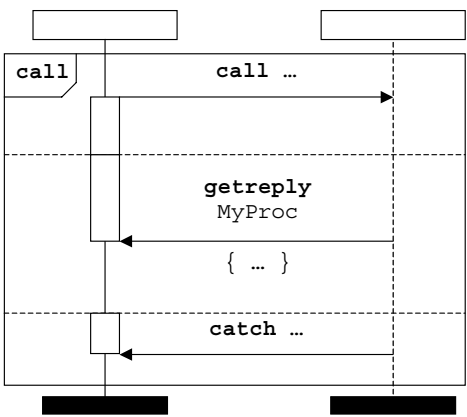
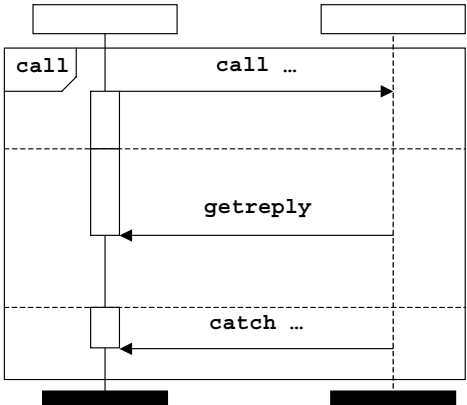
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
Reply to procedure call from remote entity	<b>reply</b>	 <pre> sequenceDiagram     participant Local as [ ]     participant Remote as [ ]     Local-&gt;&gt;Remote: reply     Note over Local,Remote: MyTemplateRef     Note over Local,Remote: value 20     Note over Local,Remote: to MyPeer   </pre>	<p>Reply to a remote procedure call. The reply is defined by a template reference and the possible return value (<b>value</b>-directive).</p> <p>NOTE 1 – The signature information is part of the template definition.</p> <p>The receiver is identified uniquely by the (optional) <b>to</b>-directive.</p>
		 <pre> sequenceDiagram     participant Local as [ ]     participant Remote as [ ]     Local-&gt;&gt;Remote: reply MyProc     Note over Local,Remote: MyTemplateRef     Note over Local,Remote: value 20   </pre>	<p>Reply to a remote procedure call of <b>MyProc</b>. The reply is defined by a template reference and the possible return value (<b>value</b>-directive).</p> <p>An (optional) <b>to</b>-directive may be present to identify the peer entity uniquely.</p>
		 <pre> sequenceDiagram     participant Local as [ ]     participant Remote as [ ]     Local-&gt;&gt;Remote: reply MyProc     Note over Local,Remote: {...}     Note over Local,Remote: value 20   </pre>	<p>Reply to a remote procedure call of <b>MyProc</b>. The reply is defined by an inline template and the possible return value (<b>value</b>-directive).</p> <p>An (optional) <b>to</b>-directive may be present to identify the peer entity uniquely.</p>
Raise exception (to an accepted call)	<b>raise</b>	 <pre> sequenceDiagram     participant Local as [ ]     participant Remote as [ ]     Local-&gt;&gt;Remote: raise MyProc     Note over Local,Remote: MyTemplateRef     Note over Local,Remote: to MyPeer   </pre>	<p>Raise an exception to an accepted call of <b>MyProc</b>. The exception is defined by a template reference.</p> <p>NOTE 2 – The type of the exception is defined within the template definition.</p> <p>The receiver is identified uniquely by the (optional) <b>to</b>-directive.</p>

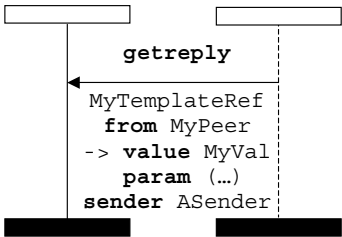
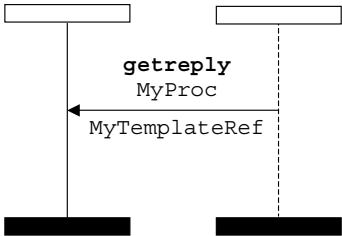
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
		 <pre> sequenceDiagram     participant Peer as Peer Entity     participant Actor as Actor     Peer-&gt;&gt;Actor: raise MyProc ExceptionType MyTemplateRef   </pre>	<p>Raise an exception to an accepted call of <b>MyProc</b>. The exception is defined by its (optional) type and a template reference.</p> <p>An (optional) <b>to</b>-directive may be present to identify the peer entity uniquely.</p>
		 <pre> sequenceDiagram     participant Peer as Peer Entity     participant Actor as Actor     Peer-&gt;&gt;Actor: raise MyProc ExceptionType {...}   </pre>	<p>Raise an exception to an accepted call of <b>MyProc</b>. The exception is defined by its type and an inline template.</p> <p>An (optional) <b>to</b>-directive may be present to identify the peer entity uniquely.</p>
Accept procedure call from remote entity	<b>getcall</b>	 <pre> sequenceDiagram     participant Peer as Peer Entity     participant Actor as Actor     Peer-&gt;&gt;Actor: getcall MyTemplateRef from MyPeer -&gt; param (...) sender ASender   </pre>	<p>Accept a procedure call from a remote entity. The call signature has to match the conditions defined by the template reference.</p> <p>NOTE 3 – The signature information is part of the template definition.</p> <p>The (optional) <b>from</b>-directive denotes that the sender of the call shall be identified by variable <b>MyPeer</b>.</p> <p>The (optional) <b>param</b>-directive assigns in-parameter values to Variables.</p> <p>The (optional) <b>sender</b>-directive retrieves the identifier of the sender and stores it in variable <b>ASender</b>.</p>

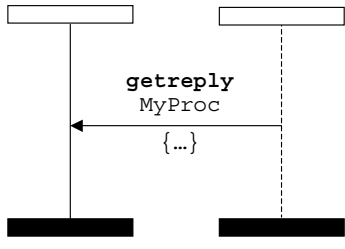
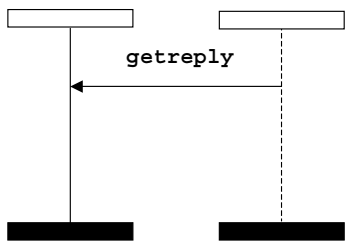
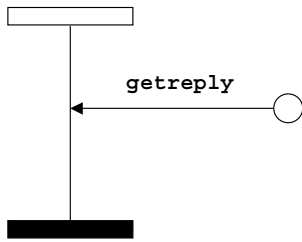


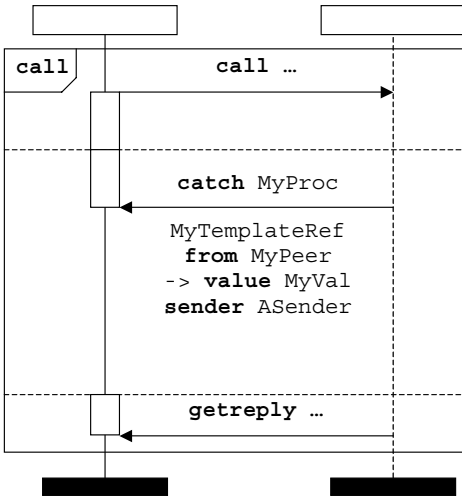
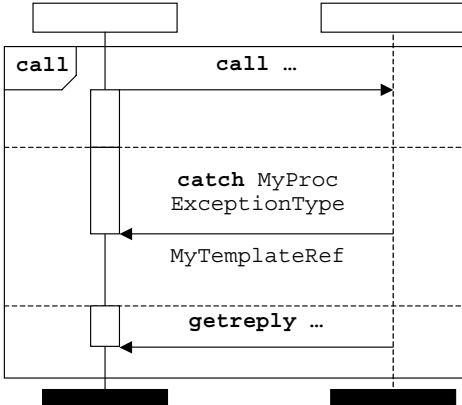
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
			<p>Accept a procedure call from a remote entity. The call signature has to match the conditions defined by signature reference and the template reference.</p> <p>Optional <b>from-</b>, <b>param-</b> and <b>sender-</b> directives may be present to identify the sender of the call, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</p>
			<p>Accept a procedure call from a remote entity. The call signature has to match the conditions defined by signature reference and the inline template definition.</p> <p>Optional <b>from-</b>, <b>param-</b> and <b>sender-</b> directives may be present to identify the sender of the call, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</p>
			<p>Accept any procedure call from any remote entity.</p> <p>Optional <b>from-</b> and <b>sender-</b> directives may be present to identify the sender of the call or to retrieve the identifier of the peer entity.</p>

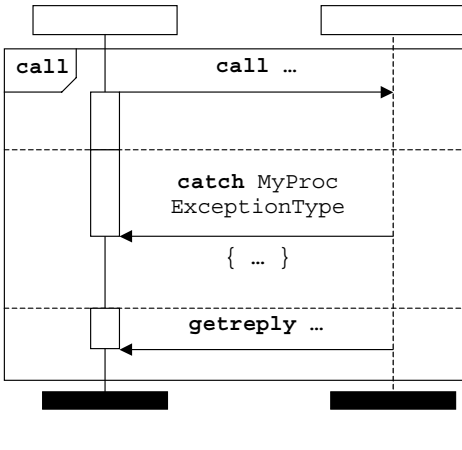
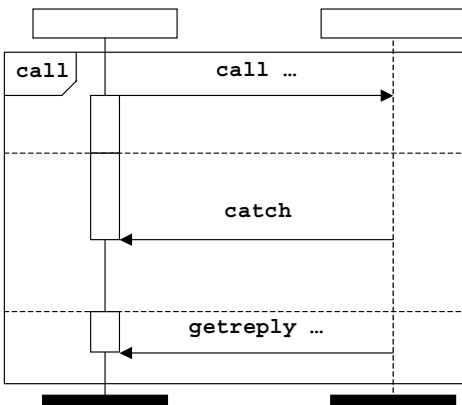
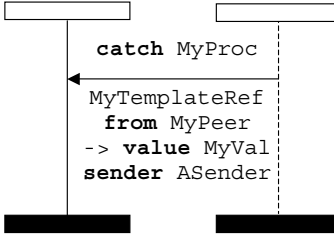
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
			<p>Accept any procedure call from any remote entity at any port.</p> <p>The call to be received from any port may be restricted by means referring to templates or by using inline templates.</p> <p>Optional <b>from-</b>, <b>param-</b> and <b>sender-</b> directives may be present to identify the sender of the call, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</p>
Handle response from a previous blocking call	<b>getreply</b>		<p>Receive a response from a blocking call. The reply has to match the conditions defined by the template reference.</p> <p>NOTE 4 – The signature information is part of the template definition.</p> <p>The (optional) <b>from-</b> directive denotes that the sender of the call shall be identified by variable MyPeer.</p> <p>The (optional) <b>value-</b> directive assigns the possible return value of the procedure to variable MyVal.</p> <p>The (optional) <b>param-</b> directive assigns out-parameter values to Variables.</p> <p>The (optional) <b>sender-</b> directive retrieves the identifier of the sender and stores it in variable ASender.</p>

