ETSI ES 201 873-3 V2.2.0 (2002-03)

ETSI Standard

Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 3: Graphical Presentation Format for TTCN-3 (GFT)

Reference
DTR/MTS-00063-3r1

Keywords
ASN.1, methodology, MSC, MTS, testing, TTCN

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Contents

Intellectual Property Rights

IPRs essential or potentially essential to the present document may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (http://www.etsi.org/ipr).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Foreword

This ETSI standard (ES) has been produced by ETSI Technical Committee Methods for Testing and Specification (MTS).

Introduction

The graphical presentation format of TTCN-3 (GFT) is based on the ITU-T Recommendation Z.120 [4] defining Message Sequence Charts (MSC). GFT uses a subset of MSC with test specific extensions and extensions of a general nature. 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 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 ES 201 873-1 [1] and provides a full text-based syntax, static semantics and operational semantics as well as defining the use of the language with ASN.1. The GFT presentation format provides an alternative way of displaying the core language (Figure 1).

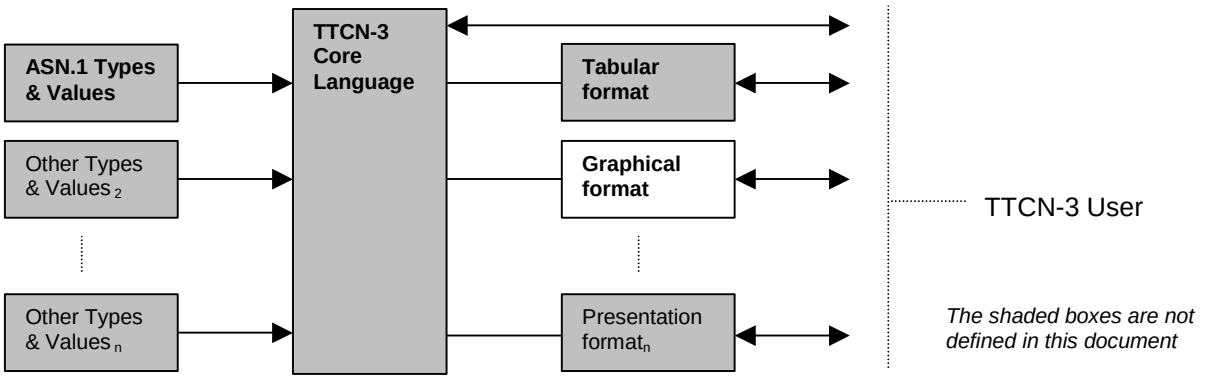


Figure 1: User's view of the core language and the various presentation formats

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.

The present document 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 notation.

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

1 Scope

The present document defines the Message Sequence Chart (MSC) presentation format for the TTCN-3 core language as defined in ES 201 873-1 [1]. This presentation format uses a subset of Message Sequence Charts as defined in [3] with test specific extensions and extensions of general nature.

The present document is based on the core TTCN-3 language defined in ES 201 873-1 [1]. 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 the present document.

2 References

For the purposes of this Technical Report (TR) the following references apply:

[1] ETSI ES 201 873-1: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language".

[2] ETSI ES 201 873-2: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 2: TTCN-3 Tabular Presentation Format (TFT)".

[3] ITU-T Recommendation Z.120 (1999): "Message Sequence Chart (MSC)".

[4] ISO/IEC 9646-3 (1994): "Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation (TTCN)".

3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

BNF	Backus-Naur Form
CATG	Computer Aided Test Generation
ETSI	European Telecommunication Standards Institute
GFT	Graphical Format of TTCN-3
HMSC	High-level Message Sequence Chart
MSC	Message Sequence Chart
MTC	Main Test Component
PTC	Parallel Test Component
SUT	System Under Test
TFT	Tabular Format of TTCN-3
TTCN	Testing and Test Control Notation

4 Overview

According to the OSI conformance testing methodology defined in ISO/IEC 9646-3 [4], 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 actions of 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.

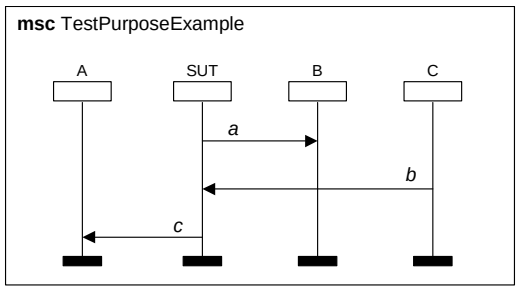


Figure 2: MSC describing the interaction of an SUT with its interfaces

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 Figure 2 and Figure 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.

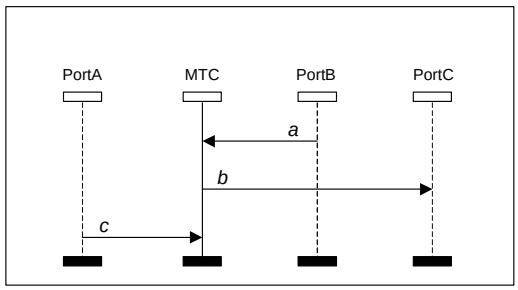


Figure 3: MSC describing the interaction of an MTC with SUT interfaces

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 notation 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.

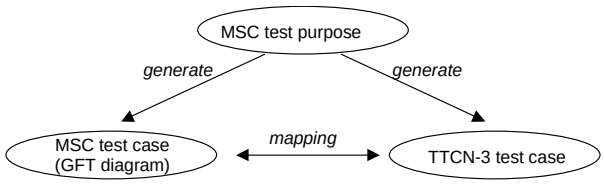


Figure 4: Relations between MSC test purpose description, MSC test case descriptions and TTCN-3

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

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 section 7 Module structure).

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

- 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 notation or in the tabular presentation format for TTCN-3 (TFT) [2].

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:

GFT is based on MSC [3] 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.

New GFT symbols have been defined for

- 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 the Section 8.

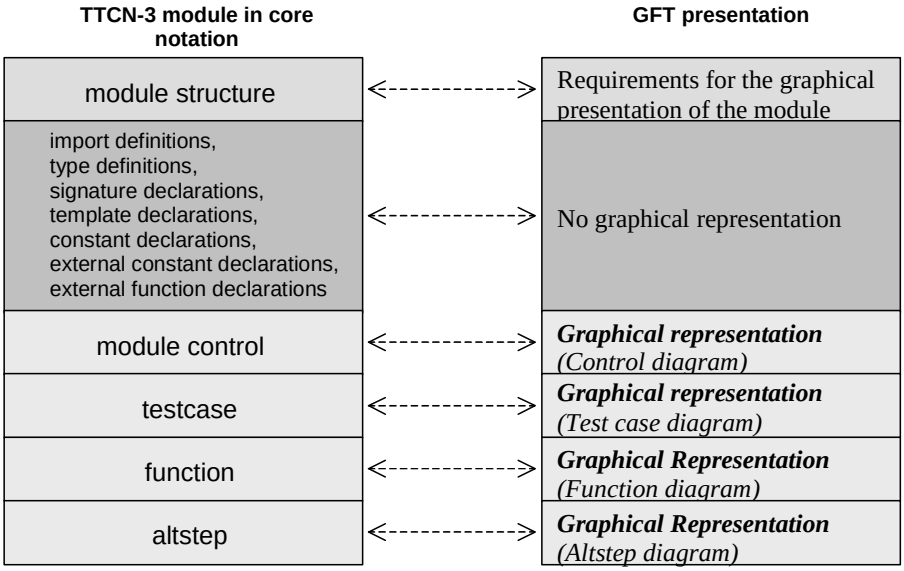


Figure 5: Relation between TTCN-3 core notation and the corresponding GFT description

6 Mapping between GFT and TTCN-3 Core Notation

GFT provides graphical means for TTCN-3 behaviour definitions. The control part and each function, altstep and test case of a TTCN-3 core notation 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 notation 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: GFT provides no graphical presentations for definitions of types, constants, signatures, templates, external constants and external functions in the module definitions part. These definitions may be presented directly in core notation 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: 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.

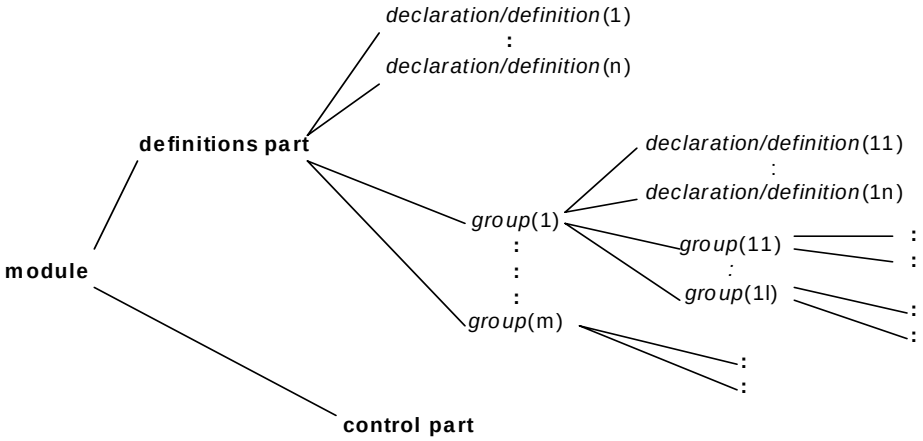


Figure 6: Structure of TTCN-3 modules

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 notation, in form of a TFT table and as GFT diagram.

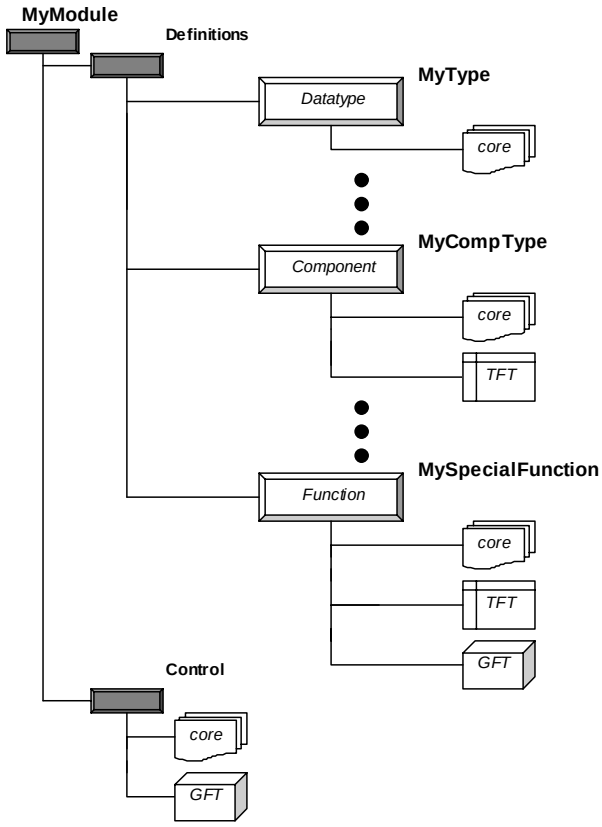


Figure 7: Various presentation formats in an organizer view of a TTCN-3 module structure

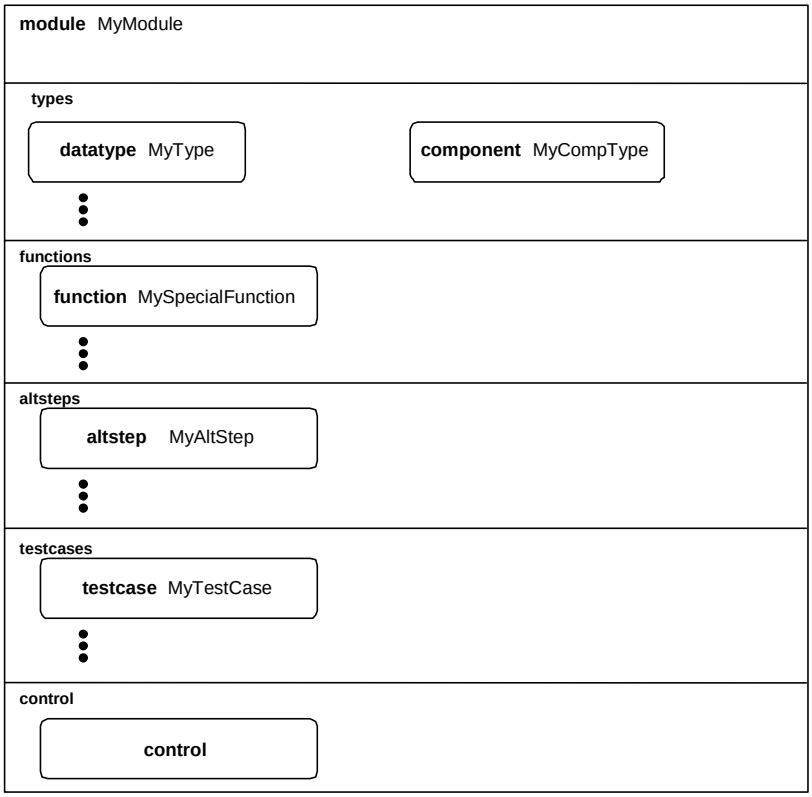
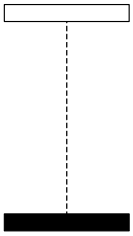
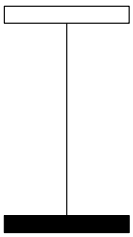
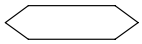
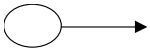
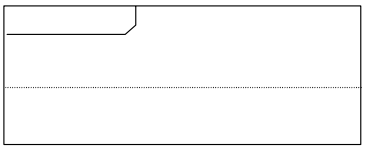
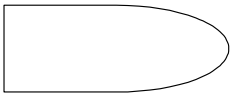
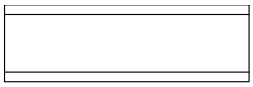


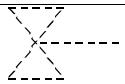
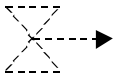
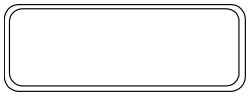
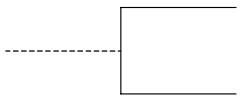
Figure 8: Graphical MSC document-like presentation of a TTCN-3 module structure

8 GFT symbols

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

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

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
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:

a) *Control diagram* for the graphical presentation of a TTCN-3 module control part,

b) *Test case diagram* for the graphical presentation of a TTCN-3 test case,

c) *Altstep diagram* for the graphical presentation of a TTCN-3 altstep, and

d) *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

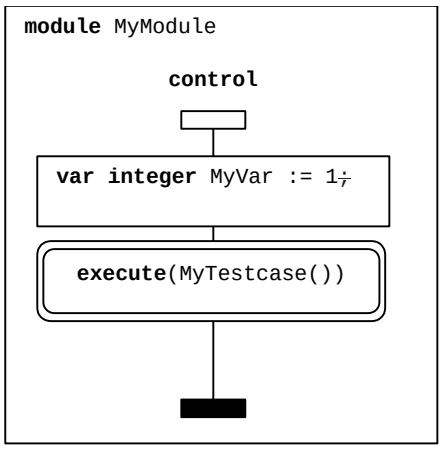
GFT diagrams may be organized in pages and a large GFT diagram may be split into several pages. Each page of a split diagram shall have a numbering in the upper right hand corner that identifies the page uniquely. The numbering is optional if the diagram is not split.

NOTE 1: The concrete numbering scheme is considered to be a tools issue and therefore is outside the scope of this standard. 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 a tools issue and therefore are outside the scope of this standard. 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 presented 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.




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

GFT **Core**

Figure 9: Principle shape of a GFT control diagram and corresponding core notation

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 sections below except that for the module control part the graphical symbols are attached to the control instance and not to a test component instance.

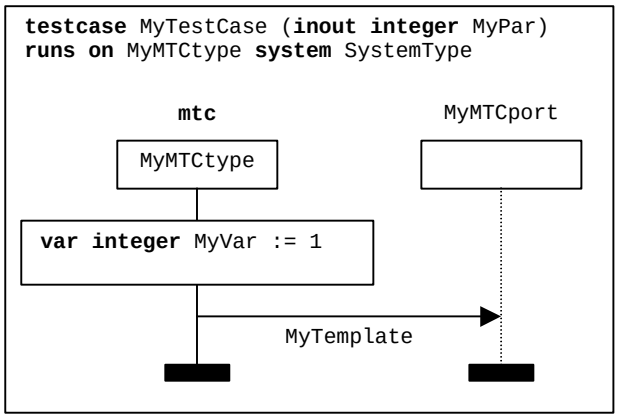
Please refer to clause 11.4 for the GFT representation of expressions, assignments, **log**, **label** and **goto**, **if-else**, **for** loop, **while** loop, **do while** loop, and **stop**, to clause 11.9 for timer operations and to clause 9.4 and clause 11.2.2 for functions and their invocation.