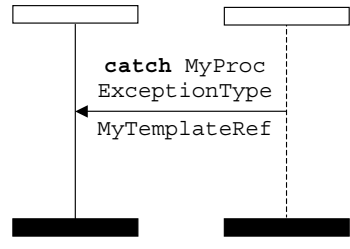
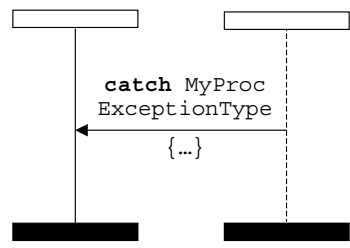
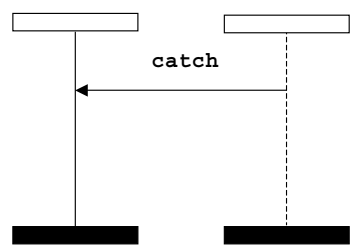
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
			<p>Receive a response from a blocking call. The reply has to match the conditions defined by signature reference and the template reference.</p> <p>Optional <b>from-</b>, <b>value-</b>, <b>param-</b> and <b>sender-</b> directives may be present to identify the sender of the reply, to retrieve the return value of the procedure, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</p>
			<p>Receive a response from a blocking call. The reply has to match the conditions defined by signature reference and the inline template definition.</p> <p>Optional <b>from-</b>, <b>value-</b>, <b>param-</b> and <b>sender-</b> directives may be present to identify the sender of the reply, to retrieve the return value of the procedure, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</p>
			<p>Accept any response from a blocking call.</p>

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
Handle response from a previous non-blocking call or independent from a call	<b>getreply</b>	 <pre> sequenceDiagram     participant A     participant B     A-&gt;&gt;B: getreply     B--&gt;A: MyTemplateRef     B--&gt;A: from MyPeer     B--&gt;A: -&gt; value MyVal     B--&gt;A: param (...)     B--&gt;A: sender ASender </pre>	<p>Receive a response from a previous call. The reply has to match the conditions defined by the template reference.</p> <p>NOTE 5 – The signature information is part of the template definition.</p> <p>The (optional) <b>from</b>-directive denotes that the sender of the call shall be identified by variable <i>MyPeer</i>.</p> <p>The (optional) <b>value</b>-directive assigns the possible return value of the procedure to variable <i>MyVal</i>.</p> <p>The (optional) <b>param</b>-directive assigns out-parameter values to Variables.</p> <p>The (optional) <b>sender</b>-directive retrieves the identifier of the sender and stores it in variable <i>ASender</i>.</p>
		 <pre> sequenceDiagram     participant A     participant B     A-&gt;&gt;B: getreply     B--&gt;A: MyProc     B--&gt;A: MyTemplateRef </pre>	<p>Receive a response from a previous call. The reply has to match the conditions defined by signature reference and the template reference.</p> <p>Optional <b>from</b>-, <b>value</b>-, <b>param</b>- and <b>sender</b>-directives may be present to identify the sender of the reply, to retrieve the return value of the procedure, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</p>

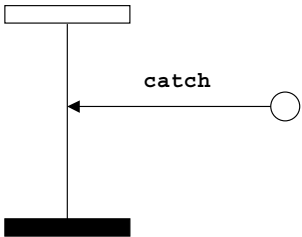
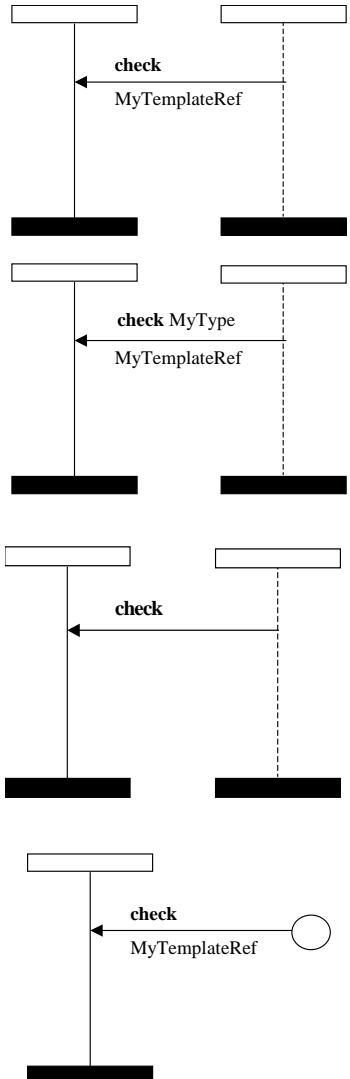
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
			<p>Receive a response from a previous call. The reply has to match the conditions defined by signature reference and the inline template definition.</p> <p>Optional <b>from-</b>, <b>value-</b>, <b>param-</b> and <b>sender-</b> directives may be present to identify the sender of the reply, to retrieve the return value of the procedure, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</p>
			<p>Accept any response from any previous call.</p> <p>Optional <b>from-</b> and <b>sender-</b> directives may be present to identify the sender of the reply or to retrieve the identifier of the peer entity.</p>
			<p>Accept any response from any previous call at any port.</p> <p>The reply to be received from any port may be restricted by means referring to templates or by using inline templates.</p> <p>Optional <b>from-</b>, <b>value-</b>, <b>param-</b> and <b>sender-</b> directives may be present to identify the sender of the reply, to retrieve the return value of the procedure, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</p>

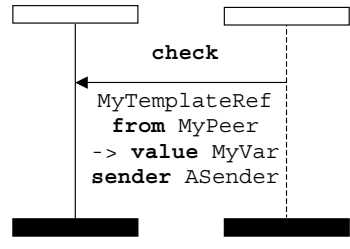
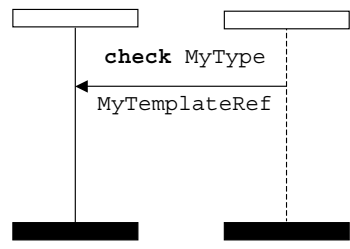
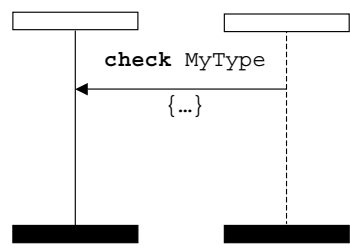
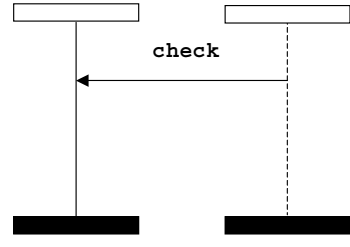
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
Catch exception from a previous blocking call	<b>catch</b>	 <pre> sequenceDiagram     participant A     participant B     A-&gt;&gt;B: call ...     activate B     B--&gt;&gt;A: catch MyProc     B--&gt;&gt;A: MyTemplateRef from MyPeer -&gt; value MyVal sender ASender     deactivate B     A--&gt;&gt;A: getreply ...   </pre>	<p>Catch an exception from a previous call. The exception has to match the conditions defined by the template reference.</p> <p>NOTE 6 – The type information is part of the template definition.</p> <p>The (optional) <b>from</b>-directive denotes that the sender of the exception shall be identified by variable MyPeer.</p> <p>The (optional) <b>value</b>-directive assigns the value of the exception to variable MyVal.</p> <p>The (optional) <b>sender</b>-directive retrieves the identifier of the sender and stores it in variable ASender.</p>
		 <pre> sequenceDiagram     participant A     participant B     A-&gt;&gt;B: call ...     activate B     B--&gt;&gt;A: catch MyProc ExceptionType     B--&gt;&gt;A: MyTemplateRef     deactivate B     A--&gt;&gt;A: getreply ...   </pre>	<p>Catch an exception from a previous call. The exception has to match the conditions defined by the exception type and the template reference.</p> <p>Optional <b>from</b>-, <b>value</b>-, and <b>sender</b>-directives may be present to identify the sender of the exception, to retrieve the exception value or to retrieve the identifier of the peer entity.</p>

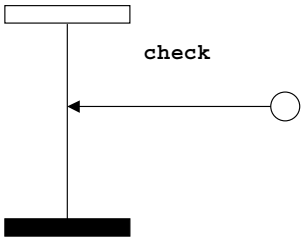
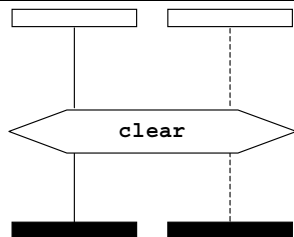
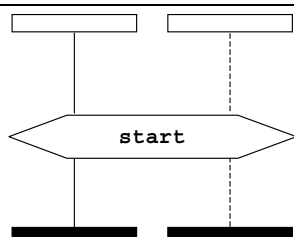
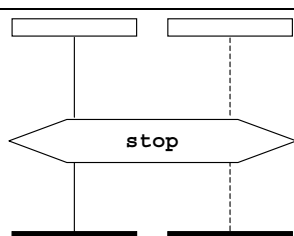
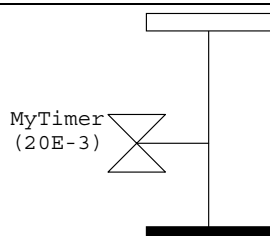
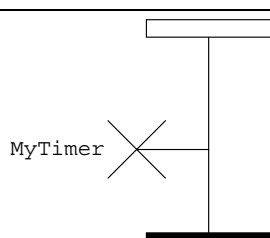
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
			<p>Catch an exception from a previous call. The exception has to match the conditions defined by the exception type and the inline template definition.</p> <p>Optional <b>from-</b>, <b>value-</b>, and <b>sender-</b>directives may be present to identify the sender of the exception, to retrieve the exception value or to retrieve the identifier of the peer entity.</p>
			<p>Accept any exception from a blocking call.</p> <p>Optional <b>from-</b>, <b>value-</b> and <b>sender-</b>directives may be present to identify the sender of the exception, to retrieve the exception value (and assign it to a variable of type <b>anytype</b>) or to retrieve the identifier of the peer entity.</p>
<p>Catch exception from a previous non-blocking call or independent from a call</p>	<p><b>catch</b></p>		<p>Catch an exception from a previous call. The exception has to match the conditions defined by the template reference.</p> <p>NOTE 7 – The type information is part of the template definition.</p> <p>The (optional) <b>from-</b>directive denotes that the sender of the exception shall be identified by variable <i>MyPeer</i>.</p> <p>The (optional) <b>value-</b>directive assigns the value of the exception to variable <i>MyVal</i>.</p> <p>The (optional) <b>sender-</b>directive retrieves the identifier of the sender and stores it in variable <i>ASender</i>.</p>

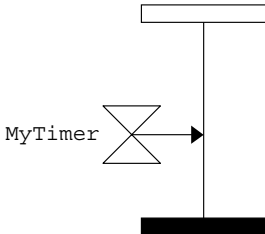
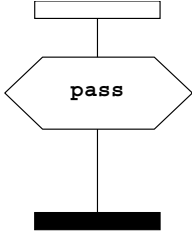
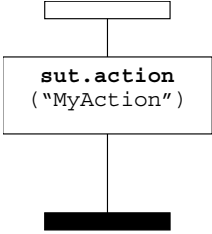
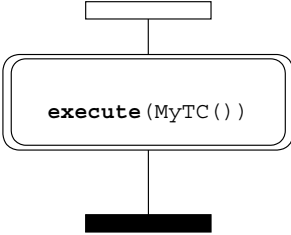
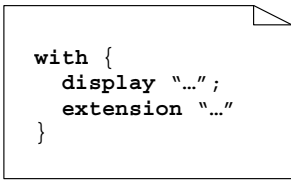
Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
		 <p>The diagram shows two rectangular boxes representing entities. A solid vertical line descends from the left box, and a dashed vertical line descends from the right box. A horizontal arrow points from the right box to the left box. The text <code>catch MyProc</code> is centered above the arrow, <code>ExceptionType</code> is centered below the arrow, and <code>MyTemplateRef</code> is centered below the arrow.</p>	<p>Catch an exception from a previous call. The exception has to match the conditions defined by the exception type and the template reference.</p> <p>Optional <b>from-</b>, <b>value-</b>, and <b>sender-</b>directives may be present to identify the sender of the exception, to retrieve the exception value or to retrieve the identifier of the peer entity.</p>
		 <p>The diagram shows two rectangular boxes representing entities. A solid vertical line descends from the left box, and a dashed vertical line descends from the right box. A horizontal arrow points from the right box to the left box. The text <code>catch MyProc</code> is centered above the arrow, <code>ExceptionType</code> is centered below the arrow, and <code>{...}</code> is centered below the arrow.</p>	<p>Catch an exception from a previous call. The exception has to match the conditions defined by the exception type and the inline template definition.</p> <p>Optional <b>from-</b>, <b>value-</b>, and <b>sender-</b>directives may be present to identify the sender of the exception, to retrieve the exception value or to retrieve the identifier of the peer entity.</p>
		 <p>The diagram shows two rectangular boxes representing entities. A solid vertical line descends from the left box, and a dashed vertical line descends from the right box. A horizontal arrow points from the right box to the left box. The text <code>catch</code> is centered above the arrow.</p>	<p>Catch any exception from any previous call.</p> <p>Optional <b>from-</b>, <b>value-</b> and <b>sender-</b>directives may be present to identify the sender of the exception, to retrieve the exception value (and assign it to a variable of type <b>anytype</b>) or to retrieve the identifier of the peer entity.</p>



Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
			<p>Catch any exception from any previous call at any port.</p> <p>The exception to be received from any port may be restricted by means referring to templates or by using inline templates.</p> <p>Optional <b>from-</b>, <b>value-</b>, and <b>sender-</b>directives may be present to identify the sender of the exception, to retrieve the exception value or to retrieve the identifier of the peer entity.</p>
Check (current) message/call received	<b>check</b>	 <p>Can be used also in combination with <code>getcall</code>, <code>getreply</code>, and <code>catch</code></p>	<p>with template, without type.</p> <p>with template, with type.</p> <p>without template, without type (any message from that port).</p> <p>with template, without type, without port (this message from that port).</p>

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
Check current message, call, reply or exception	<b>check</b>		<p>Check if a message with a value defined by a template reference has been received.</p> <p>The syntax follows the syntax for the reception of messages.</p> <p>NOTE 8 – Check may also be used in combination with <b>getcall</b>, <b>getreply</b> and <b>catch</b>.</p>
			<p>Check if a message with a value defined by a template reference has been received.</p> <p>The syntax follows the syntax for the reception of messages.</p> <p>NOTE 9 – Check may also be used in combination with <b>getcall</b>, <b>getreply</b> and <b>catch</b>.</p>
			<p>Check if a message with a value defined by an inline template definition has been received.</p> <p>The syntax follows the syntax for the reception of messages.</p> <p>NOTE 10 – Check may also be used in combination with <b>getcall</b>, <b>getreply</b> and <b>catch</b>.</p>
			<p>Check if any message (no value and no type is specified) has been received.</p> <p>The syntax follows the syntax for the reception of messages.</p> <p>NOTE 11 – Check may also be used in combination with <b>getcall</b>, <b>getreply</b> and <b>catch</b>.</p>

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
			<p>Check if any message (no value and no type is specified) has been received at any port.</p> <p>The syntax follows the syntax for the reception of messages.</p> <p>NOTE 12 – Check may also be used in combination with <b>getcall</b>, <b>getreply</b> and <b>catch</b>.</p>
Clear port	<b>clear</b>		<p>The <b>clear</b> port statement is put into a condition symbol. The condition shall cover the instance of the port to be cleared only.</p>
Clear and give access to port	<b>start</b>		<p>The <b>start</b> port statement is put into a condition symbol. The condition shall cover the instance of the port to be started only.</p>
Stop access (receiving and sending) at port	<b>stop</b>		<p>The <b>stop</b> statement is put into a condition symbol. The condition shall cover the instance of the port to be stopped only.</p>
<b>Timer operations</b>			
Start timer	<b>start</b>		
Stop timer	<b>stop</b>		

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
Read elapsed time	<b>read</b>		No special GFT symbol, used within statements or expressions.
Check if timer running	<b>running</b>		No special GFT symbol, used within statements or expressions.
Timeout operation	<b>timeout</b>		
Set local verdict	<b>verdict.set</b>		The verdict is put into a condition symbol.
Get local verdict	<b>verdict.get</b>		No special GFT symbol, used within statements or expressions.
<b>SUT operations</b>			
Remote action to be done by the SUT	<b>sut.action</b>		The <b>action</b> statement is put into an action box.
<b>Execution of test cases</b>			
Execute test case	<b>execute</b>		The <b>execute</b> statement is put into a testcase execution symbol.
<b>Attributes</b>			
Definition of attributes for control, testcases, teststeps and functions	<b>with</b>		The <b>with</b> statement is put into a text symbol.

Language element	Associated keyword	GFT symbols, if existent, and typical usage	Explanation
<b>Comments</b>			
Comments within text		<pre>/* My several lines comment */ // My single line comment</pre>	Can be used wherever text can be placed.
Comments for instance events		<pre>----- /* My instance event comment */</pre>	Shall be attached to events on a control, test component or port instance.
Comments control, test case, function or test step diagrams		<pre>/* My Comment explains a little bit more */</pre>	Shall be attached to events on a control, test component or port instance.

# Annex C

## Examples

(This annex forms an integral part of this Recommendation)

### C.1 The Restaurant example

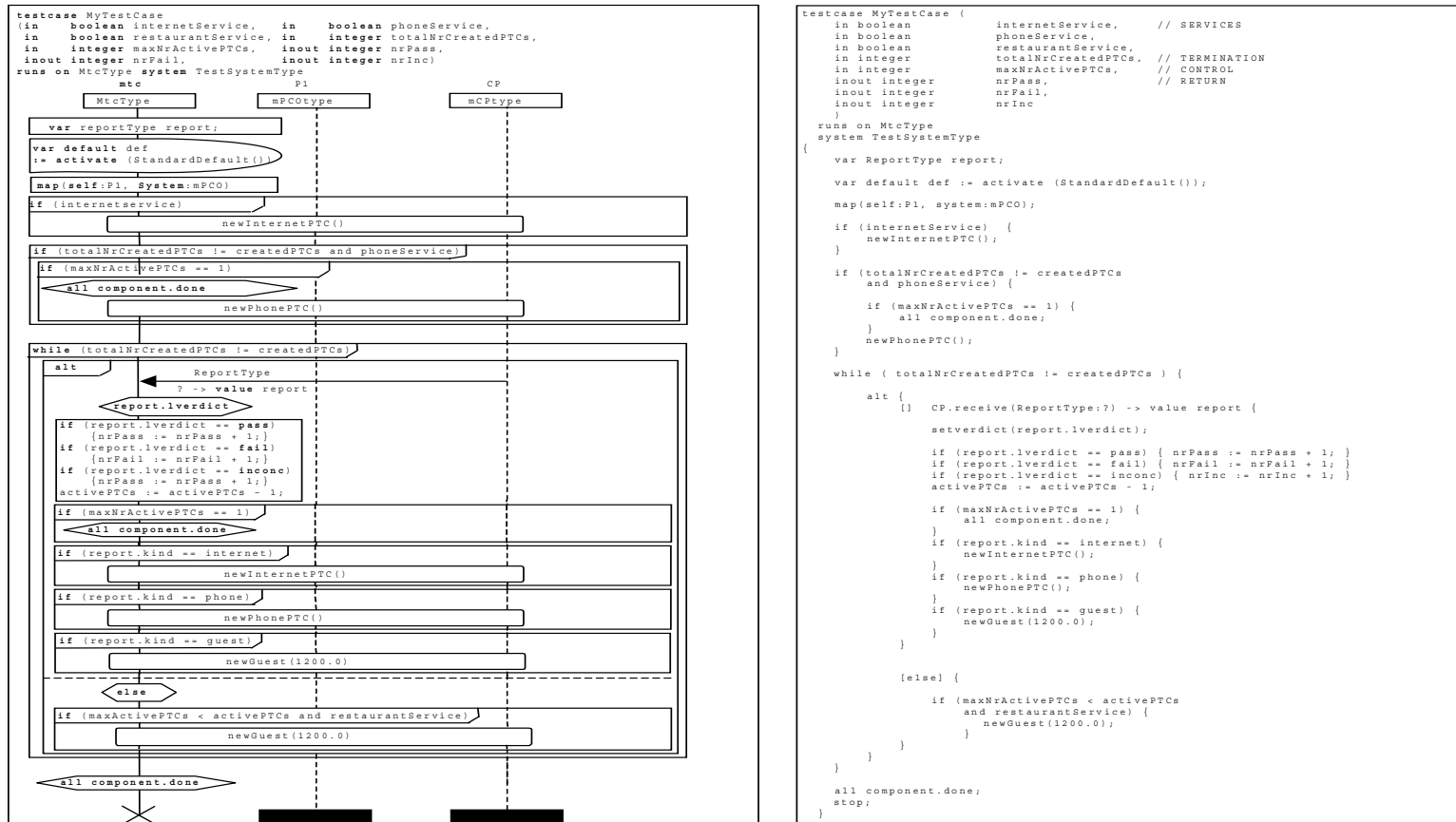
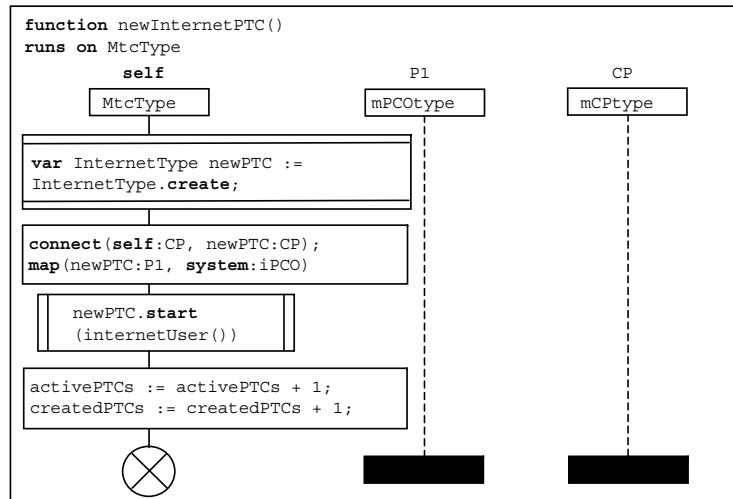