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 notation. If the system clause is specified in the corresponding core notation, 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.




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

    var integer MyVar := 1;

    MyMTCPort.send(MyTemplate);

        :
        :
}
```

GFT **Core**

Figure 10: Principle shape of a GFT test case diagram and corresponding core notation

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 notation. If these clauses are specified in the corresponding core notation, 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.

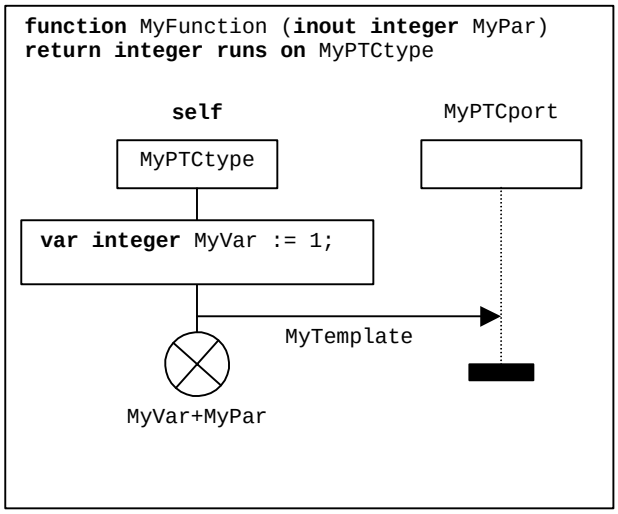
NOTE: The names and types of the ports that are usable by a function are passed in as parameters or are the port names and types that are defined in the component type definition referenced in the **runs on** clause.

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.

The **self** and port types are displayed in the component or 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.




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

        var integer MyVar := 1;

        MyPTCport.send(MyTemplate);

        :
        :

        return MyVar+MyPar
}
```

GFT **Core**

Figure 11: Principle shape of a GFT function diagram and corresponding core notation

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 notation. If the **runs on** clause is specified in the corresponding core notation, 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.

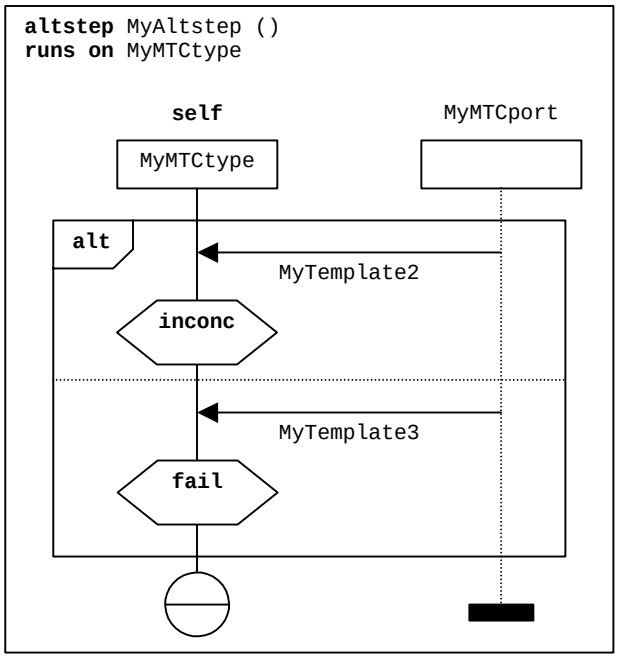
NOTE: The names and types of the ports that are usable by an altstep are passed in as parameters or are the port names and types that are defined in the component type definition referenced in the **runs on** clause.

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.

The **self** and port types are displayed in the component or port instance head symbol.

Attributes associated to the altstep presented in GFT 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 description are sketched in Figure 12.




```
altstep MyAltstep () runs on MyMTCtype {

    alt {

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

GFT **Core**

Figure 12: Principle shape of a GFT altstep diagram and corresponding core notation

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 Section 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

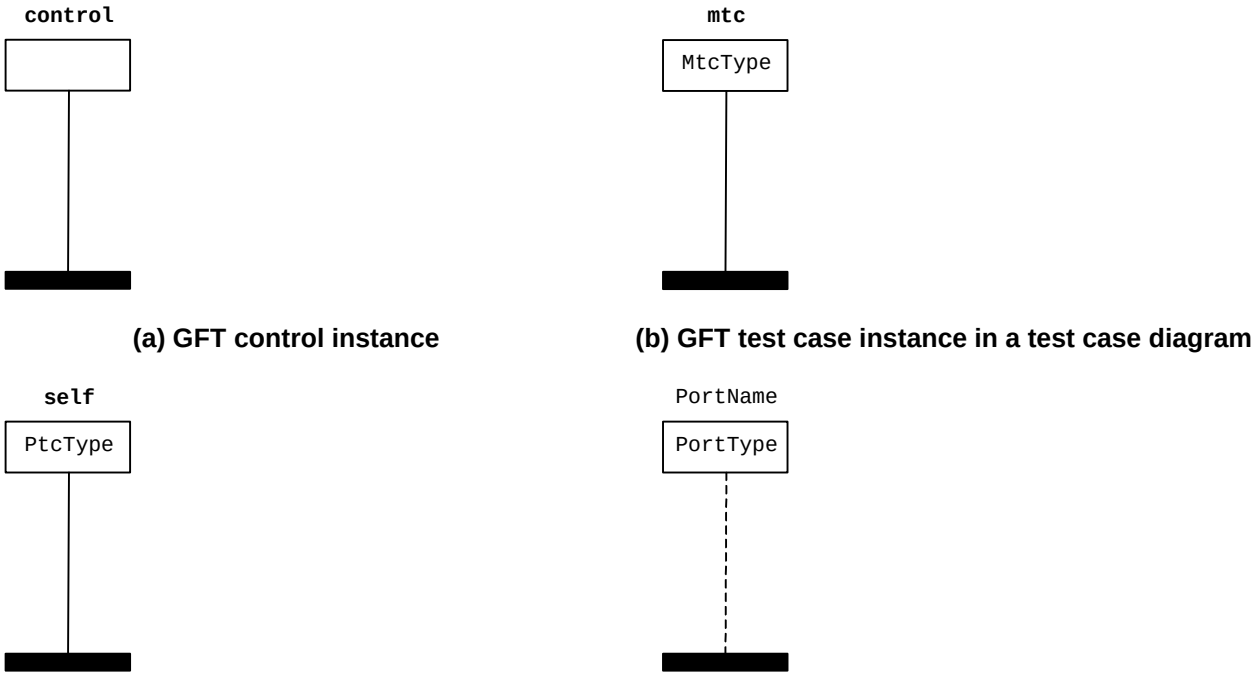
- the mandatory name **mtc** placed on top of the instance head symbol in the case of a test case diagram
- the mandatory name **self** placed on top of the instance head symbol in the case of a function or altstep diagram.

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.



(a) GFT control instance

(b) GFT test case instance in a test case diagram

(c) GFT test component instance in a function or altstep diagram

(d) GFT port instance

Figure 13: Principle shape of instance kinds in GFT diagrams

11 Elements of GFT diagrams

This section defines general drawing rules for the representation of specific TTCN3 syntax elements (semicolons, comments). It describes how to display the execution of GFT diagrams and the graphical symbols associated with TTCN3 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 notation. 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 notation (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 notation 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

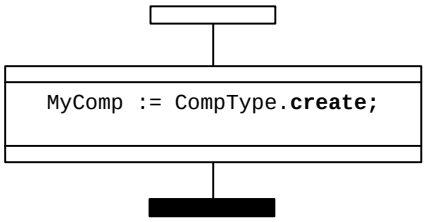
The following TTCN-3 declarations, statements and operations are specified within action symbols: declarations (with the restrictions defined in clause 11.3), assignments, **log**, **connect**, **disconnect**, **map**, **unmap** and **action**.

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.

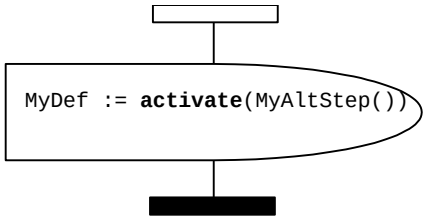
11.1.3 Comments

GFT provide three possibilities to put comments into GFT diagrams:

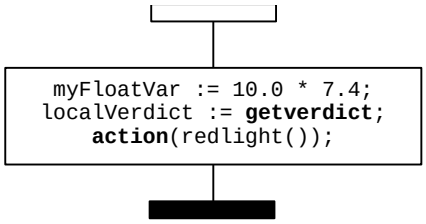
- Comments may be put into GFT symbols following the symbol inscription and using the syntax for comments of the TTCN-3 core notation (Figure 14-d).
- Comments in the syntax for comments of the TTCN-3 core notation can be put into text symbols and freely placed in the GFT diagram area (Figure 14-e).
- 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 notation need not to be used (Figure 14-f).



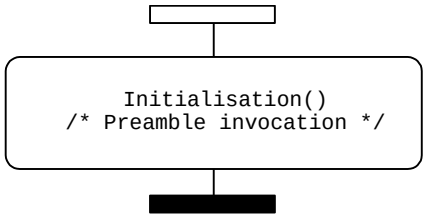
(a) Component creation with an optional terminating semicolon



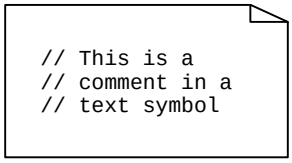
(b) Default activation without a terminating semicolon



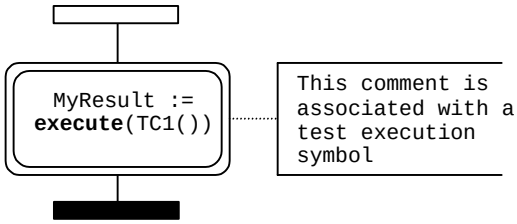
(c) Sequence of statements in an action symbol



(d) Comment within a GFT reference symbol



(e) Comment in a text symbol



(f) Comment within a comment symbol associated with an execution symbol

Figure 14: Examples for the effects of the general drawing rules

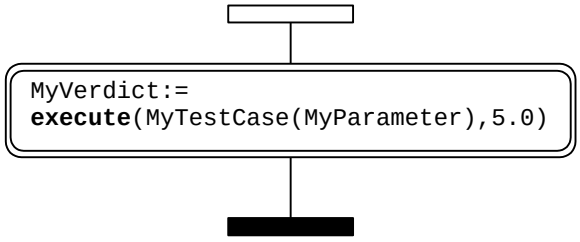
11.2 Invoking GFT diagrams

This section 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 section 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.




```
MyVerdict :=
execute(MyTestCase(MyParameter),5.0);
```

GFT **Core**

Figure 15: Test case execution

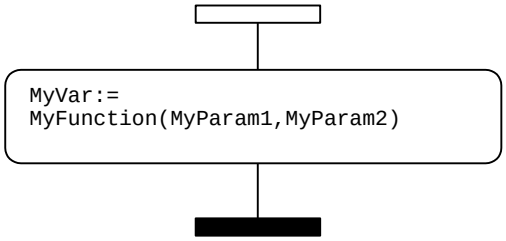
11.2.2 Invocation of functions

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 TTCN3 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.

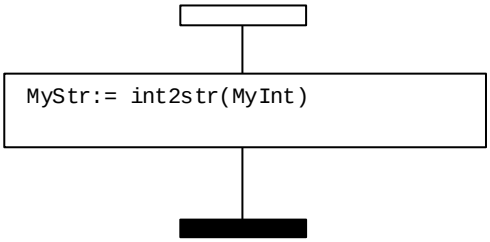
The reference symbol is only used for user defined functions defined within the current module. It shall not be used for external functions or predefined TTCN3 functions, which shall be represented in their text form within an action form (Figure 17) or other GFT symbols (see example in Figure 18).




```
MyVar:= MyFunction(MyParam1,MyParam2);
```

GFT **Core**

Figure 16: Invocation of user defined function

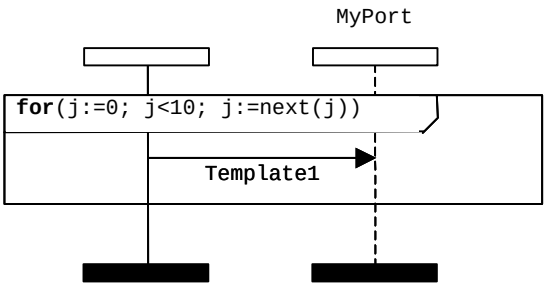



```
MyStr:= int2str(MyInt);
```

GFT **Core**

Figure 17: Invocation of predefined/external function

Functions called inside a TTCN3 construct with an associated GFT symbol are represented as text within that symbol.



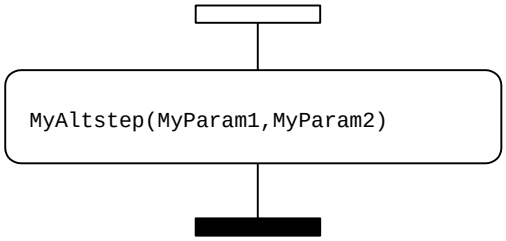

```
for(j:=0; j<10; j:=next(j)) {
   MyPort.send(Template1)
}
```

GFT **Core**

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).




```
MyAltstep(MyParam1,MyParam2);
```

GFT **Core**

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 initialisation of timers, constants and variables at the beginning of statement blocks. GFT uses the syntax of the TTCN-3 core notation for declarations in several symbols. The type of a symbol depends on the specification of the initialisation, e.g., a variable of type **default** that is initialised 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 initialisation,
- declarations of variables and constants with initialisation,

  -if the initialisation is not made by means of functions that include communication functions, or

  -if a declaration is

  - of a component type that is not initialised by means of a **create** operation,

- of type **default** that is not initialised by means of an **activate** operation,
- of type **verdicttype** that is not initialised by means of an **execute** statement,
- of any other basic type,
- of a basic string type, or
- of a user-defined structured type with fields that fulfil all restrictions mentioned in this bullet for 'declarations of variables and constants with initialisation'.

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 Figure 20-a and Figure 20-b.

11.3.2 Declaration of constants and variables within inline expression symbols

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

11.3.2 Declaration of constants and variables within create symbols

Constants and variable declarations of a component type that are initialised by means of **create** operations shall be made within a create symbols. In contrast to declarations within action symbols, each declaration that is initialised 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.

11.3.3 Declaration of constants and variables within default symbols

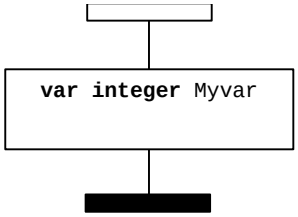
Constants and variable declarations of type **default** that are initialised by means of **activate** operations shall be made within a default symbol. In contrast to declarations within action symbols, each declaration that is initialised by means of an **activate** operation shall be presented in a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

11.3.4 Declaration of constants and variables within reference symbols

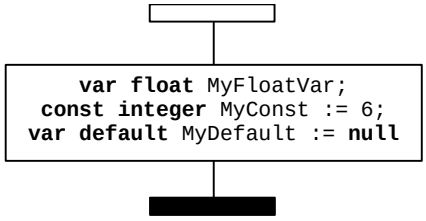
Constants and variable declarations that are initialised by means of a function that includes communication operations shall be made within reference symbols. In contrast to declarations within action symbols, each declaration that is initialised by means of a function that includes communication functions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

11.3.5 Declaration of constants and variables within execute test case symbols

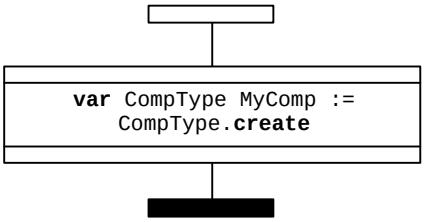
Constants and variable declarations of type **verdicttype** that are initialised by means of **execute** statements shall be made within execute test case symbols. In contrast to declarations within action symbols, each declaration that is initialised by means of an **execute** statement shall be presented 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.



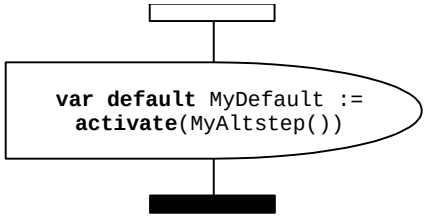
(a) Variable declaration within an action symbol



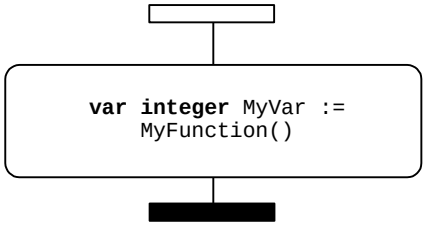
(b) Sequence of declarations within an action symbol



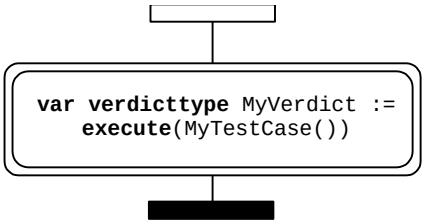
(c) Variable declaration within a create symbol



(d) Variable declaration within a default symbol



(e) Variable declaration within a reference symbol



(d) Variable declaration within an execute test case symbol

Figure 20: Examples for declarations in GFT

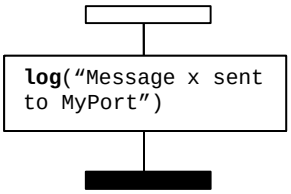
11.4 Basic program statements

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).



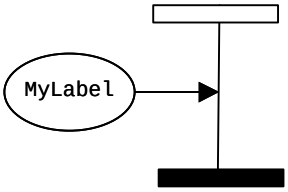

```
log("Message x sent to MyPort");
```

GFT **Core**

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`.



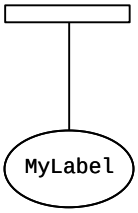

```
label MyLabel;
```

GFT **Core**

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**.



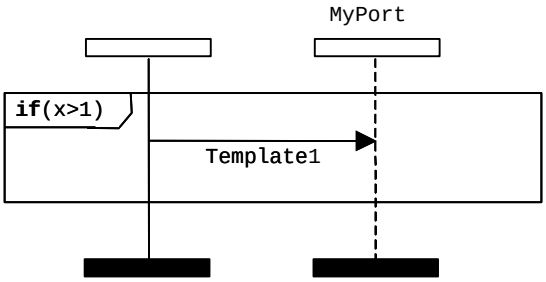

```
goto MyLabel;
```

GFT **Core**

Figure 23: Goto Statement

11.4.4 The If-else statement

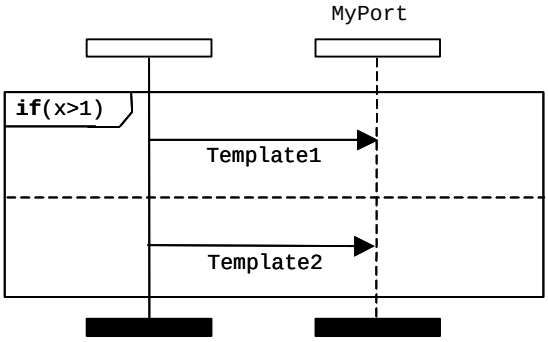
The **if-else** statement shall be represented by an in-line expression symbol labelled with the **`if`** keyword and a Boolean expression. The if-else in-line expression symbol may contain one or two operands, separated by a dashed line. Figure 24 illustrates an **`if`** statement with a single operand, 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.




```
if (x>1) {
  MyPort.send(Template1)
}
```

GFT **Core**

Figure 24: If-Statement



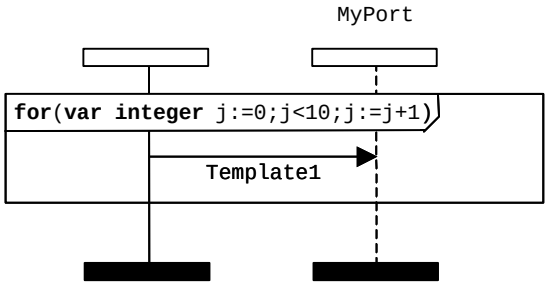

```
if (x>1) {
  MyPort.send(Template1)
}
else {
  MyPort.send(Template2)
}
```

GFT **Core**

Figure 25: If-else Statement

11.4.5 The For statement

The **for** statement shall be represented by an in-line expression symbol labelled with a **`for`** definition as defined in ES 201 873-1 [1], clause 19.7. 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 initialised within the **`for`** statement.



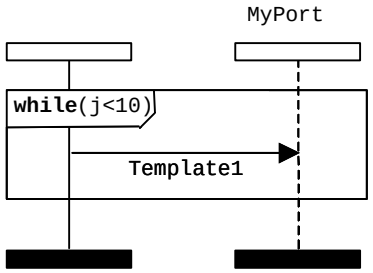

```
for(var integer j:=0;j<10;j:=j+1) {
   MyPort.send(Template1)
}
```

GFT **Core**

Figure 26: For Statement

11.4.6 The While statement

The **`while`** symbol shall be represented by an in-line expression symbol labelled with a **`while`** definition as defined in ES 201 873-1 [1], clause 19.8. The **while** body shall be represented as the operand of the while inline expression symbol. Figure 27 represents an example of a **`while`** statement.



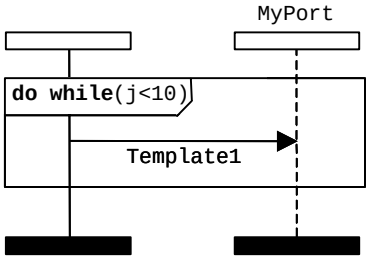

```
while(j<10) {
  MyPort.send(Template1)
}
```

GFT **Core**

Figure 27: While Statement

11.4.7 The Do-while statement

The **do-while** statement shall be represented by an in-line expression symbol labelled with a **do-while** definition as defined in ES 201 873-1 [1], clause 19.9. 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.




```
do {
    MyPort.send(Template1);
} while(j<10);
```

GFT **Core**

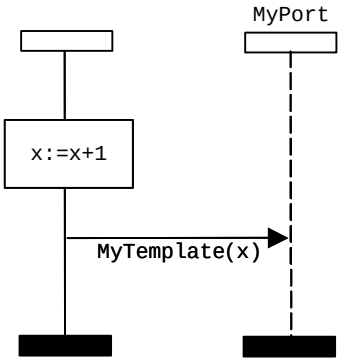
Figure 28: Do-while Statement

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`.

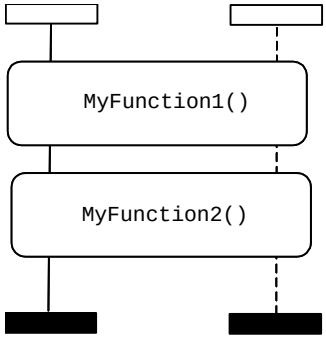



```
x:=x+1;
MyPort.send(MyTemplate(x));
```

GFT **Core**

Figure 29: Sequential behavior

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`.




```
MyFunction1();
MyFunction2();
```

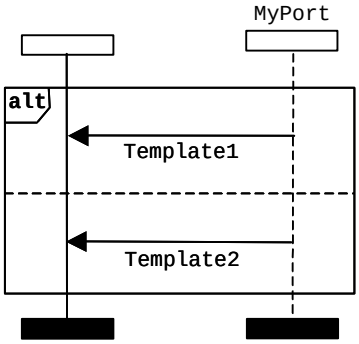
GFT **Core**

Figure 30: Sequencing using references.

11.5.2 Alternative Behaviour

Alternative behaviour shall be represented using in-line 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 in-line 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 in-line expression is shown in Figure 32.

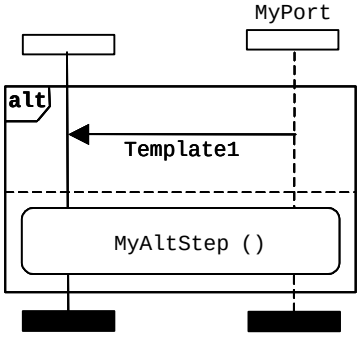



```
alt {
[] MyPort.receive(Template1) {}
[] MyPort.receive(Template2) {}
};
```

GFT **Core**

Figure 31: Alternative behaviour statement

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



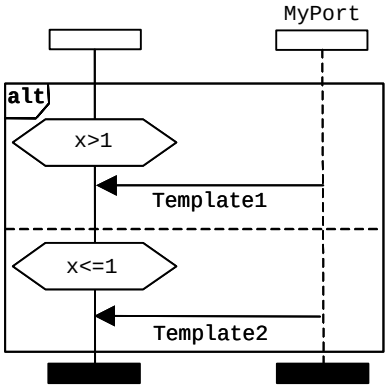

```
alt {
[] MyPort.receive(Template1) {}
[] MyAltStep()
};
```

GFT **Core**

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<=1`.



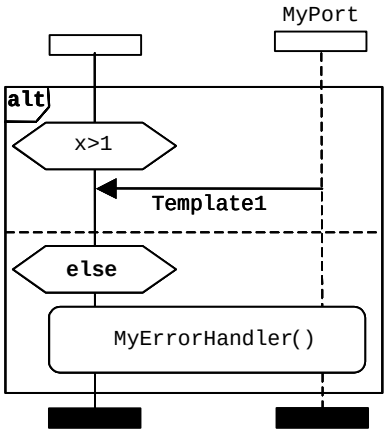

```
alt {
[x>1]  MyPort.receive(Template1) {}
[x<=1] MyPort.receive(Template2) {}
};
```

GFT **Core**

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.




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

GFT **Core**

Figure 34: Else within an alternative

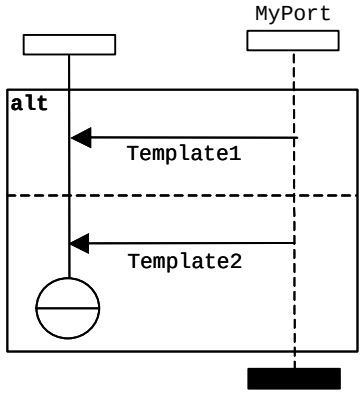
Note that the reference symbol within an else branch should always cover all port instances, if communication operations are involved.

The re-evaluation of an alt statement can be specified using a repeat statement. In GFT, a circle with a line through it denotes a repeat statement (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 matched a message receive event with the value defined by `Template2`, causes the alternative to be repeated.



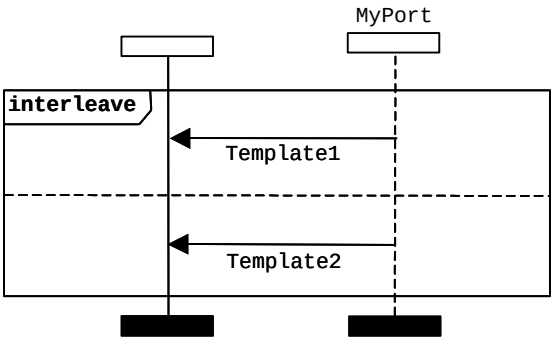

```
alt {
[] MyPort.receive(Template1) {}
[] MyPort.receive(Template2) { repeat; }
};
```

GFT **Core**

Figure 35: Repeat within an alternative

11.5.4 Interleaved Behaviour

Interleave behaviour shall be represented using an in-line 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.




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

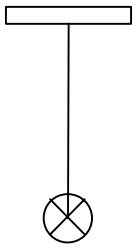
GFT **Core**

Figure 36: Interleave statement.

(1) Note that an interleave in-line expression should always cover all port instances if communication operators are involved.

11.5.5 The Return statement

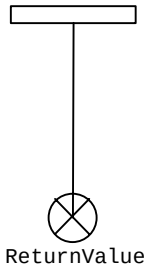
The **return** statement shall be represented by a return symbol. This may be optionally associated with a return value. A return symbol shall only be used in a GFT function diagram. It shall only be used as last event of a component instance or as last event of an operand in an inline expression symbol. 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**.**



```
return;
```

GFT **Core**

Figure 37: Return symbol without a return value.




```
return ReturnValue;
```

GFT **Core**

Figure 38: Return symbol with a return value.

11.6 Default handling

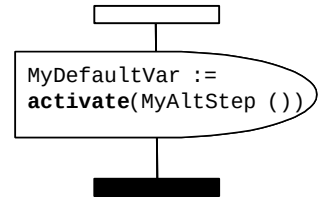
GFT provides graphical representation for the activation and deactivation of defaults – ES 201 873-1 [1], Section 21.

11.6.1 Default references

Variables of **default type** can either be declared within an action symbol or within a default symbol as part of an activate statement. Section 11.3.1 and Section 11.3.3 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).



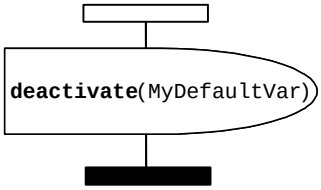

```
MyDefaultVar:=activate(MyAltStep());
```

GFT **Core**

Figure 39: Default activation.

11.4.3 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.




```
deactivate(MyDefaultVar);
```

GFT **Core**

Figure 40: Deactivation of defaults.

11.7 Configuration operations

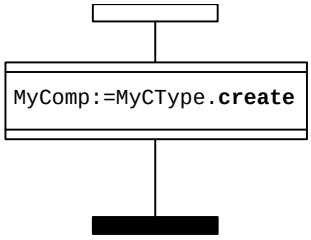
Configuration operations are used to set up and control test components. These operations shall only be used in GFT test case, function, and altstep diagrams.

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.



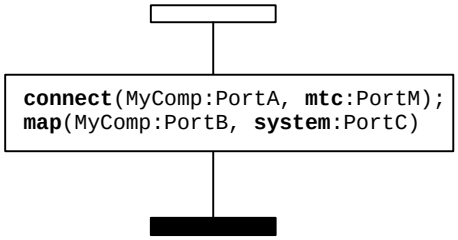

```
MyComp:=MyCType.create;
```

GFT **Core**

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.



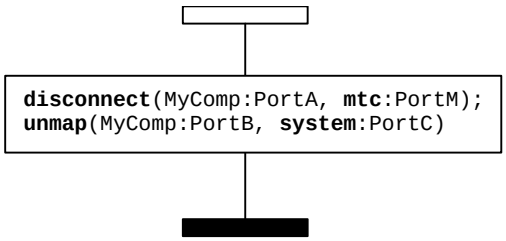

```
connect(MyComp:PortA, mtc:PortM);
map(MyComp:PortB, system:PortC);
```

GFT **Core**

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.



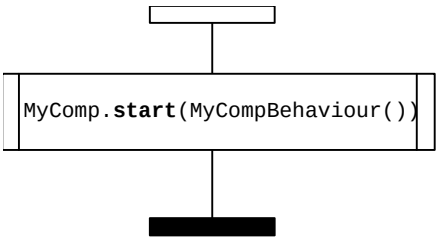

```
disconnect(MyComp:PortA, mtc:PortM);
unmap(MyComp:PortB, system:PortC);
```

GFT **Core**

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.




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

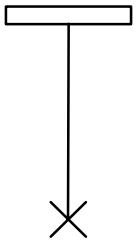
GFT **Core**

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 a *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.



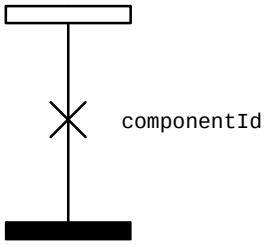
```
stop;
```

GFT **Core**

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, left hand side). A PTC can stop the test execution by stopping the MTC (see Figure 47, right hand side). 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`**).

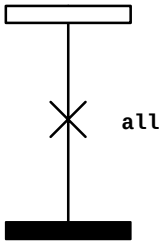
NOTE: The stop symbol has an associated expression. It is not always possible to determine statically, if a stop component operation stops the instance that executes the stop operation or stops the test execution.




```
ComponentId.stop;
```

GFT **Core**

Figure 46: Stop test component operation

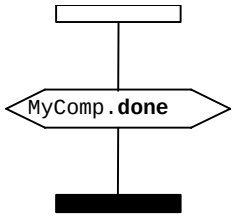


Stopping all PTCs **Stop test case execution**

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.