Figure C.1 – Restaurant example – MyTestCase test case



```

function newInternetPTC ()
runs on MtcType {

var InternetType newPTC := InternetType.create;

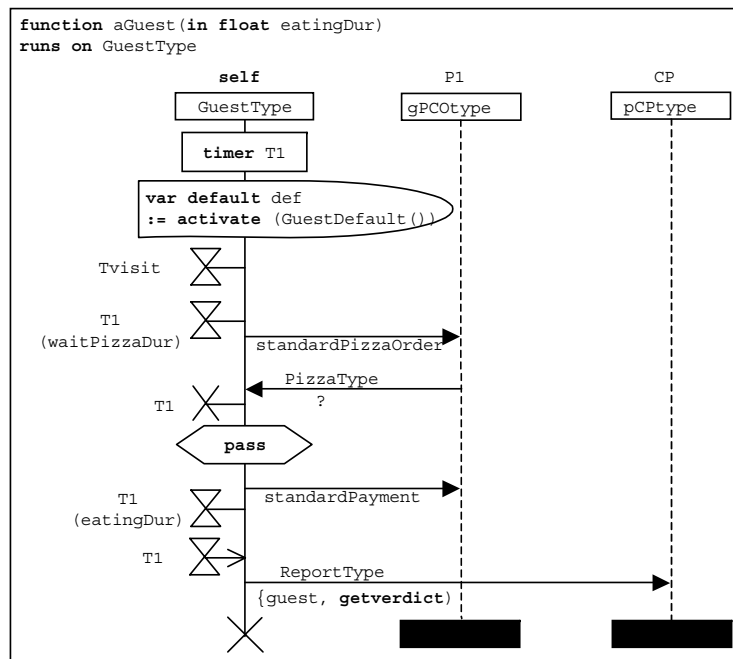
connect(self:CP, newPTC:CP);
map(newPTC:P1, system:iPCO);

newPTC.start (internetUser());

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

return;
}

```



```

function aGuest (in float eatingDur) runs on GuestType {

timer T1;

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