```
MyComp.done;
```

GFT **Core**

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.

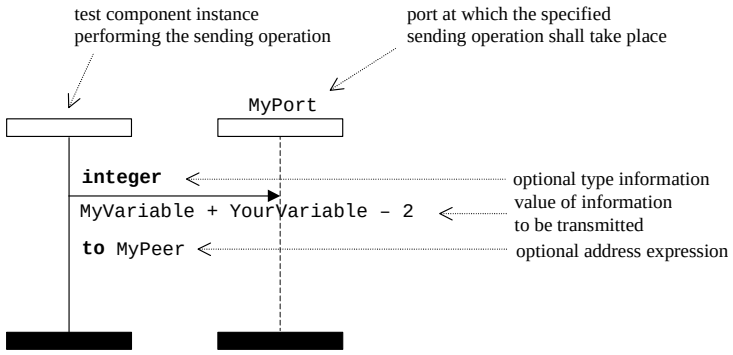


Figure 49: General format of sending operations

The structure of the **call** operation is more specific. Please refer to Section 11.8.2.1 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. it 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).

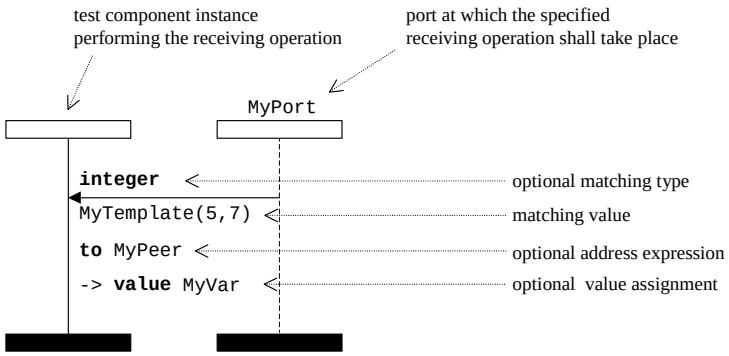


Figure 50: General format of receiving operations with address and value assignment

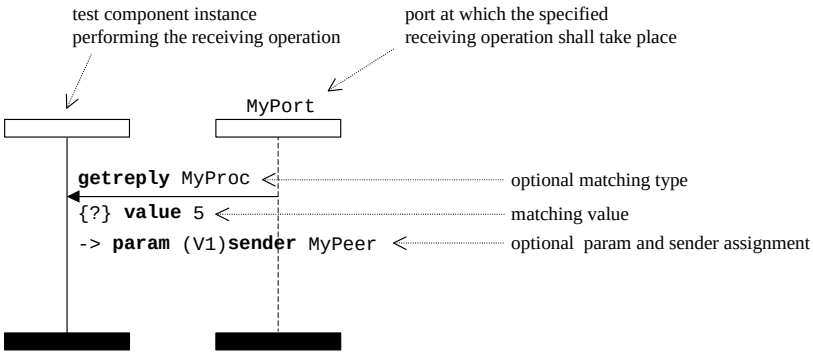
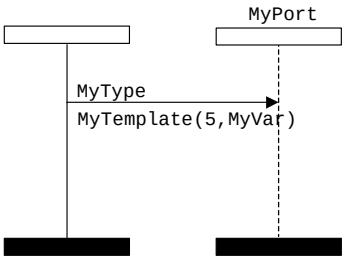


Figure 51: General format of receiving operations with param and sender assignment

11.8.1 Message-based communication

11.8.1.1 The Send operation

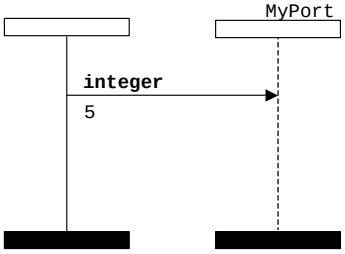
The send operation shall be represented by an outgoing message symbol from the test component to the port instance. The optional type information shall be placed above the message arrow. The (inline) template shall be placed underneath the message arrow (see Figure 52 and Figure 53).




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

GFT **Core**

Figure 52: Send operation with template reference



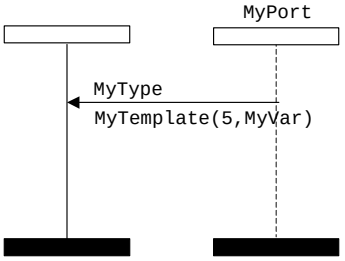

```
MyPort.send(integer:5);
```

GFT **Core**

Figure 53: Send operation with inline template

11.8.1.2 The Receive operation

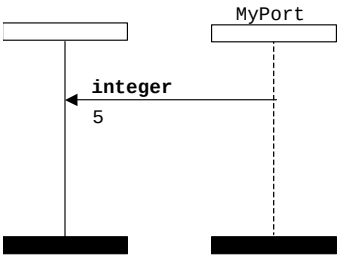
The receive operation shall be represented by an incoming message arrow from the port instance to the test component. The optional type information shall be placed above the message arrow. The (inline) template shall be placed underneath the message arrow (see Figure 54 and Figure 55).




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

GFT **Core**

Figure 54: Receive operation with template reference



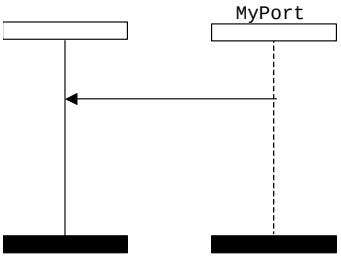

```
MyPort.receive(integer:5);
```

GFT **Core**

Figure 55: Receive operation with inline template

11.8.1.2.1 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).



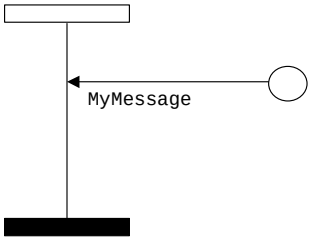

```
MyPort.receive;
```

GFT **Core**

Figure 56: Receive any message

11.8.1.2.2 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).



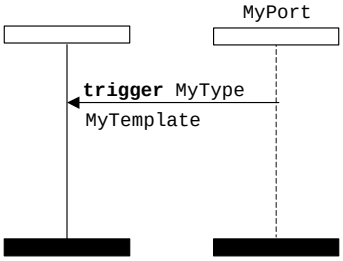

```
any port.receive(MyMessage);
```

GFT **Core**

Figure 57: Receive on any port

11.8.1.3 The Trigger operation

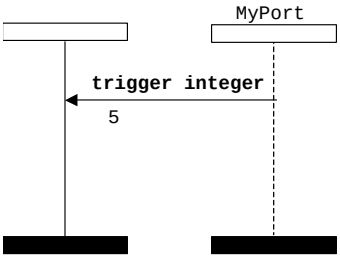
The trigger operation shall be represented by an incoming message arrow from the port instance to the test component and the keyword **trigger** above the message arrow preceding the type information if present. The optional type information is placed above the message arrow subsequent to the keyword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




```
MyPort.trigger(MyType: MyTemplate);
```

GFT **Core**

Figure 58: Trigger operation with template reference



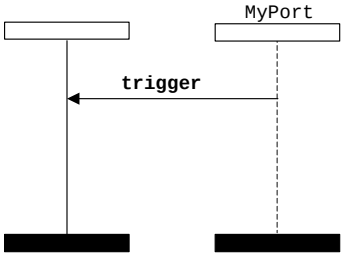

```
MyPort.trigger(integer:5);
```

GFT **Core**

Figure 59: Trigger operation with inline template

11.8.1.3.1 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).



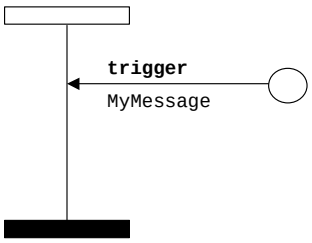

```
MyPort.trigger;
```

GFT **Core**

Figure 60: Trigger on any message operation

11.8.1.3.2 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).




```
any port.trigger(MyMessage);
```

GFT **Core**

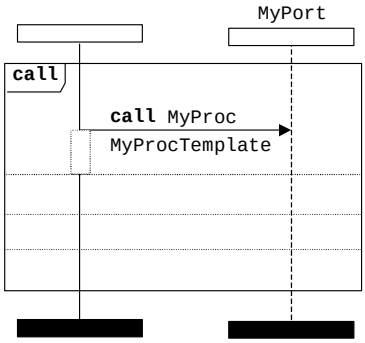
Figure 61: Trigger on any port operation

11.8.2 Procedure-based communication

11.8.2.1 The Call operation

11.8.2.1.1 Calling blocking procedures

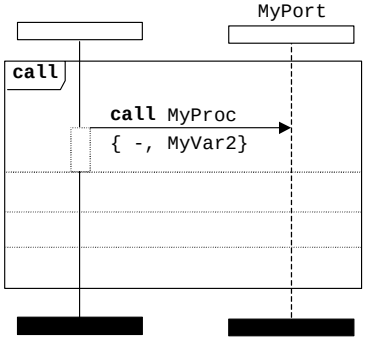
The blocking **call** operation is represented by an outgoing message symbol from the test component to the port instance with a subsequent suspension region on the test component and the keyword **call** above the message arrow preceding the signature if present. The (inline) template is placed underneath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




```
MyPort.call(MyProc: MyProcTemplate) {
    [] …
    [] …
    [] …
}
```

GFT **Core**

Figure 62: Blocking call operation with template reference

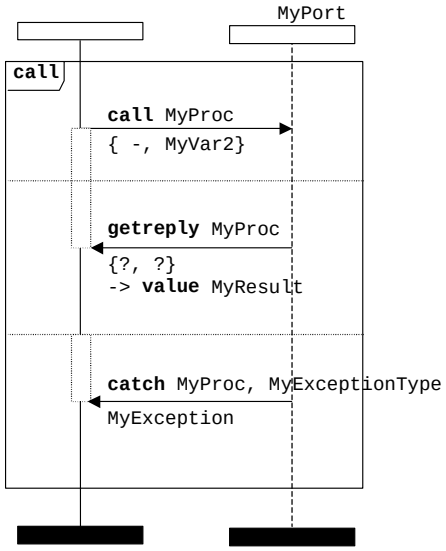



```
MyPort.call(MyProc:{ -, MyVar2}) {
    [] …
    [] …
    [] …
}
```

GFT **Core**

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).

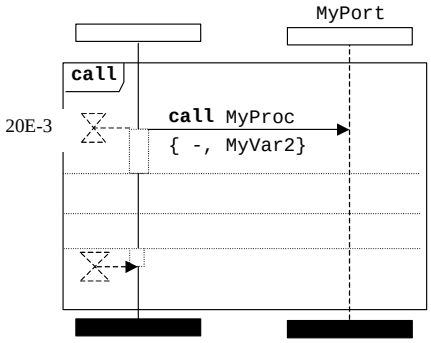



```
MyPort.call(MyProc:{ -, MyVar2}) {
    [] MyPort.getreply(MyProc:{?, ?})
       -> value MyResult { }
    [] MyPort.catch
       (MyProc, MyExceptionType: MyException) { }
}
```

GFT **Core**

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).



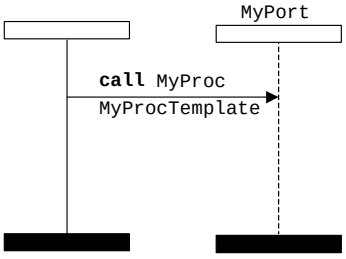

```
MyPort.call(MyProc:{ -, MyVar2},20E-3) {
    [] …
    [] …
    [] MyPort.catch(timeout) {
       …
       }
}
```

Core

Figure 65: Blocking call operation followed by timeout exception

11.8.2.1.2 Calling non-blocking procedures

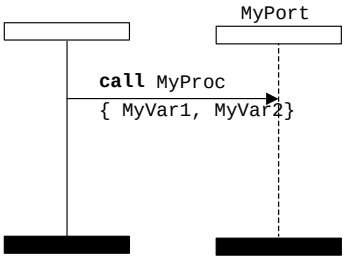
The non-blocking call operation shall be represented by an outgoing message symbol from the test component to the port and the keyword **call** above the message arrow preceding the signature. There shall be no suspension region symbol attached to the message symbol. The optional signature is represented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




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

GFT **Core**

Figure 66: Non-blocking call operation with template reference



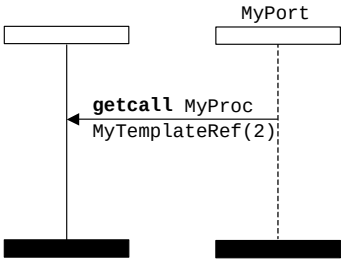

```
MyPort.call(MyProc: {MyVar1,MyVar2}, nowait);
```

GFT **Core**

Figure 67: Non-blocking call operation with inline template

11.8.2.2 The Getcall operation

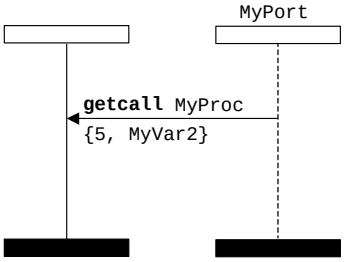
The getcall operation shall be represented by an incoming message arrow from the port instance to the test component and the keyword **getcall** above the message arrow preceding the signature. The signature is placed above the message arrow subsequent to the keyword **getcall**. The (inline) template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




```
MyPort.getcall(MyProc: MyTemplateRef(2));
```

GFT **Core**

Figure 68: Getcall operation with template reference



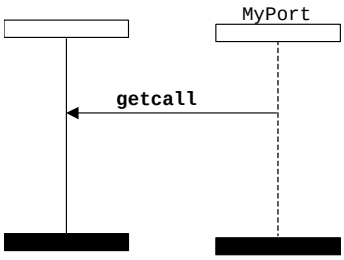

```
MyPort.getcall(MyProc: { 5, MyVar2});
```

GFT **Core**

Figure 69: Getcall operation with inline template

11.8.2.2.1 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).



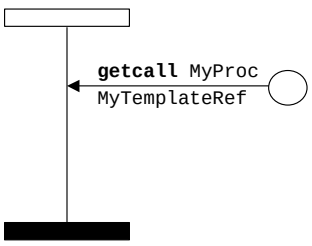

```
MyPort.getcall;
```

GFT **Core**

Figure 70: Getcall on any call operation

11.8.2.2.2 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).



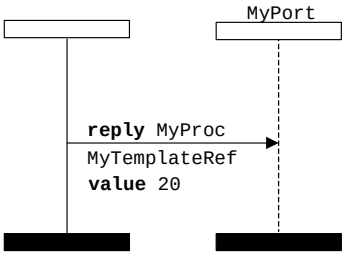

```
any port.getcall(MyProc: MyTemplateRef);
```

GFT **Core**

Figure 71: Getcall on any port operation with template reference

11.8.2.3 The Reply operation

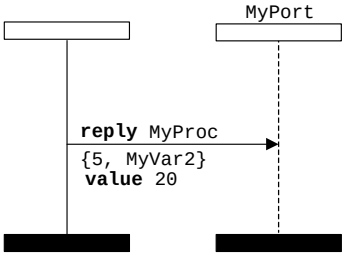
The reply operation shall be represented by an outgoing message symbol from the test component to the port instance and the keyword **reply** above the message arrow preceding the signature. The signature shall be placed above the message arrow subsequent to the keyword **reply**. The (inline) template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




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

GFT **Core**

Figure 72: Reply operation with template reference




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

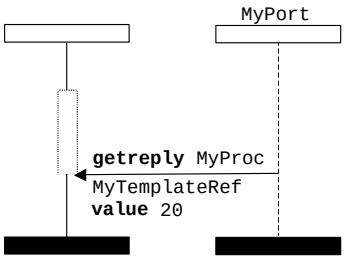
GFT **Core**

Figure 73: Reply operation with inline template

11.8.2.4 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 Figure 74 and Figure 75). Outside a call symbol, the message arrow head shall not be attached to a preceding suspension region on the test component (see Figure 76 and Figure 77).

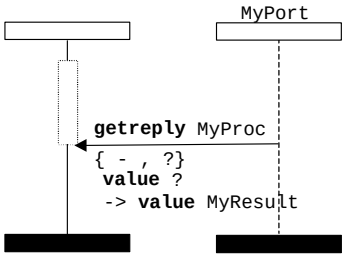
The signature shall be placed above the message arrow subsequent to the keyword **getreply**. The (inline) template shall be placed underneath the message arrow.




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

GFT **Core**

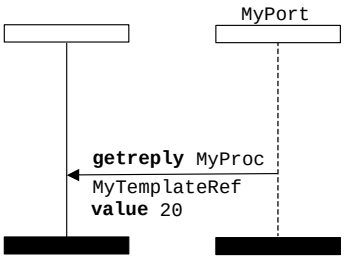
Figure 74: Getreply operation with template reference (within a call symbol)




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

GFT **Core**

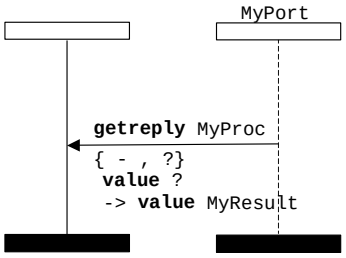
Figure 75: Getreply operation with inline template (within a call symbol)




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

GFT **Core**

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



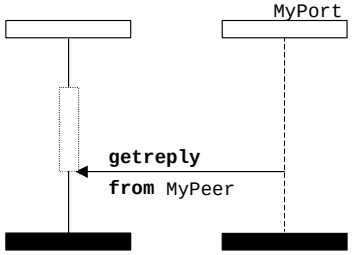

```
MyPort.getreply(MyProc: {-, ?))
value ? -> value MyResult;
```

GFT **Core**

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

11.8.2.4.1 Get any reply from any call

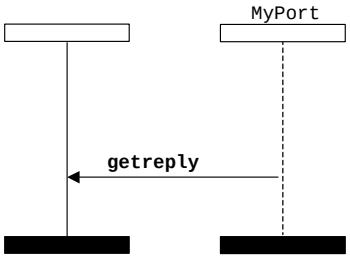
The get any reply from any call operation shall be represented by an incoming message arrow from the port instance to the test component and the keyword **getreply** above the message. No signature shall follow the **getreply** keyword. Within a call symbol, the message arrow head shall be attached to 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).




```
[] MyPort.getreply from Mypeer { … }
```

GFT **Core**

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




```
MyPort.getreply;
```

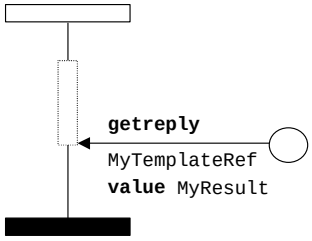
GFT **Core**

Figure 79: Getreply from any call (outside a call symbol)

11.8.2.4.2 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).

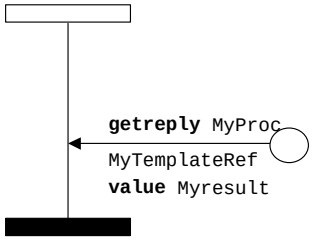
The signature if present shall be placed above the message arrow subsequent to the keyword **getreply**. The optional (inline) template is placed underneath the message arrow.




```
[] any port.getreply(MyProc: MyTemplateRef
   value MyResult) { … }
```

GFT **Core**

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



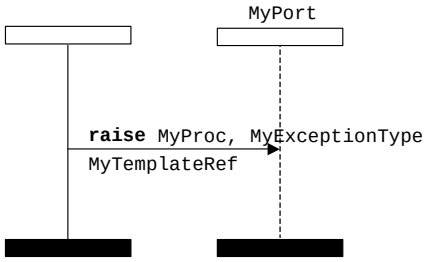

```
any port.getreply(MyProc: MyTemplateRef
value MyResult);
```

GFT **Core**

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

11.8.2.5 The Raise operation

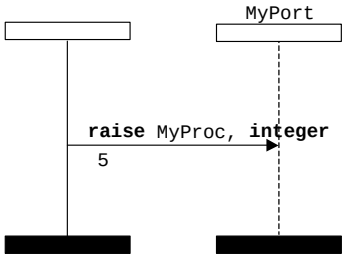
The raise operation shall be represented by an outgoing message symbol from the test component to the port instance. The keyword **raise** shall be placed above the message arrow preceding the signature and the exception type, which are separated by a comma. The (inline) template shall be placed underneath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




```
MyPort.raise(MyProc,
MyExceptionType: MyTemplateRef);
```

GFT **Core**

Figure 82: Raise operation with template reference




```
MyPort.raise(MyProc, integer:5});
```

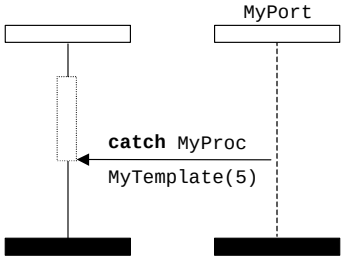
GFT **Core**

Figure 83: Raise operation with inline template

11.8.2.6 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 Figure 84 and Figure 85). Outside a call symbol, the message arrow head shall not be attached to a preceding suspension region on the test component (see Figure 86 and Figure 87).

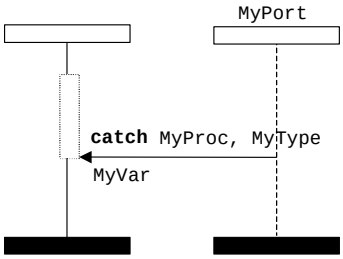
The signature and optional exception type information are placed above the message arrow subsequent to the keyword **catch** and are separated by a comma if the exception type is present. The (inline) template is placed underneath the message arrow.




```
[] MyPort.catch(MyProc, MyTemplate(5)) { … }
```

GFT **Core**

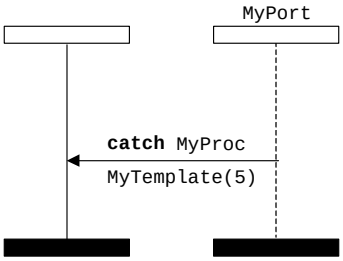
Figure 84: Catch operation with template reference (within a call symbol)




```
[] MyPort.catch(MyProc, MyType: MyVar) { … }
```

GFT **Core**

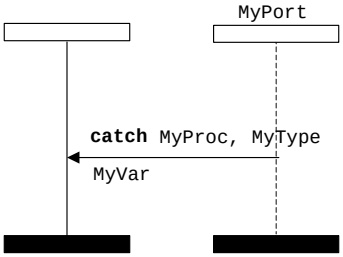
Figure 85: Catch operation with inline template (within a call symbol)




```
MyPort.catch(MyProc, MyTemplate(5));
```

GFT **Core**

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



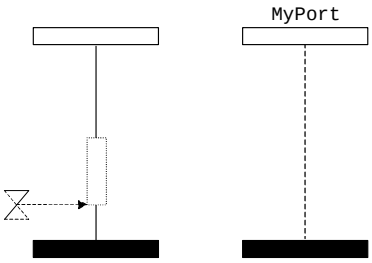

```
MyPort.catch(MyProc, MyType: MyVar);
```

GFT **Core**

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

11.8.2.6.1 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.




```
[] MyPort.catch(timeout) { … }
```

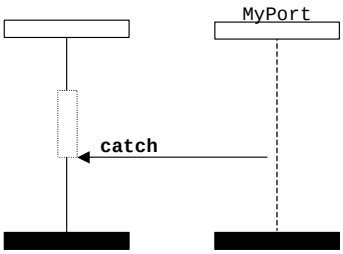
GFT **Core**

Figure 88: Timeout exception (within a call symbol)

11.8.2.6.2 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

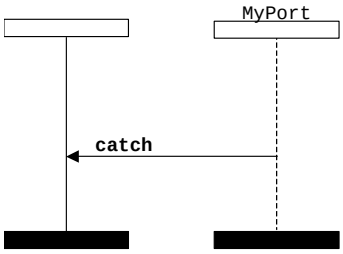
attached to a preceding suspension region on the test component (see Figure 89). Outside a call symbol, the message arrow head shall not be attached to a preceding suspension region on the test component (see Figure 90). The catch any exception shall have no template and no exception type.




```
[] MyPort.catch { … }
```

GFT **Core**

Figure 89: Catch any exception (within a call symbol)



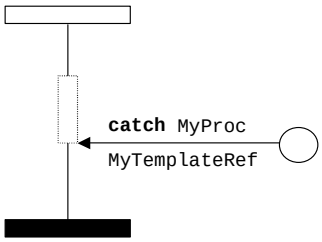

```
MyPort.catch;
```

GFT **Core**

Figure 90: Catch any exception (outside a call symbol)

11.8.2.6.3 Catch on any port

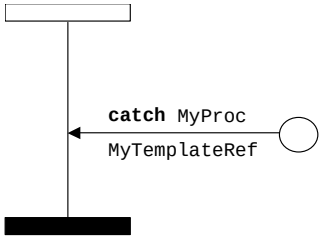
The catch on any port operation is represented by a found symbol representing any port to the test component and the keyword **catch** above the message arrow. Within a call symbol, the message arrow head shall be attached to a preceding suspension region on the test component (see Figure 91). Outside 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.




```
[] any port.catch(MyProc: MyTemplateRef) { … }
```

GFT **Core**

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



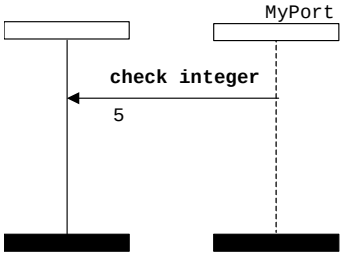

```
any port.catch(MyProc: MyTemplateRef);
```

GFT **Core**

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

11.8.3 The Check operation

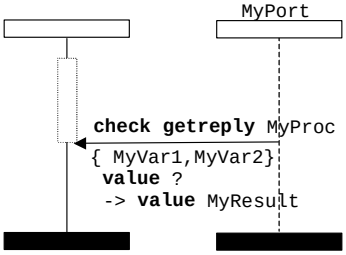
The check operation shall be represented by an incoming message arrow from the port instance to the test component. The keyword **check** shall be placed above the message arrow. The attachment of the information related to the **receive** (see Figure 93), **getcall**, **getreply** (see Figure 94 and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




```
MyPort.check(receive(integer: 5));
```

GFT **Core**

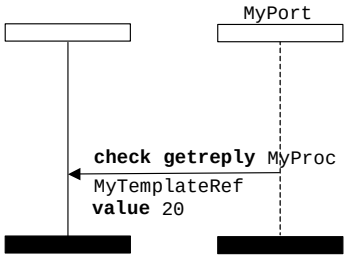
Figure 93: Check a receive with inline template




```
[] MyPort.check(getreply(MyProc1:{MyVar1, MyVar2}
          value ?) -> value MyResult)
   { … }
```

GFT **Core**

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



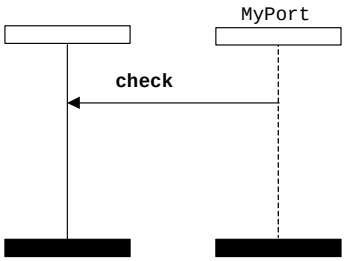

```
MyPort.check(getreply(MyProc1:{MyVar1, MyVar2}
value ?) -> value MyResult);
```

GFT **Core**

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

11.8.3.1 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.



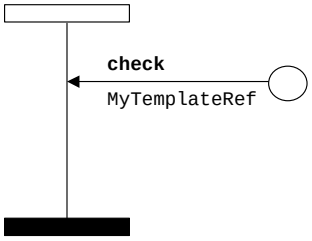

```
MyPort.check;
```

GFT **Core**

Figure 96: Check any operation

11.8.3.2 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.




```
any port.check(receive(MyTemplateRef));
```

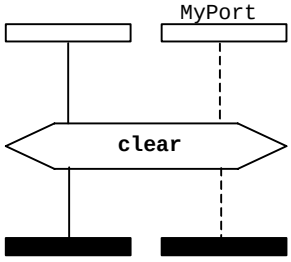
GFT **Core**

Figure 97: Check a receive on any port

11.8.3 Controlling communication ports

11.8.3.1 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).



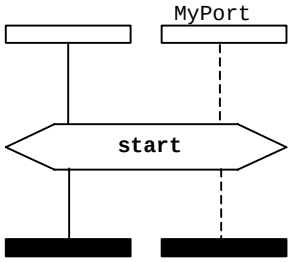

```
MyPort.clear;
```

GFT **Core**

Figure 98: Clear port operation

11.8.3.2 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).



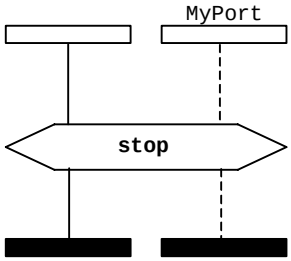

```
MyPort.start;
```

GFT **Core**

Figure 99: Start port operation

11.8.3.3 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).




```
MyPort.stop;
```

GFT **Core**

Figure 100: Stop port operation

11.8.3.4 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 the respective subsections of Section 11.8.

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

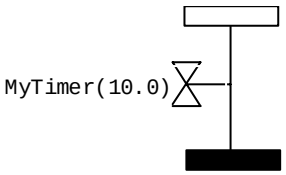
In GFT, there are two different timer symbols: one for identified timers and one for call timers (see Figure 101). They differ in appearance as dashed timer symbols are used for call timers. The difference in syntax is that the identified timer shall have its name attached to its symbol, whereas the call timer does not have a name. Identified timers are described in this section. The call timer will be dealt within Section 11.8.2.

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 their 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).



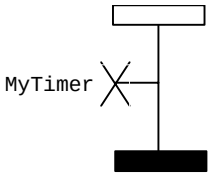

```
MyTimer.start(10.0);
```

GFT **Core**

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).

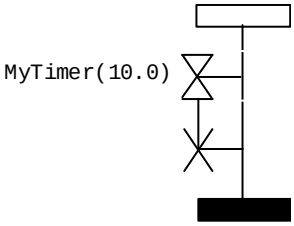



```
MyTimer.stop;
```

GFT **Core**

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 (Figure 104).



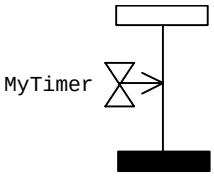

```
MyTimer.start(10.0);
MyTimer.stop;
```

GFT **Core**

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).

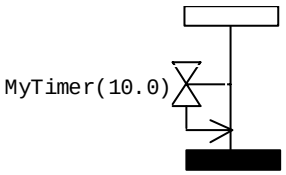



```
MyTimer.timeout;
```

GFT **Core**

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).



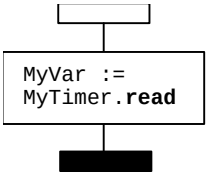

```
MyTimer.start(10.0);
MyTimer.timeout;
```

GFT **Core**

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).



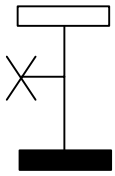

```
MyVar := MyTimer.read;
```

GFT **Core**

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 (Figure 108).

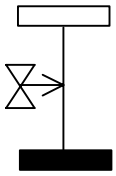


```
all timer.stop;
```

GFT **Core**

Figure 108: Stopping all timers

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



```
any timer.timeout;
```

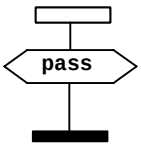
GFT **Core**

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.




```
setverdict(pass);
```

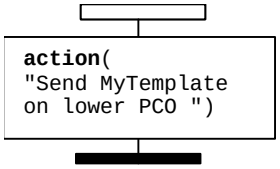
GFT **Core**

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 their 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.



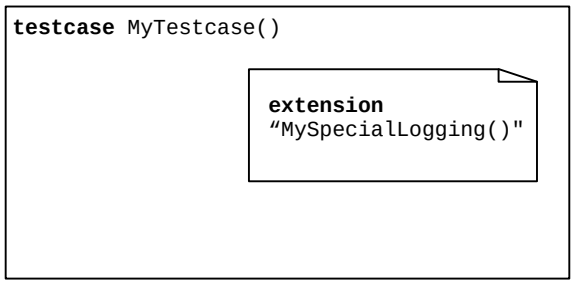

```
action("Send MyTemplate on lower PCO ");
```

GFT **Core**

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.




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

GFT **Core**

Figure 112: Specifying attributes

Annex A (normative): GFT BNF

A.1 Meta-Language for GFT

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

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

See [3] for more details.

A.2 Conventions for the syntax description

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

Table 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	**abc**	**abc**	**abc** or <name> or <character string>	X

A.3 The GFT Grammar

A.3.1 Diagrams

A.3.1.1 Control Diagram

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

```
ControlHeading ::=
        ControlKeyword TTCN3ModuleId
        { LocalDefinition [ SemiColon ] }

ControlBodyArea ::=
        { ControlInstanceArea TextLayer ControlEventLayer } set

TextLayer ::=
        { TextArea } set

ControlEventLayer::=
        ControlEventArea | ControlEventArea above ControlEventLayer

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

A.3.1.2 Testcase Diagram

```
TestcaseDiagram ::=
        Frame contains ( TestcaseHeading TestcaseBodyArea )

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

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

InstanceLayer ::=
        { InstanceArea } set


InstanceEventLayer ::=
        InstanceEventArea | InstanceEventArea above InstanceEventLayer

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

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

```
InstanceCallEventArea ::=
        InstanceBlockingCallEventArea
```

```
    |   InstanceNonBlockingCallEventArea


PortEventLayer ::=
        PortEventArea | PortEventArea ***above*** PortEventLayer

PortEventArea ::=
        PortOutEventArea
    |   PortOtherEventArea

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

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

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

A.3.1.3 Function Diagram

```
FunctionDiagram ::=
        Frame ***contains*** ( FunctionHeading FunctionBodyArea )

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

FunctionBodyArea ::=
        TestcaseBodyArea
```

A.3.1.4 Altstep Diagram

```
AltstepDiagram ::=
        Frame ***contains*** (AltstepHeading AltstepBodyArea )

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

AltstepBodyArea ::=
        TestcaseBodyArea
```

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

A.3.1.5 Comments

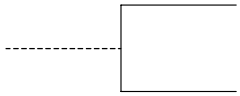
```
TextArea ::=
      TextSymbol
      contains ( { TTCN3Comments } [ MultiWithAttrib ] { TTCN3Comments } )
```

Note that there is no explicit rule for TTCN3 comments, they are explained in ES 201 873-1 [1], clause A.1.4.