```

Figure C.2 – Restaurant example – newInternetPTC and aGuest functions

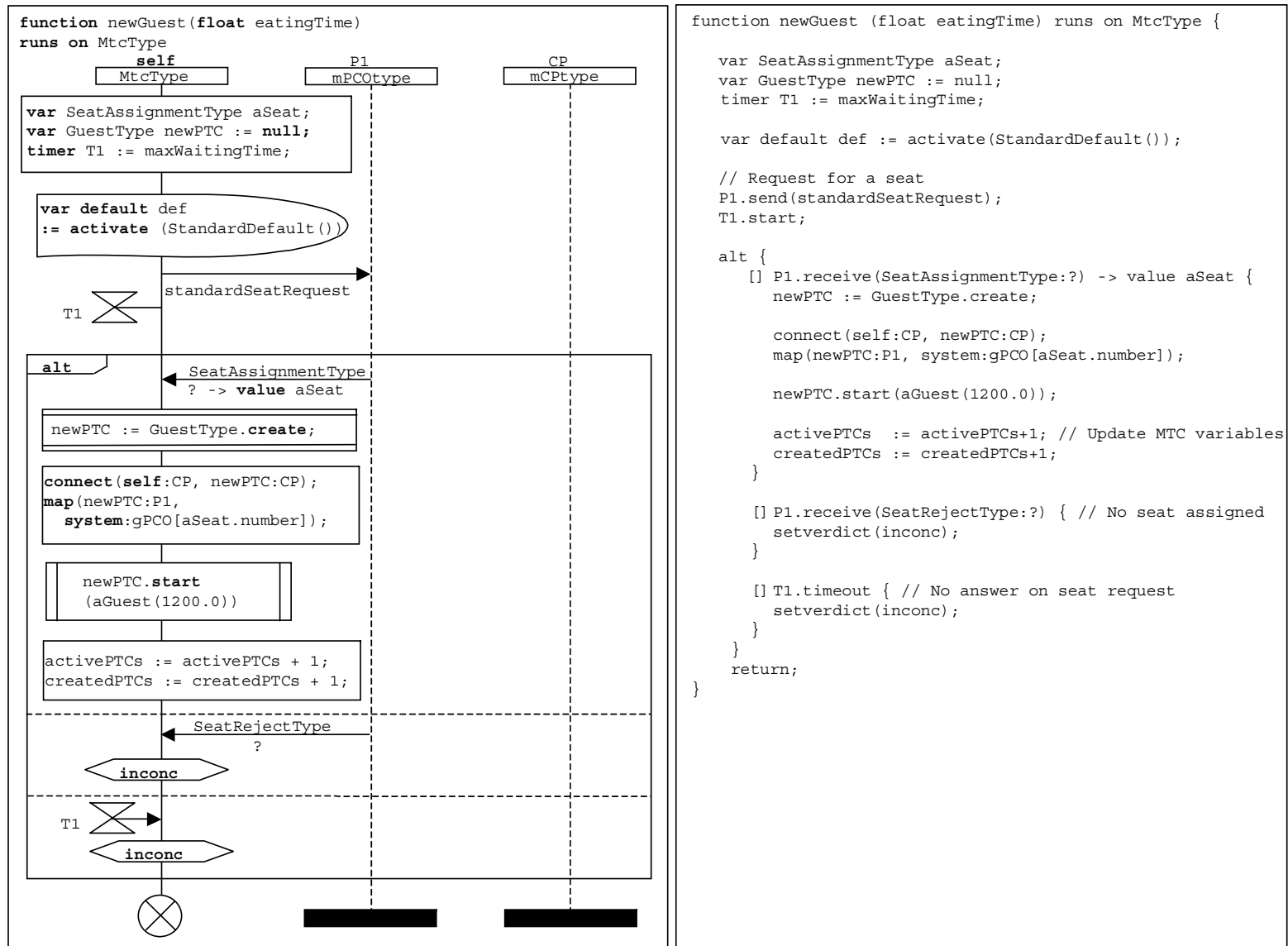


Figure C.3 – Restaurant example – newGuest function



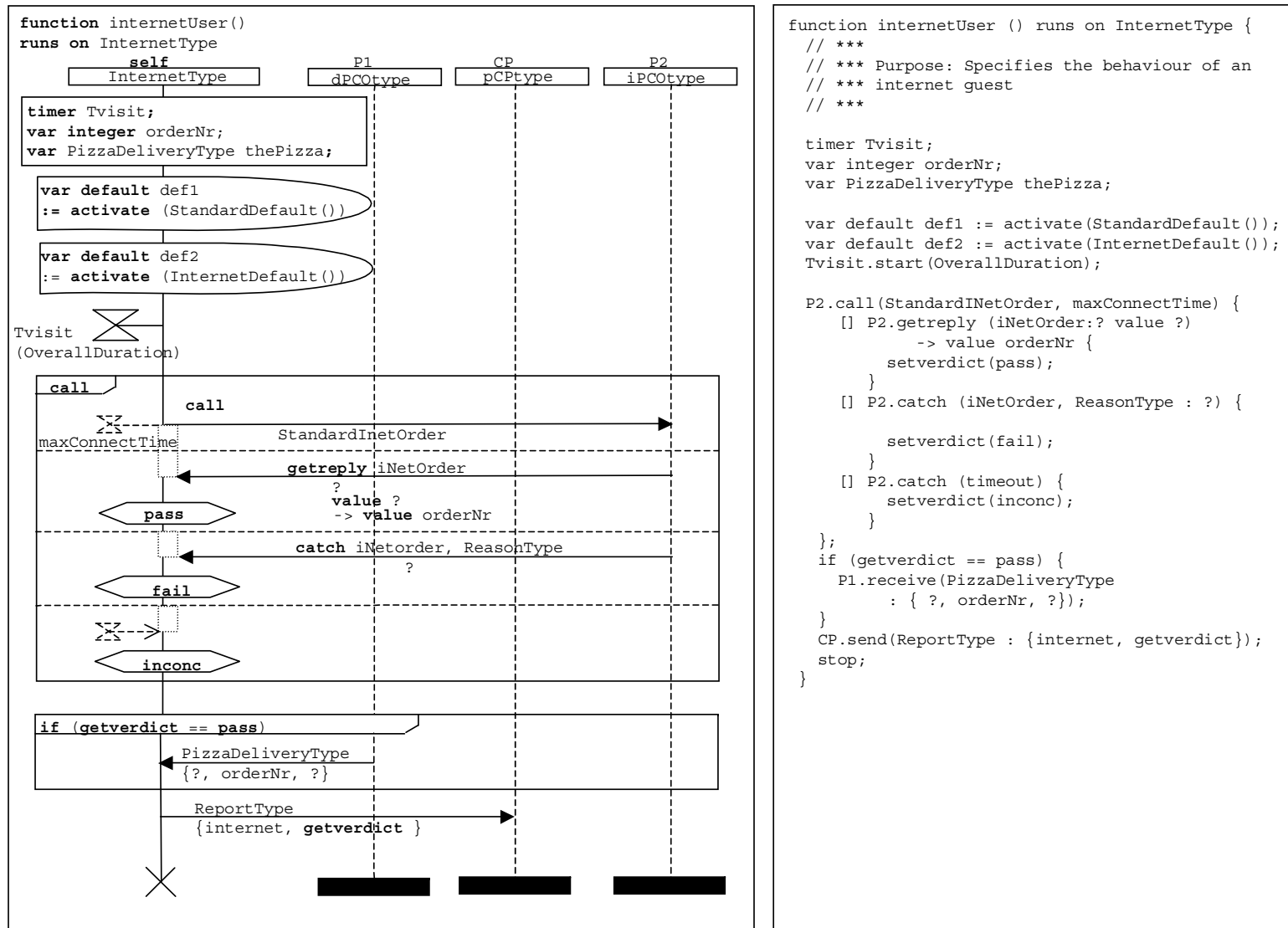
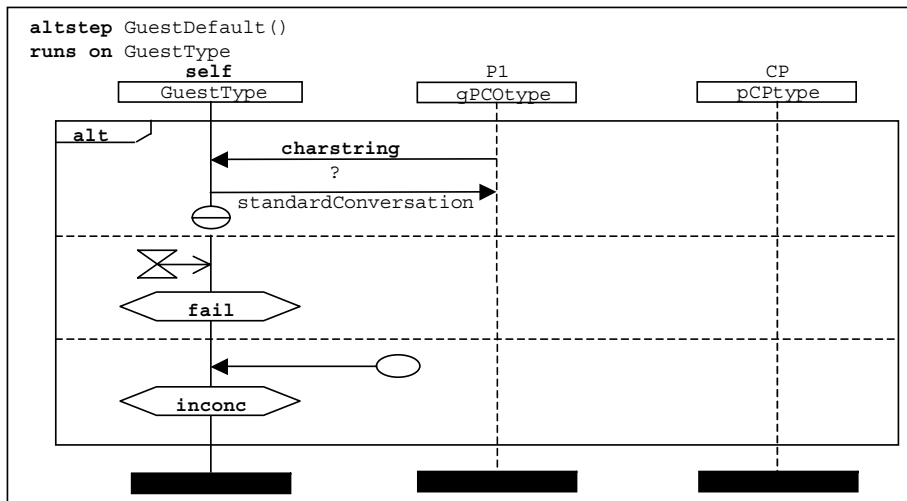


Figure C.4 – Restaurant example – internetUser function



```

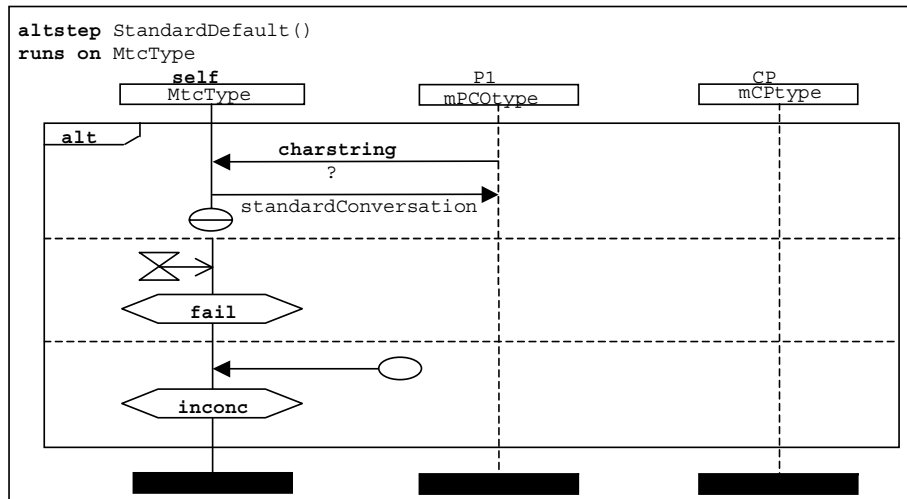
altstep GuestDefault() runs on GuestType {
// ***
// *** Purpose: Default behaviour for
// *** message based ports
// ***

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

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

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

```



```

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

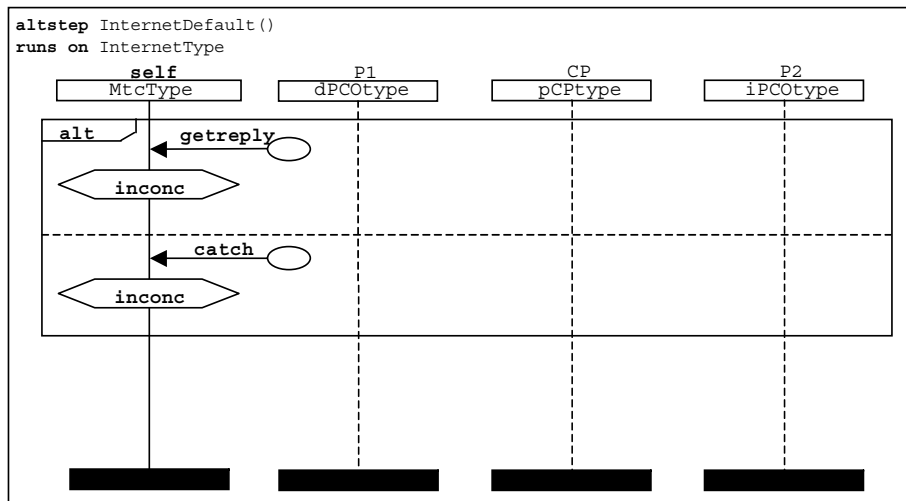
[] P1.receive(charstring : ?) {
P1.send(standardConversation);
repeat;
}

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

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

```

Figure C.5 – Restaurant example – GuestDefault and StandardDefault functions



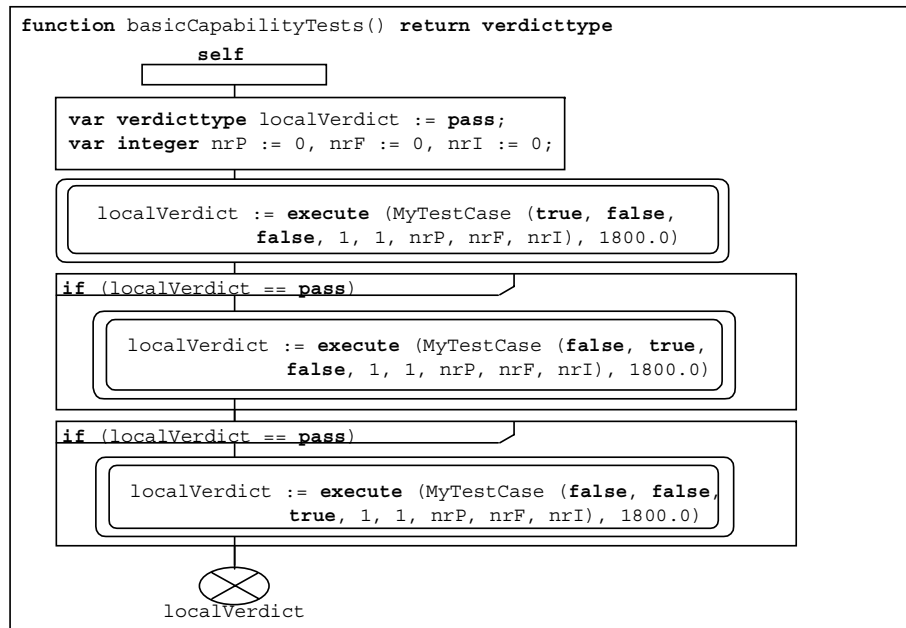
```

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

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

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

```



```

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

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

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

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

```

Figure C.6 – Restaurant example – internetDefault altstep and basicCapabilityTests functions