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

```
TextSymbol ::=
```

```
CommentArea ::=
      EventCommentSymbol contains TTCN3Comments

EventCommentSymbol ::=
```



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

A.3.1.6 Diagram

```
Frame ::=
```

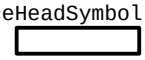
```
LocalDefinition ::=
            ConstDef
        |   VarInstance
        |   TimerInstance
```

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

A.3.2 Instances

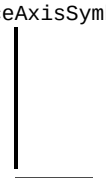
A.3.2.1 Component Instances

```
InstanceArea ::=
        ComponentInstanceArea
        | PortInstanceArea

ComponentInstanceArea ::=
        ComponentHeadArea is followed by ComponentBodyArea

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

InstanceHeadSymbol ::=
```



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

InstanceAxisSymbol::=



ComponentEndArea ::=
InstanceEndSymbol
| StopArea
| ReturnArea
| RepeatSymbol
| GotoArea

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

A.3.2.2 Port Instances

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

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

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

PortAxisSymbol::=

A.3.2.3 Control Instances

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

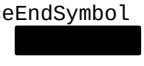
ControlInstanceHeadArea ::=
ControlKeyword
is followed by InstanceHeadSymbol

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

ControlInstanceEndArea ::=
InstanceEndSymbol

A.3.2.4 Instance End

InstanceEndSymbol ::=

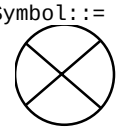


StopArea::=
StopSymbol
is associated with (Expression)

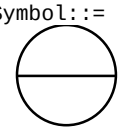
/* STATIC SEMANTICS – the expression shall refer to either the mtc or to self */

StopSymbol::=

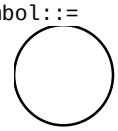
```
ReturnArea::=
        ReturnSymbol
        [ is associated with Expression ]

ReturnSymbol::=
```



```
RepeatSymbol::=
```

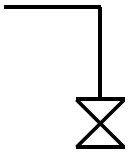


```
GotoArea::=
        GotoSymbol
        contains LabelIdentifier

GotoSymbol::=
```

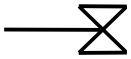


A.3.3 Timer

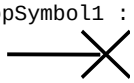
```
InstanceTimerEventArea::=
        InstanceTimerStartArea
        | InstanceTimerStopArea
        | InstanceTimeoutArea


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


TimerStartSymbol ::=
        TimerStartSymbol1 | TimerStartSymbol2

TimerStartSymbol1 ::=
```



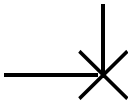
```
TimerStartSymbol2 ::=
```



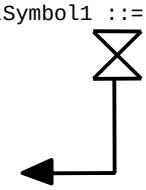
```
InstanceTimerStopArea ::=
        TimerStopArea1 | TimerStopArea2

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

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

TimerStopSymbol1 ::=
```



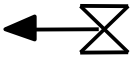
```
TimerStopSymbol2 ::=
```



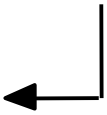
```
InstanceTimeoutArea ::=
        TimeoutArea1 | TimeoutArea2

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

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

TimeoutSymbol ::=
        TimeoutSymbol1 | TimeoutSymbol2

TimeoutSymbol1 ::=
```



```
TimeoutSymbol2 ::=
```



```
TimeoutSymbol3 ::=
```



A.3.4 Action

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

ActionSymbol ::=
```

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

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

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

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

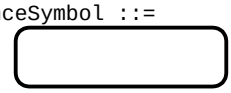
/* STATIC SEMANTICS – declarations of constants and variables with create, activate, and execute statements as well as with function invocations of user-defined functions must not be made textually within an action box, but must be made graphically within create, default, execute, and reference symbols,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

A.3.5 Invocation

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

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

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

ReferenceSymbol ::=
```



A.3.5.1 Function and Altstep Invocation on Component/Control Instances

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

InstanceInvocationBeginSymbol ::=
        VoidSymbol

InstanceInvocationEndSymbol ::=
        VoidSymbol
```

A.3.5.2 Function and Altstep Invocation on Ports

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

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

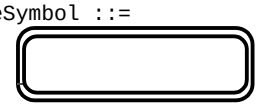
```
PortInvocationBeginSymbol ::=
        VoidSymbol

PortInvocationEndSymbol ::=
        VoidSymbol
```

A.3.5.3 Testcase Execution

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

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

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

```
ExecuteSymbol ::=
```

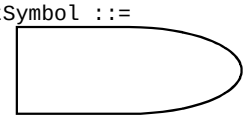


A.3.6 Activation/Deactivation of Defaults

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

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

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

```
DefaultSymbol ::=
```



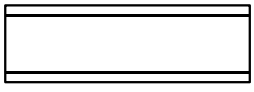
A.3.7 Test Components

A.3.7.1 Creation of Test Components

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

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

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

```
CreateSymbol ::=
```



A.3.7.2 Starting Test Components

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

StartSymbol ::=
```

A.3.7.2 Stopping Test Components

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

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

A.3.8 Inline Expressions

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

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

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

```
InstanceInlineExpressionBeginSymbol ::=
        VoidSymbol

InstanceInlineExpressionSeparatorSymbol::=
        VoidSymbol

InstanceInlineExpressionEndSymbol ::=
        VoidSymbol

VoidSymbol ::=      .

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

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

```
ForArea ::=
        ForInlineExpressionArea
        is attached to InstanceInlineExpressionBeginSymbol
```

```
        is attached to InstanceInlineExpressionEndSymbol
        [ is attached to { PortInlineExpressionBeginSymbol } set
          is attached to { PortInlineExpressionEndSymbol } set ]

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

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

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

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

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

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

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

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

GuardedOperandArea ::=
        GuardOpLayer is followed by
        ConnectorLayer
```

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

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

ElseOperandArea ::=
        ElseConditionArea
        is followed by ConnectorLayer
```

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

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

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

UnguardedOperandArea ::=
        UnguardedOpLayer is followed by
        ConnectorLayer
```

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

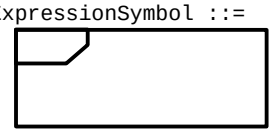
```
UnguardedOpLayer ::=
        ReceiveArea
        | TriggerArea
        | GetcallArea
        | CatchOutsideCallArea
        | CheckArea
        | GetreplyOutsideCallArea

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

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

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

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

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

```
GuardedConditionLayer ::=
        BooleanExpressionConditionArea
        | DoneArea

CallBodyOpsLayer ::=
        GetreplyWithinCallArea
        | CatchWithinCallArea

InlineExpressionSymbol ::=
```



```
SeparatorSymbol::=
        ------------
```

A.3.8.1 Inline Expressions on Component Instances

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

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

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

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

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

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

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

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

```
InstanceGuardOpArea ::=
        (   InstanceTimeoutArea
        |   InstanceReceiveEventArea
        |   InstanceTriggerEventArea
        |   InstanceGetcallEventArea
        |   InstanceGetreplyOutsideCallEventArea
        |   InstanceCatchOutsideCallEventArea
        |   InstanceCheckEventArea
        |   InstanceDoneArea )
        is attached to InstanceAxisSymbol
```

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


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

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

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

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

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

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

A.3.8.2 Inline Expressions on Ports

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

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

PortInlineExpressionBeginSymbol ::=
        VoidSymbol
```

```
PortInlineExpressionSeparatorSymbol::=
        VoidSymbol

PortInlineExpressionEndSymbol::=
        VoidSymbol

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

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

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

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

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

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

A.3.8.3 Inline Expressions on Control Instances

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

```
ControlIfArea ::=
        ( InstanceInlineExpressionBeginSymbol
          [ is followed by ControlEventArea ]
          [ is followed by InstanceInlineExpressionSeparatorSymbol
            is followed by ControlEventArea ]
          is followed by InstanceInlineExpressionEndSymbol )
        is attached to InstanceAxisSymbol
        is attached to IfInlineExpressionArea

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

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

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

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

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

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

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


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

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

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

A.3.9 Condition

```
ConditionArea ::=
        PortOperationArea
```

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

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

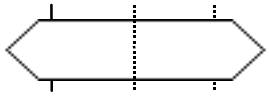
```
InstanceConditionBeginSymbol ::=
        VoidSymbol

InstanceConditionEndSymbol ::=
        VoidSymbol

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

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

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

/* STATIC SEMANTICS – SingleExpression must resolve to a value of type verdict */
/* STATIC SEMANTICS – the SetLocalVerdict shall not be used to assign the value error */
/* STATIC SEMANTICS – if the keywords pass, fail, inconc, and fail are used, the form with the setverdict keyword shall not be used */

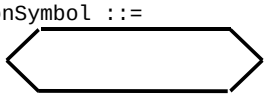
```
PortOperationArea ::=
        ConditionSymbol
        contains PortOperationText
        is attached to InstanceConditionBeginSymbol
        is attached to InstanceConditionEndSymbol
        is attached to { PortInlineExpressionBeginSymbol }+ set
        is attached to { PortInlineExpressionEndSymbol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

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

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



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

ElseConditionArea ::=
        ConditionSymbol
        contains ElseKeyword
        is attached to InstanceAxisSymbol

ConditionSymbol ::=
```



A.3.9.1 Condition on Component Instances

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

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

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

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

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

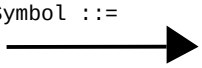
A.3.9.2 Condition on Ports

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

PortConditionBeginSymbol ::=
        VoidSymbol

PortConditionEndSymbol ::=
        VoidSymbol
```

A.3.10 Message-based Communication

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

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

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

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

```
MessageSymbol ::=
```



A.3.10.1 Message-based Communication on Component Instances

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

MessageOutSymbol ::=
        VoidSymbol
```

The VoidSymbol is a geometric point without spatial extension.

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

MessageInSymbol ::=
        VoidSymbol
```

A.3.10.2 Message-based Communication on Port Instances

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

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

A.3.11 Signature-based Communication

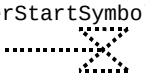
```
NonBlockingCallArea ::=
        MessageSymbol
        is associated with CallKeyword [ Signature ]
        is associated with ( [ DerivedDef AssignmentChar ] TemplateBody
                             [ ToClause ] )
        is attached to InstanceCallEventArea
        is attached to PortCallInEventArea

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

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

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

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

/* STATIC SEMANTICS – a signature, if existent, shall be put on top of the message symbol */
```

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

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

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

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

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

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

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

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

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

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

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

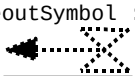
/* STATIC SEMANTICS – a template, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – a from clause, if existent, shall be put underneath the message symbol */
/* STATIC SEMANTICS – a port redirect, if existent, shall be put underneath the message symbol */

A.3.11.1 Signature-based Communication on Component Instances

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

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

CallTimerStartSymbol ::=
```



```
InstanceNonBlockingCallEventArea ::=
        InstanceSendEventArea

InstanceGetcallEventArea ::=
        InstanceReceiveEventArea

InstanceReplyEventArea ::=
        InstanceSendEventArea

InstanceGetreplyWithinCallEventArea ::=
        InstanceReceiveEventArea
        is attached to SuspensionRegionSymbol

InstanceGetreplyOutsideCallEventArea ::=
        InstanceReceiveEventArea

InstanceRaiseEventArea ::=
        InstanceSendEventArea

InstanceCatchWithinCallEventArea ::=
        InstanceReceiveEventArea
        is attached to SuspensionRegionSymbol

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

CallTimeoutSymbol ::=
```



```
InstanceCatchOutsideCallEventArea ::=
        InstanceReceiveEventArea
```

A.3.11.2 Signature-based Communication on Ports

```
PortGetcallOutEventArea::=
        PortOutMsgEventArea

PortGetreplyOutEventArea::=
        PortOutMsgEventArea

PortCatchOutEventArea::=
        PortOutMsgEventArea

PortCallInEventArea::=
        PortInMsgEventArea

PortReplyInEventArea::=
        PortInMsgEventArea
```

```
PortRaiseInEventArea::=
        PortInMsgEventArea
```

A.3.12. Trigger and Check

A.3.12.1 Trigger and Check on Component Instances

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

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

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

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

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

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

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

InstanceTriggerEventArea ::=
        InstanceReceiveEventArea

InstanceCheckEventArea ::=
        InstanceReceiveEventArea
```

A.3.12.2 Trigger and Check on Port Instances

```
PortTriggerOutEventArea ::=
        PortOutMsgEventArea

PortCheckOutEventArea ::=
        PortOutMsgEventArea
```

A.3.13 Handling of Communication from Any Port

```
InstanceFoundEventArea ::=
        FoundSymbol
        contains FoundEvent
        is attached to InstanceAxisSymbol

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

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

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

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

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

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


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

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

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

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

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

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

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

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

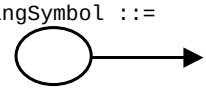
```
FoundSymbol ::=
```

A.3.14 Labelling

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

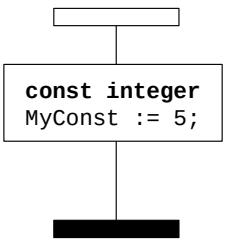
/* STATIC SEMANTICS – the label identifier shall be placed inside the circle of the labelling symbol */

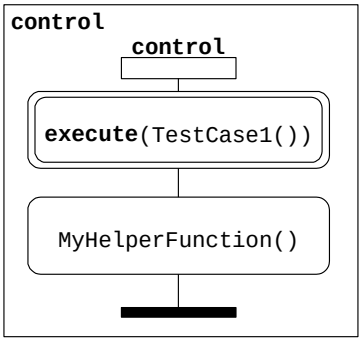
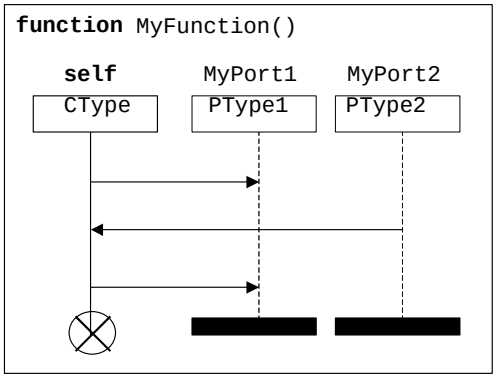
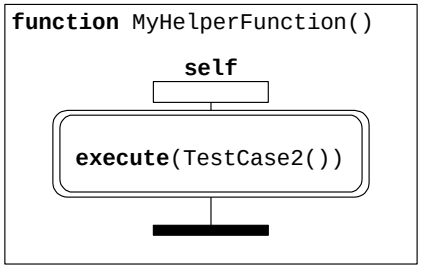
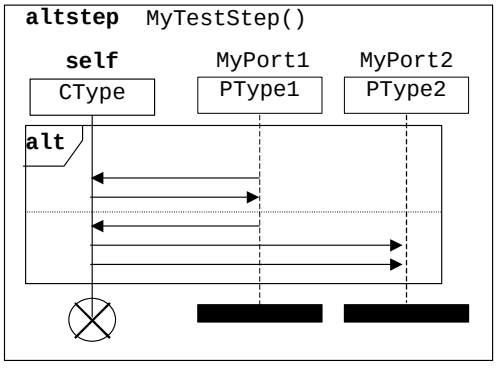
```
LabellingSymbol ::=
```

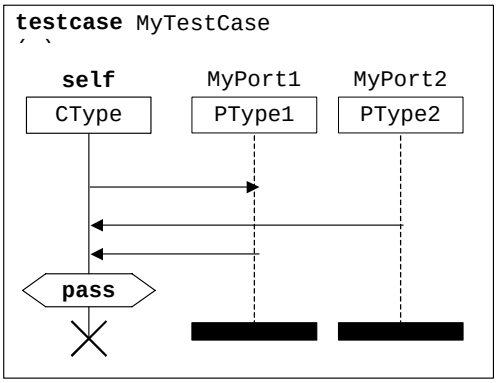
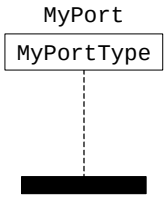
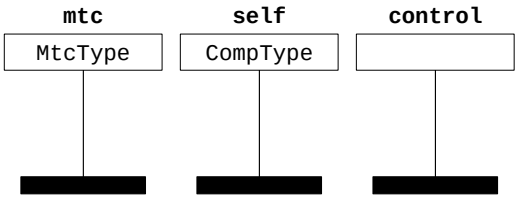
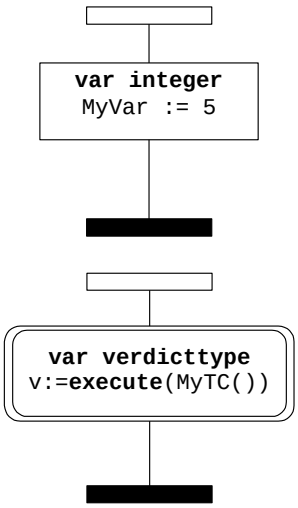


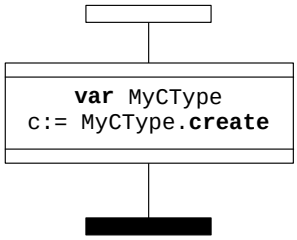
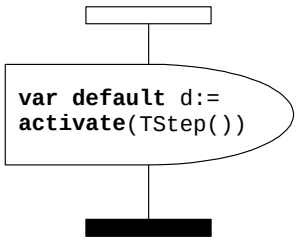
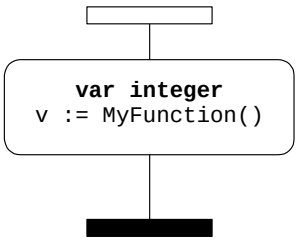
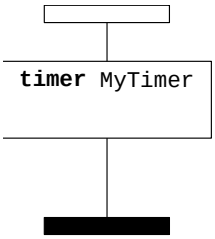
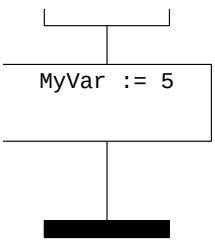
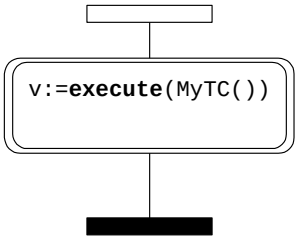
Annex B (normative): Reference Guide for GFT

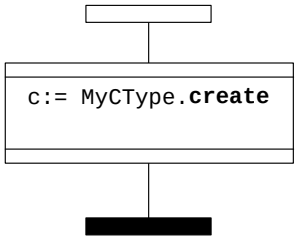
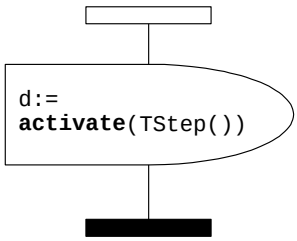
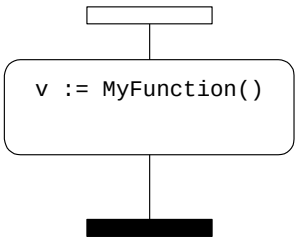
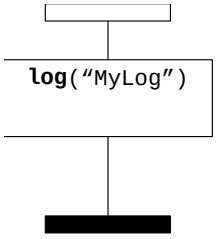
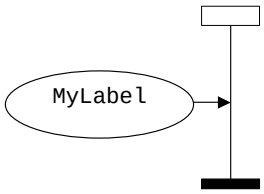
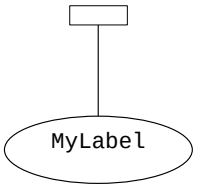
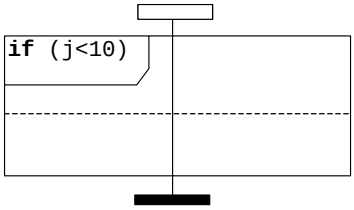
This annex lists the main TTCN-3 language elements and their representation in GFT. For a complete description of the GFT symbols and their use please refer to the main text.

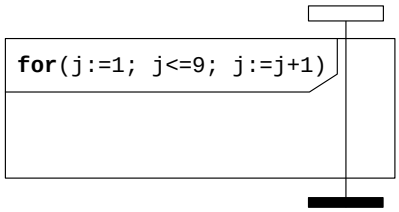
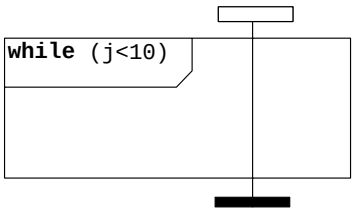
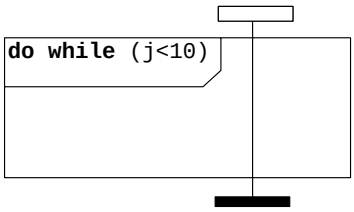
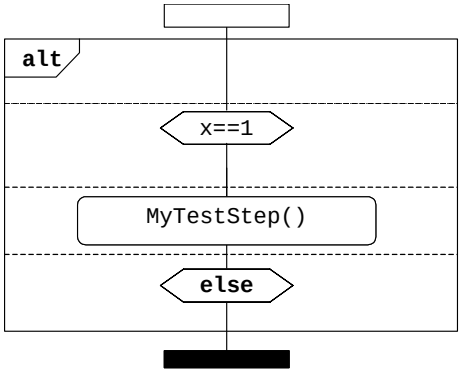
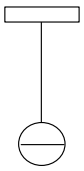
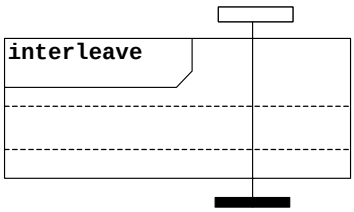
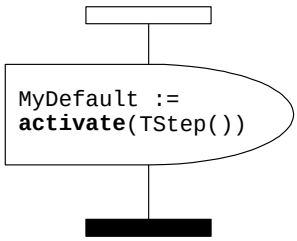
Language Element	Associated Keyword	GFT symbols, if existent, and typical usage	Note
Module Definitions			
TTCN-3 module definition	**module**		No special GFT symbol, i.e., the core notation or another presentation format may be used.
Import of definitions from other module	**import**		No special GFT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

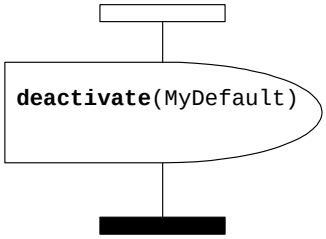
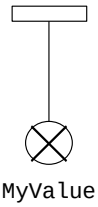
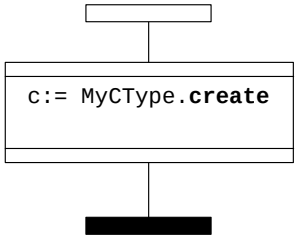
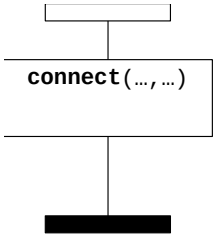
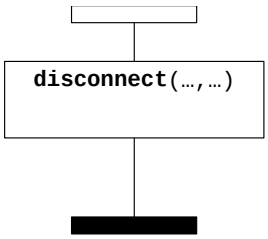
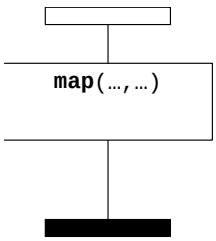
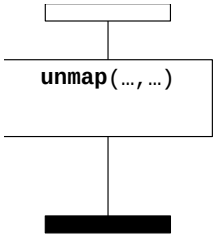
Data/signature template definitions	**template**		No special GFT symbol, i.e., the core notation or another presentation format may be used.
Control definitions	**control**		GFT control diagram represents the control part of a TTCN-3 module.
Function definitions	**function**	 	GFT function diagrams are used to represent functions. GFT function diagrams may be defined to structure the behavior of the control part of a TTCN-3 module.
Altstep definitions	**altstep**		GFT altstep diagrams are used to represent altsteps.

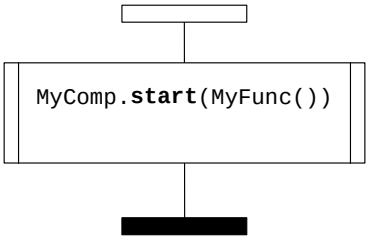
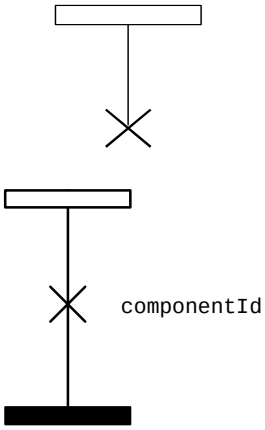
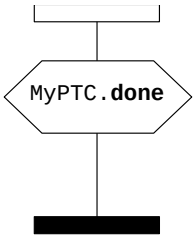
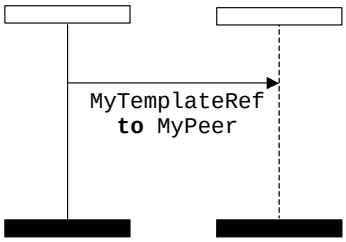
Test case definitions	**testcase**		GFT test case diagrams are used to represent test cases.
Usage of Component Instances and Ports			
Port instance			A Port in a test case, test step and function diagram is represented by and instance with a dashed instance line. The port name is specified above and the (optional) port type is described within the instance header.
Test component instance			An **mtc** instance represents the main test component in a test case diagram. A **self** instance represents a test component in a test step or function diagam. A **control** instance represents the instance that executes the module control part in a control diagram.
Declarations			
Variable declarations	**var**	**var integer** MyVar := 5 	Textual variable declaration in the header of a control, test case, test step or function diagram. Variable declaration in an action box. Variable declaration within a test case execution symbol.

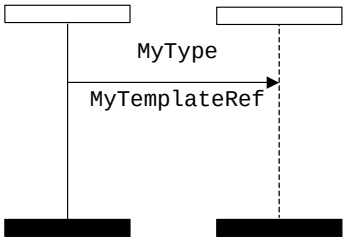
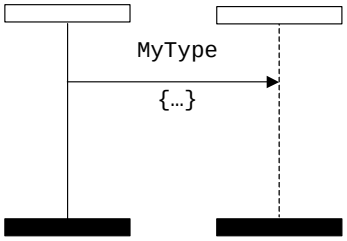
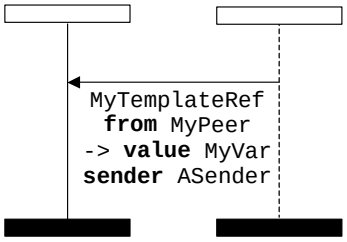
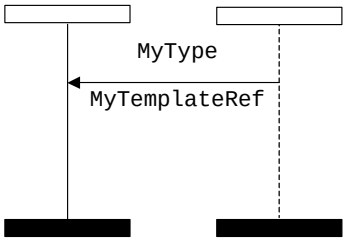
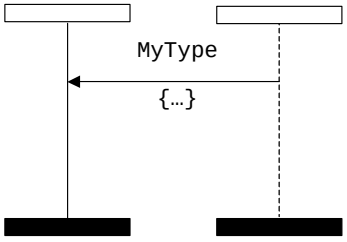
		  	Variable declaration within a test component creation symbol. Variable declaration within a default activation symbol. Variable declaration within a reference symbol.
Timer declarations	**timer**	**timer** MyTimer 	Textual timer declaration in the header of a control, test case, test step or function diagram. Timer declaration in an action box.
Basic program statements			
Expressions	**(…)**		No special GFT symbol, i.e., the core notation or another presentation format may be used.
Assignments	**:=**	 	Assignment in an action box. Assignment within a test case execution symbol.

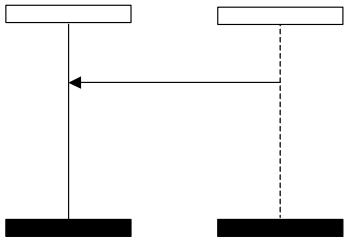
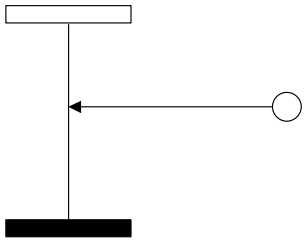
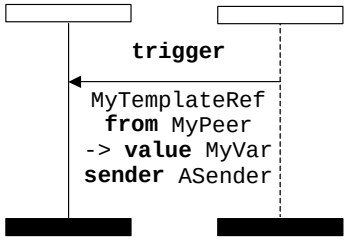
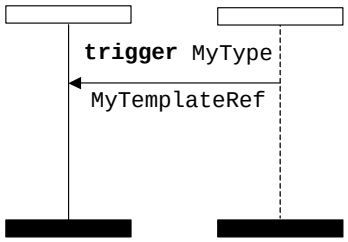
			Assignment within a test component creation symbol.
			Assignment within a default activation symbol.
			Assignment within a reference symbol.
Logging	**log**		The log statement is put into an action box.
Label and Goto	**label**		Definition of a label.
	goto		Go to label.
If-else	**if (…) {…}** **else {…}**		

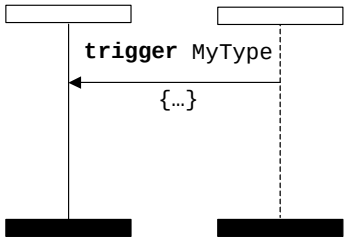
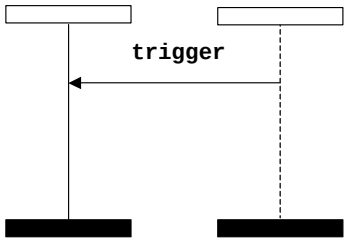
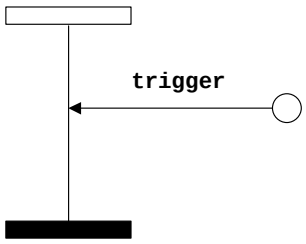
For loop	**for (…) {…}**		
While loop	**while (…) {…}**		
Do while loop	**do {…} while (…)**		
Behavioral program statements			
Alternative behavior	**alt {…}**		
Repeat	**repeat**		To be used within alternative behavior and test steps.
Interleaved behaviour	**interleave {…}**		
Activate a default	**activate**		The activate statement is put into a default symbol.

Deactivate a default	**deactivate**		The deactivate statement is put into a default symbol.
Returning control	**return**		The optional return value is attached to the return symbol.
Configuration operations			
Create parallel test component	**create**		The create statement is put into a test component creation symbol.
Connect component to component	**connect**		The connect statement is put into an action box.
Disconnect two components	**disconnect**		The disconnect statement is put into an action box.
Map port to test system interface	**map**		The map statement is put into an action box.
Unmap port from test system interface	**unmap**		The unmap statement is put into an action box.

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

			Send a message defined by a template reference and with type information. An (optional) **to**-directive may be present to identify the peer entity uniquely.
			Send a message defined by an inline template definition. An (optional) **to**-directive may be present to identify the peer entity uniquely.
Receive message	**receive**		Receive a message with a value defined by a template reference but without type information. The (optional) **from**-directive denotes that the sender of the message shall be identified by variable `MyPeer`. The (optional) **value**-directive assigns received message to variable `MyVar`. The (optional) **sender**-directive retrieves the identifier of the sender and stores it in variable `ASender`.
			Receive a message with a value defined by a template reference and with type information. Optional **from**-, **value**-and **sender**-directives may be present to identify the sender of the message, to assign the message to a variable or to retrieve the identifier of the peer entity.
			Receive a message with a value defined by an inline template definition. Optional **from**-, **value**-and sender-directives may be present to identify the sender of the message, to assign the message to a variable or to retrieve the identifier of the peer entity.