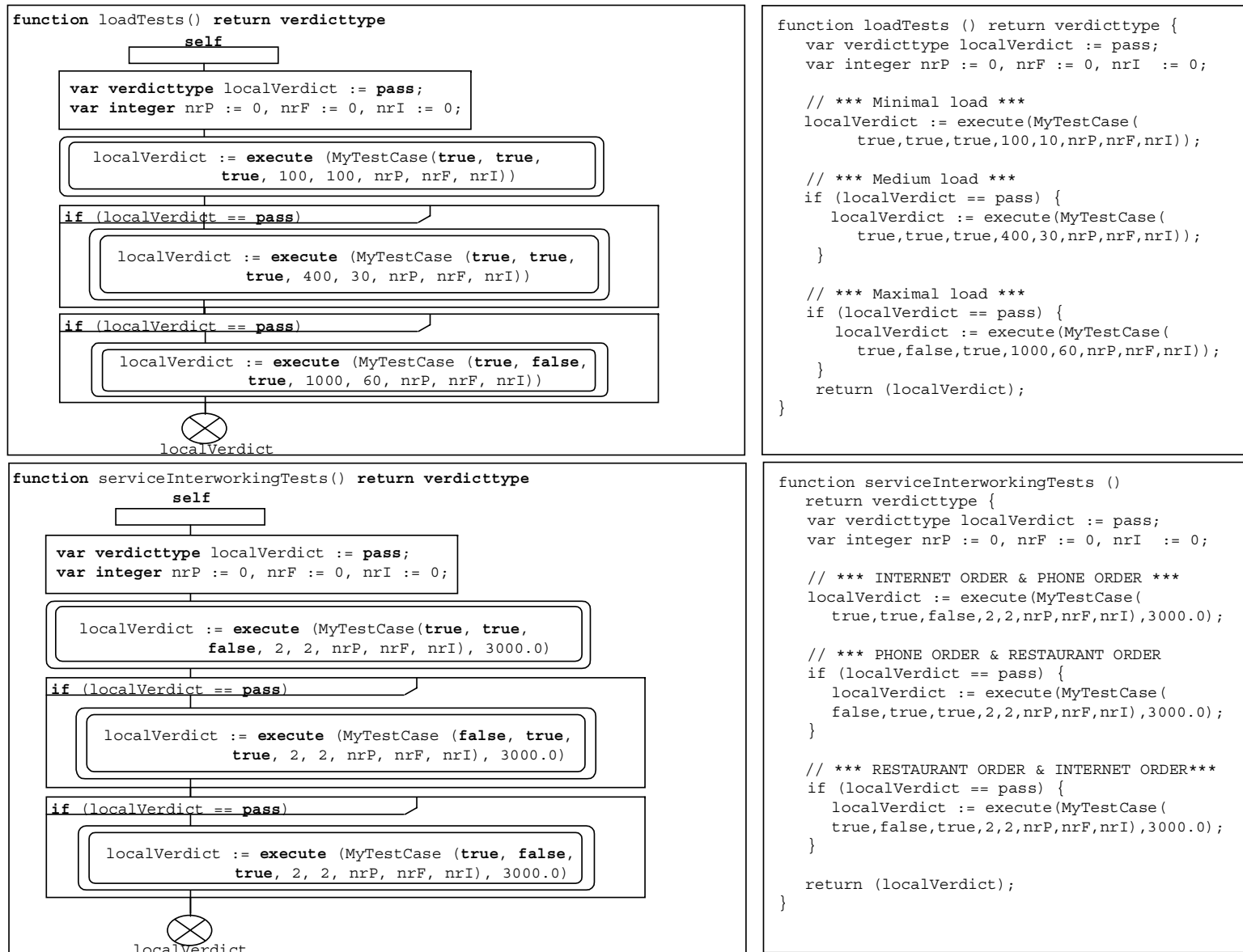
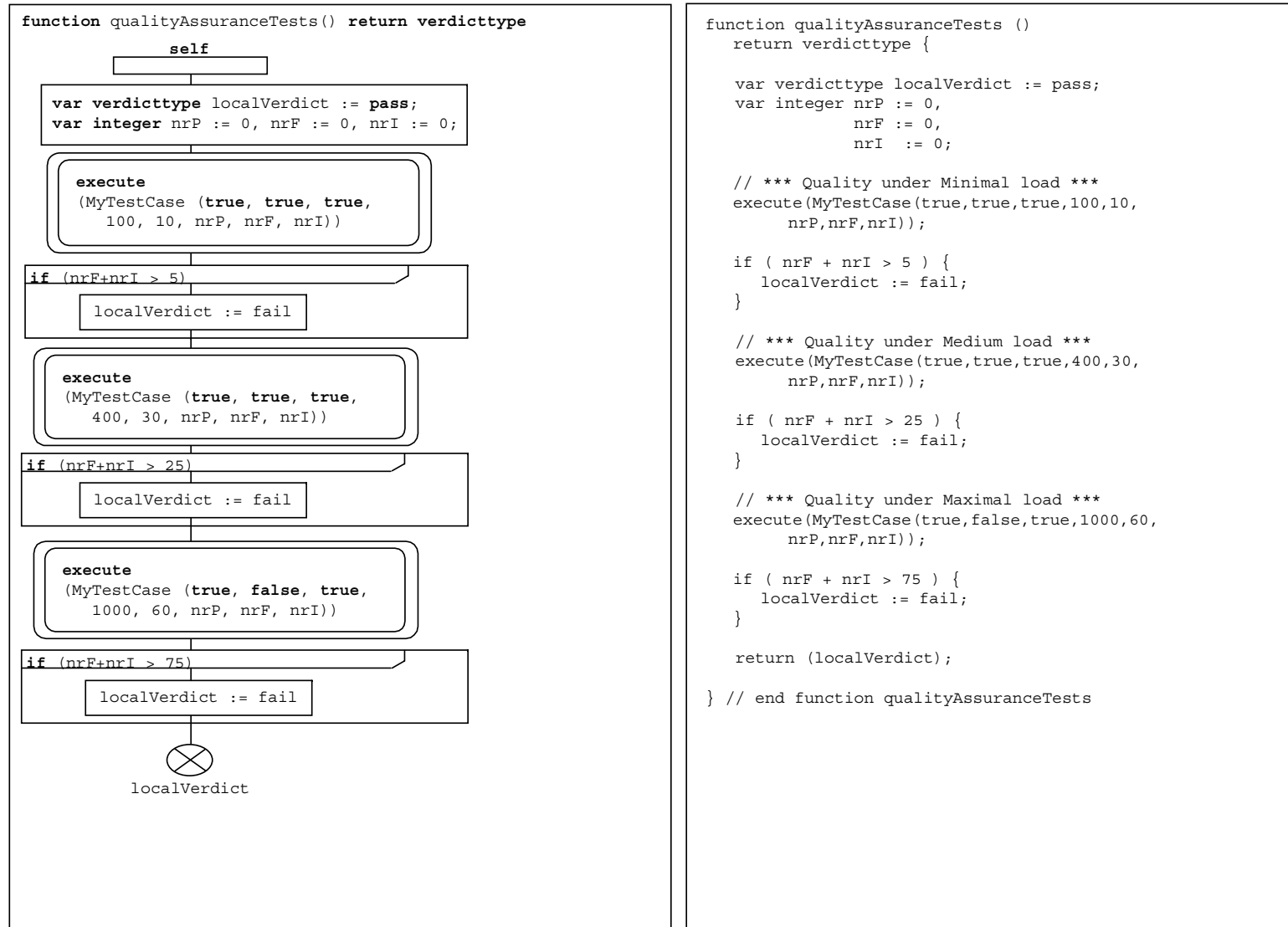
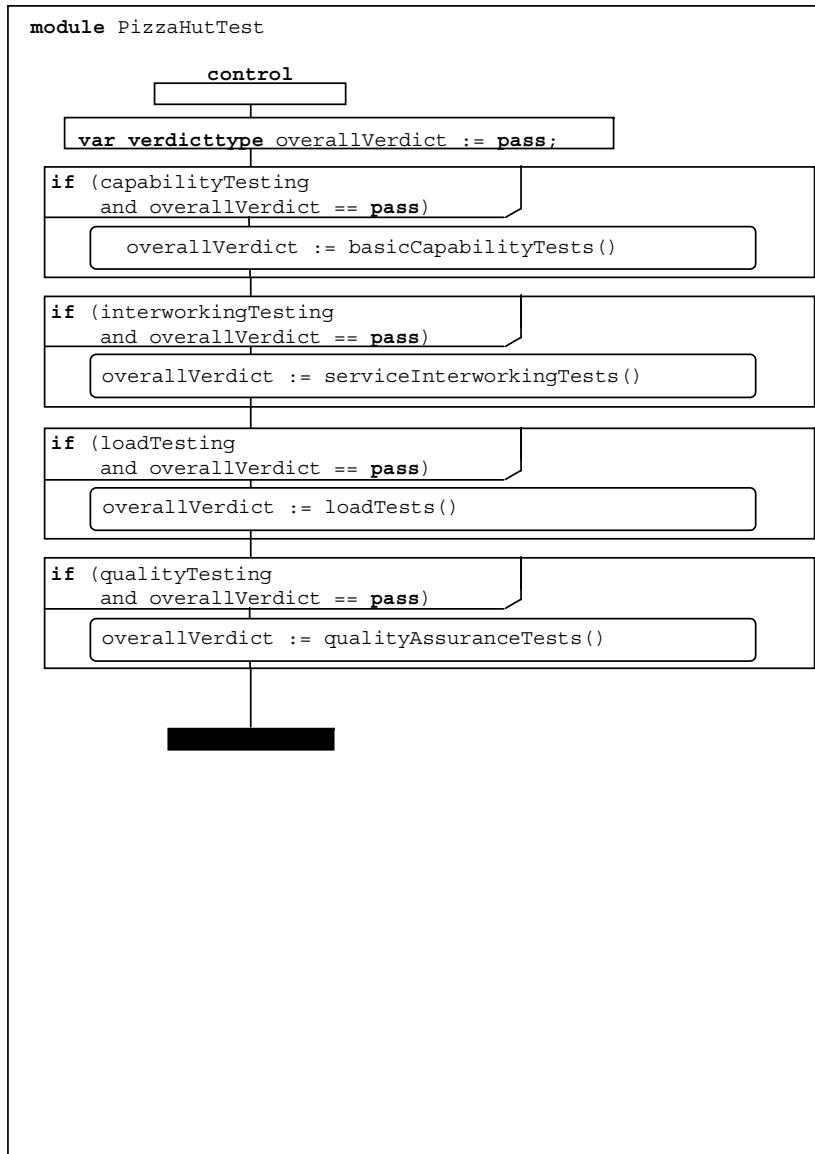


Figure C.7 – Restaurant example – loadTests and serviceInterworkingTests functions



**Figure C.8 – Restaurant example – qualityAssuranceTests**



```

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

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

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

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

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

```

**Figure C.9 – Restaurant example – PizzaHutTest module**

## C.2 The INRES example

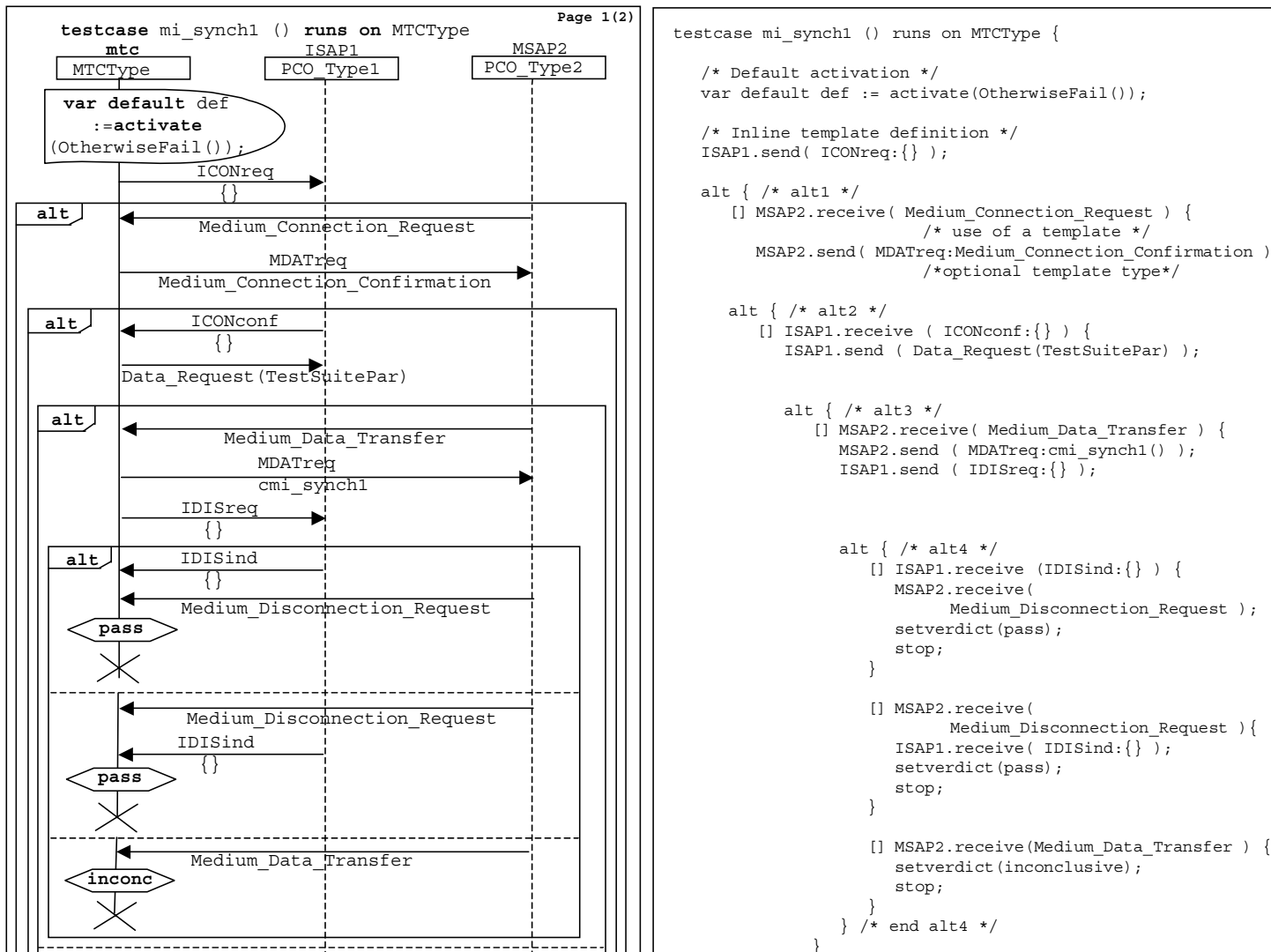


Figure C.10 – INRES example – mi\_synch1 1(2) test case

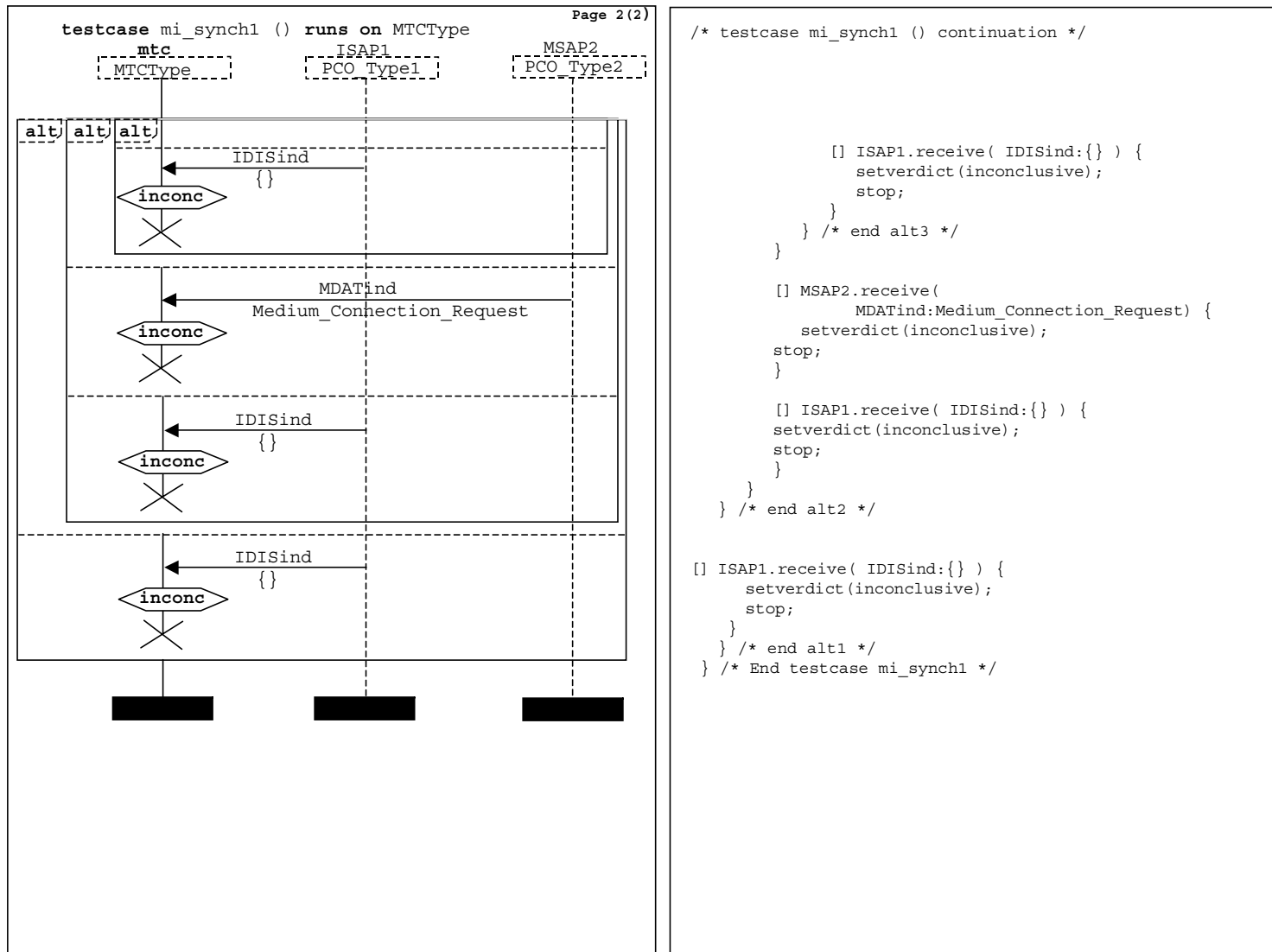
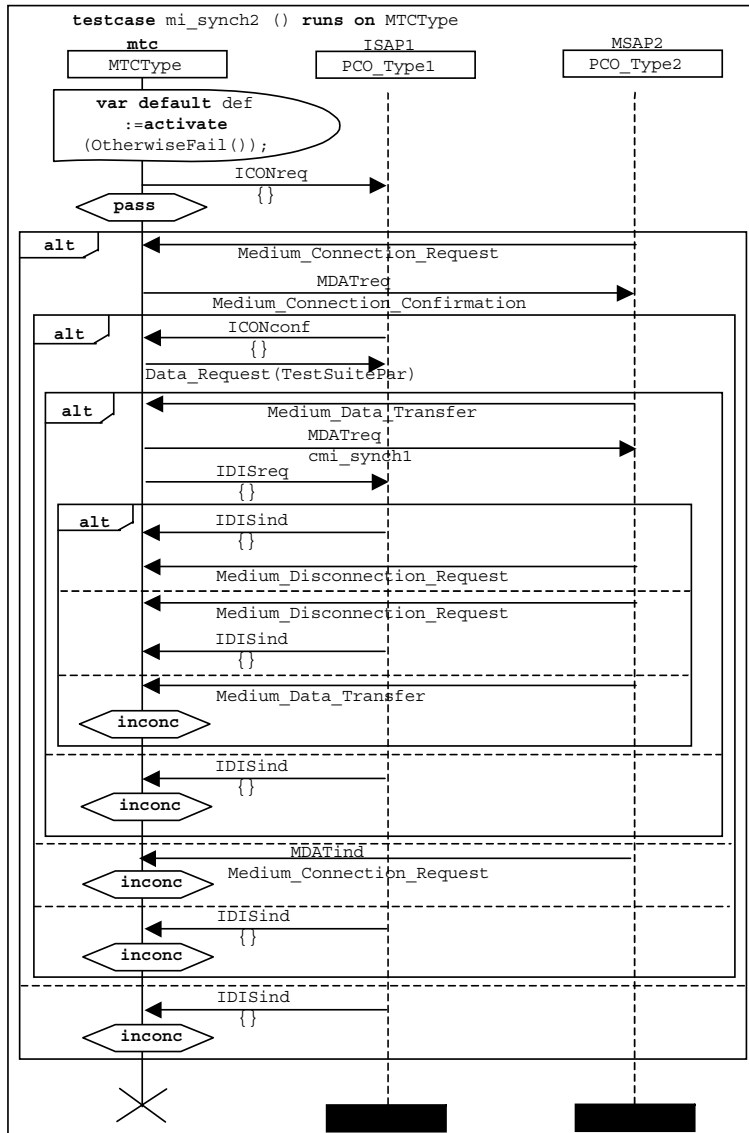


Figure C.11 – INRES example – mi\_synch1 2(2) test case





```
testcase mi_synch2 () runs on MTCType {

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

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

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

Figure C.12 – INRES example – mi\_synch2 test case

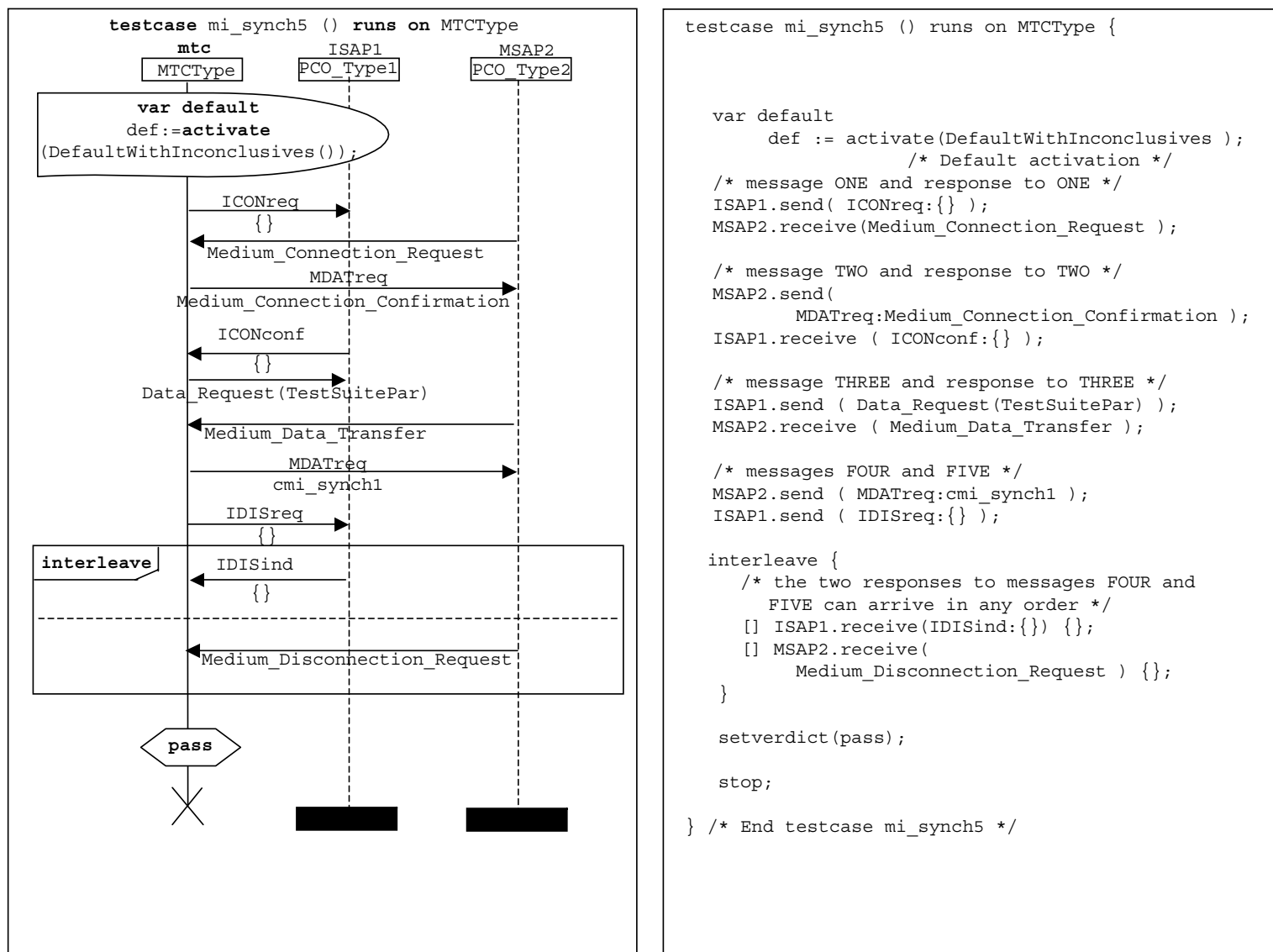


Figure C.13 – INRES example – mi\_synch5 test case

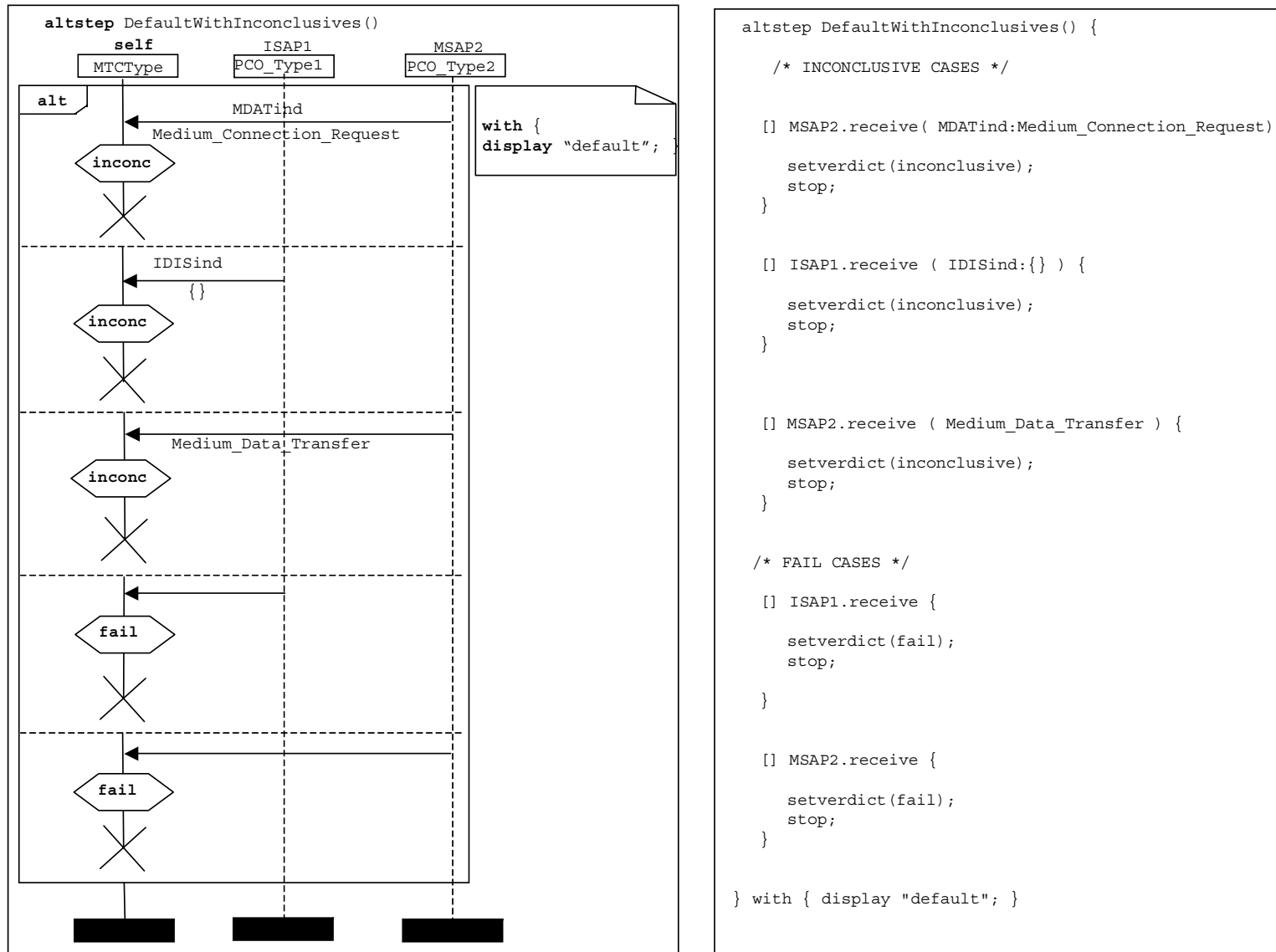


Figure C.14 – INRES example – DefaultWithInconclusives altstep

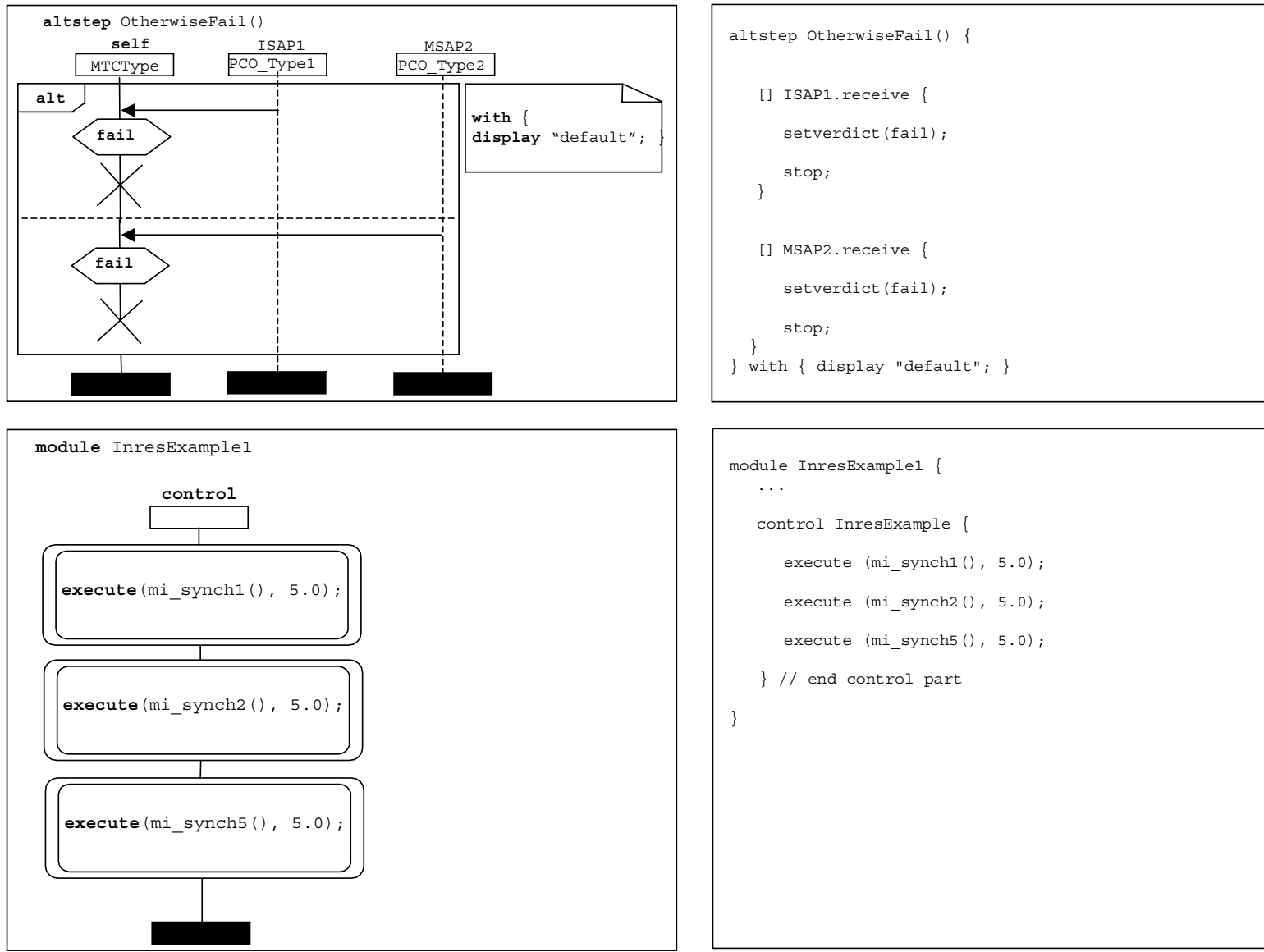


Figure C.15 – INRES example – OtherwiseFail altstep and InresExample1 module definitions



## SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	Telecommunication management, including TMN and network maintenance
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks, open system communications and security
Series Y	Global information infrastructure, Internet protocol aspects and next-generation networks
<b>Series Z</b>	<b>Languages and general software aspects for telecommunication systems</b>