			Receive any message (no value and no type is specified). Optional **from**-, **value**-and sender-directives may be present to identify the sender of the message, to assign the message to a variable or to retrieve the identifier of the peer entity.
			Receive any message (no value and no type is specified) from any port. The message value to be received from any port may be restricted by means referring to templates or by using inline templates. Optional **from**-, **value**-and **sender**-directives may be present to identify the sender of the message, to assign the message to a variable or to retrieve the identifier of the peer entity.
Trigger message	**`trigger`**		Trigger on a message with a value defined by a template reference but without type information. The (optional) **from**-directive denotes that the sender of the message shall be identified by variable `MyPeer`. The (optional) **value**-directive assigns received message to variable `MyVar`. The (optional) **sender**-directive retrieves the identifier of the sender and stores it in variable `ASender`.
			Trigger on a message with a value defined by a template reference and with type information. Optional **from**-, **value**-and **sender**-directives may be present to identify the sender of the message, to assign the message to a variable or to retrieve the identifier of the peer entity.

			Trigger on a message with a value defined by an inline template definition. Optional **from**-, **value**-and **sender**-directives may be present to identify the sender of the message, to assign the message to a variable or to retrieve the identifier of the peer entity.
			Trigger on any message (no value and no type is specified). Optional **from**-, **value**-and **sender**-directives may be present to identify the sender of the message, to assign the message to a variable (of type **anytype**) and to retrieve the identifier of the peer entity.
			Trigger on any message (no value and no type is specified) from any port. The value of the message that shall cause the trigger from any port may be restricted by means referring to templates or by using inline templates. Optional **from**-, **value**-and **sender**-directives may be present to identify the sender of the message, to assign the message to a variable (of type **anytype**) and to retrieve the identifier of the peer entity.

Annex C (informative): Mapping GFT to TTCN-3 Core Notation

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

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

C.1 Approach

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

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

C.1.2 Overview of SML/NJ

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

C.1.2.1 Types and datatypes

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

C.1.2.2 Functions

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

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

C.1.2.3 Pattern Matching

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

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

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

C.1.2.4 Recursion

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

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

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

C.2 Modelling GFT Graphical Grammar in SML

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

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

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

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

C.2.1 SML Modules

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

```
signature MyStructSig =
sig
type MyStructType
val f : int -> int (* A function f that takes an integer argument and returns an integer *)
end
```

```
structure MyStruct : MyStructSig =
struct
    type MyStructType = int
    fun f x = x+1
end
```

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

C.2.2 Function Naming and References

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

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

C.2.3 Given Functions

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

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

```
val GuardOpLayer_to_ConnectorLayer        : gft_type.GuardOpLayer -> gft_type.ConnectorLayer
val UnguardOpLayer_to_ConnectorLayer      : gft_type.UnguardedOpLayer -> gft_type.ConnectorLayer
```

C.2.4 GFT and Core SML Types

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

C.2.5 GFT to CN Mapping Functions

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

structure gfttocn : GFTTOCN =
struct

open gft_type

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

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

(*************************************************************************
    p_ReceiveArea : ReceiveArea * InstanceLayer
                        -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_ReceiveArea  ((Type,
                     (SOME (DerivedDef,TemplateBody),FromClause,PortRedirect),
                     InstanceReceiveEventArea,
                     PortOutMsgEventArea),p)
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.RECEIVESTATEMENT(
                                        given_functions.get_attached_port(PortOutMsgEventArea,p),
                                        (SOME(Type,DerivedDef,TemplateBody),FromClause,PortRedirect)
                                )
                        )
                )
 | p_ReceiveArea   ((Type,
                     (NONE,FromClause,PortRedirect),
                     InstanceReceiveEventArea,
                     PortOutMsgEventArea),p)
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.RECEIVESTATEMENT(
                                        given_functions.get_attached_port(PortOutMsgEventArea,p),
                                        (NONE,FromClause,PortRedirect)
```

```
                                )
                        )
                )
(* end of p_ReceiveArea *)

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

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

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

```
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.REPLYSTATEMENT(
                                        given_functions.get_attached_port(PortReplyInEventArea,p),
                                        ((Signature,DerivedDef,TemplateBody),ReplyValue,ToClause)
                                )
                        )
                )
(* end of p_ReplyArea *)

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

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

(*****************************************************************************
    p_CatchOutsideCallArea : CatchOutsideCallArea * InstanceLayer
                        -> cn_type.FunctionStatementOrDef
*****************************************************************************)
```

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

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

```
                                        PortRedirect
                                        )
                                )
                        )
                )
(* End of p_TriggerArea *)

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

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

```
            )
(* end of p_CheckOpInformation *)

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

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

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

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

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

(* end of p_InstanceCallEventArea *)

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

```
        end

(* end of p_InstanceGetCallEventArea *)

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

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

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

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

(*************************************************************************
    p_InstanceCatchTimeoutWithinCallEventArea :
        InstanceCatchTimeoutWithinCallEventArea *
                        (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                        -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_InstanceCatchTimeoutWithinCallEventArea (InstanceCatchTimeoutWithinCallEventArea,
(_,InstanceLayer,ConnectorLayer,_)) =
        let
        in
        cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.CATCHSTATEMENT(
                        given_functions.get_attached_port(PortCatchoutEventArea,p),
                                        (
```

```
                                        SOME (cn_type.CATCHTIMEOUT),
                                        NONE,
                                        NONE
                                        )
                                )
                        )
                )
        end
(* end of p_InstanceCatchTimeoutWithinCallEventArea *)

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

(* end of p_InstanceTriggerEventArea*)

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

(* end of p_InstanceCheckEventArea*)

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

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

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

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

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

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

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

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

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

"any",(SOME(Type,DerivedDef,TemplateBody),FromClause,PortRedirect)
                                )
                        )
                )
 | p_FoundMessage (Type,(NONE,FromClause,PortRedirect))
        =
```

```
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.RECEIVESTATEMENT(
                                        "any",(NONE,FromClause,PortRedirect)
                                )
                        )
                )

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

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

(*****************************************************************************
    p_FoundGetReply : FoundGetReply -> cn_type.FunctionStatementOrDef
*****************************************************************************)
fun p_FoundGetReply (Signature,
                     (SOME (DerivedDef,TemplateBody,ValueMatchSpec),
                     FromClause,
                     PortRedirectWithParam))
        =
                cn_type.FUNCTIONSTATEMENT(
                        cn_type.COMMUNICATIONSTATEMENTS (
                                cn_type.GETREPLYSTATEMENT(
                                        "any",
                                        (
                                        SOME ((Signature,DerivedDef,TemplateBody),ValueMatchSpec),
                                        FromClause,
                                        PortRedirectWithParam
```

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

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

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

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

(*************************************************************************
    p_ActionStatement : ActionStatement -> cn_type.FunctionStatementOrDef
*************************************************************************)
```

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

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

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

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

(* end of p_InstanceDoneArea *)

(*************************************************************************
    p_SetVerdictArea : SetVerdictArea -> cn_type.FunctionStatementOrDef
*************************************************************************)
fun p_SetVerdictArea (SETVERDICTKEYWORD SingleExpression) =
                cn_type.FUNCTIONSTATEMENT (
            cn_type.VERDICTSTATEMENTS SingleExpression)
 |  p_SetVerdictArea (PASSKEYWORD)      =
                cn_type.FUNCTIONSTATEMENT (
            cn_type.VERDICTSTATEMENTS "pass")
 |  p_SetVerdictArea (FAILKEYWORD)      =
                cn_type.FUNCTIONSTATEMENT (
            cn_type.VERDICTSTATEMENTS "fail")
 |  p_SetVerdictArea (INCONCKEYWORD)    =
                cn_type.FUNCTIONSTATEMENT (
            cn_type.VERDICTSTATEMENTS "inconc")
 |  p_SetVerdictArea (NONEKEYWORD)      =
                cn_type.FUNCTIONSTATEMENT (
            cn_type.VERDICTSTATEMENTS "none")
(* end of p_SetVerdictArea *)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(*****************************************************************************
    p_InstanceComponentStartArea : InstanceComponentStartArea
                                        -> cn_type.FunctionStatementOrDef
*****************************************************************************)
fun p_InstanceComponentStartArea StartTCStatement =
                    cn_type.FUNCTIONSTATEMENT (
                            cn_type.CONFIGURATIONSTATEMENTS (
                                cn_type.STARTTCSTATEMENT StartTCStatement))
(* end of p_InstanceComponentStartArea *)

(*****************************************************************************
    p_InstanceEventArea : InstanceEventArea *
(InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer)
                                -> cn_type.FunctionStatementOrDef list
```

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

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

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

```
            SOME ElseInstanceEventAreaList,                         (* Else Event list *)
           (InstanceEventLayer,InstanceLayer,ConnectorLayer,PortEventLayer))
        cn_type.FUNCTIONSTATEMENT (
            cn_type.BASICSTATEMENTS (
                cn_type.CONDITIONALCONSTRUCT
                    (BooleanExpression,
                     p_InstanceEventAreaList IfInstanceEventAreaList
                    (InstanceEventLayer,InstanceLayer,OperandaArea1,PortEventLayer),
                     [],     (* No if else clauses - represented by nested inline expressions *)
                     SOME (p_InstanceEventAreaList ElseInstanceEventAreaList
                    (InstanceEventLayer,InstanceLayer,OperandaArea2,PortEventLayer))
                     )
                )
            )
(* end of p_IfArea *)

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

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

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

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

```
                    cn_type.WHILESTATEMENT (BooleanExpression,
                    p_InstanceEventAreaList InstanceEventAreaList
                    (InstanceEventLayer,InstanceLayer,OperandaArea,PortEventLayer)
                    )
                )
            )
        )
(* end of p_WhileArea *)

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


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

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

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

(*************************************************************************
p_InstanceBooleanExpressionConditionArea :InstanceBooleanExpressionConditionArea
                    -> cn_type.AltGuardChar
*************************************************************************)
fun p_InstanceBooleanExpressionConditionArea x = p_BooleanExpressionConditionArea x
(* end of p_InstanceBooleanExpressionConditionArea *)

(*************************************************************************
p_InstanceGuardOpArea : InstanceGuardOpArea -> cn_type.GuardOp
*************************************************************************)
fun p_InstanceGuardOpArea ((INSTANCEGUARDRECEIVEEVENTAREA x),p) =
    let
```

```
        val (cn_type.FUNCTIONSTATEMENT(
             cn_type.COMMUNICATIONSTATEMENTS(
                 cn_type.RECEIVESTATEMENT ReceiveStatement
                 )
             )
         ) = p_InstanceReceiveEventArea (x,p)
    in
        cn_type.GUARDRECEIVESTATEMENT ReceiveStatement
    end
(* end of p_InstanceGuardOpArea *)

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

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

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

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

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

```
                cn_type.BEHAVIOURSTATEMENTS (
                    cn_type.ALTCONSTRUCT (
                        p_InstanceGuardArea
(InstanceGuardArea,(InstanceEventLayer,InstanceLayer,GuardedOperandArea,PortEventLayer))
                            ::p_InstanceGuardAreaList
(InstanceGuardAreaList,(InstanceEventLayer,InstanceLayer,GuardedOperandAreaList,PortEventLayer)),
                        p_InstanceElseGuardArea
(InstanceElseGuardArea,(InstanceEventLayer,InstanceLayer,ElseOperandArea,PortEventLayer))
                    )
                )
            )
    (* end of p_AltArea *)

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

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

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

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

    (*************************************************************************
    p_InterleaveArea : InterleaveArea * InstanceGuardArea * InstanceGuardArea list *
InstanceElseGuardArea option*
```

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

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

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

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

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

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

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

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

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

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

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

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

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

Annex D (normative): Mapping TTCN-3 Core Notation to GFT

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

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

D.1 Approach

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

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

D.1.2 Overview of SML/NJ

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

D.2 Modelling GFT Graphical Grammar in SML

D.2.1 SML Modules

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

D.2.2 Function Naming and References

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

D.2.3 Given Functions

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

D.2.4 Core and GFT SML Types

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

D.2.5 Core to GFT Mapping Functions

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

****

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

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

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

*****************************************************************************
p_TimerStatements : TimerStatements
                    -> gft_type.InstanceTimerEventArea
*****************************************************************************

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

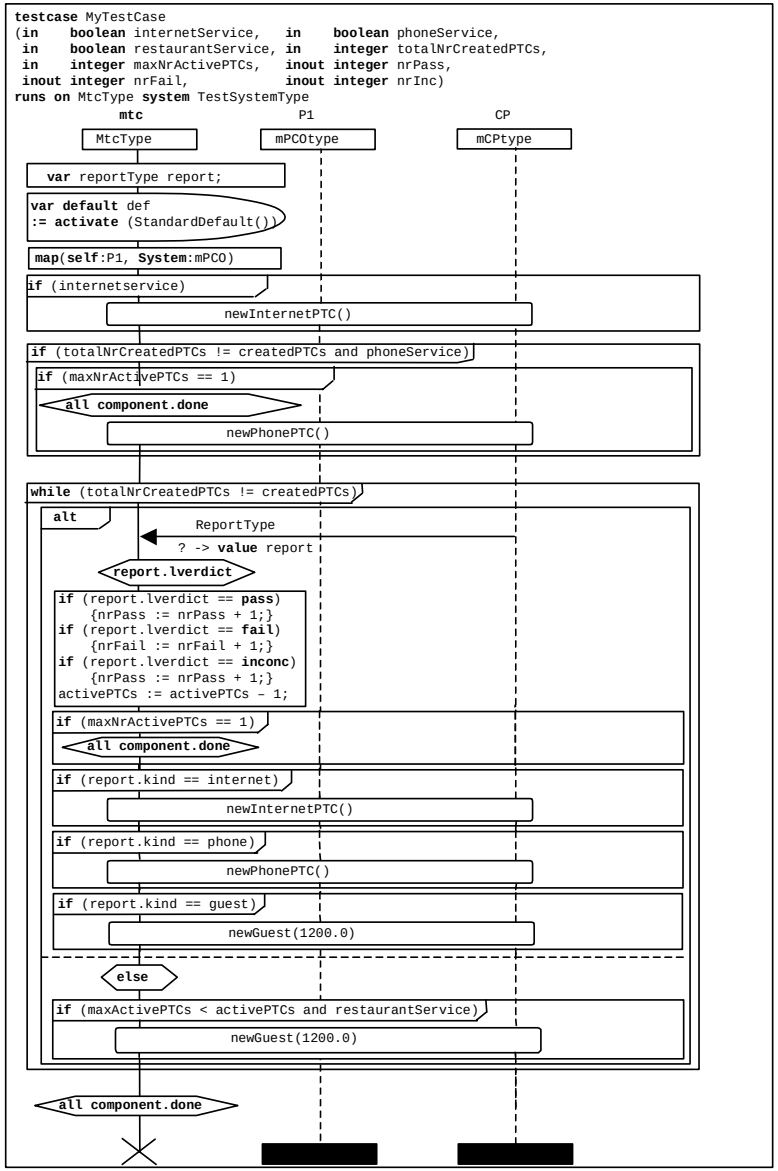
```
*****************************************************************************
p_GetReplyStatement : TestcaseFormalPar * GetReplyStatement
        -> gft_type.GetReplyWithInCallArea * gft_type.GetReplyOutsideCallArea
*****************************************************************************

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

*****************************************************************************
p_StatementBlock : StatementBlock * WithStatement
                                        -> gft_type.TestcaseBodyArea
*****************************************************************************

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

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

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

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

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

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

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

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

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

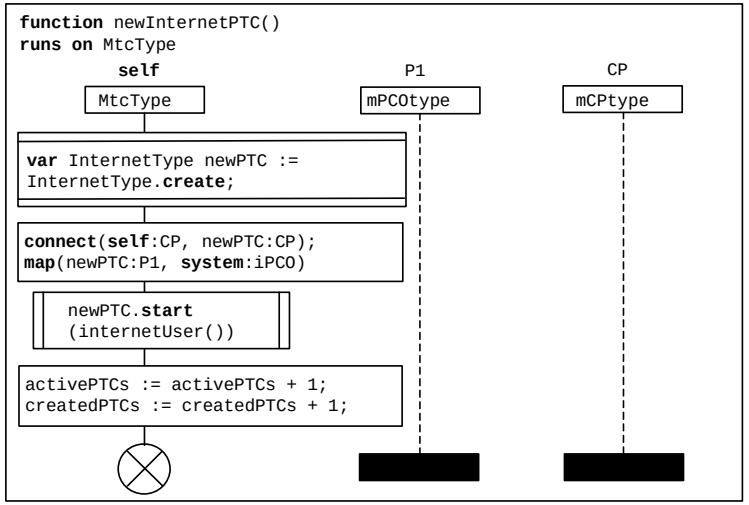
Annex E (informative): Examples

E.1 The Restaurant Example

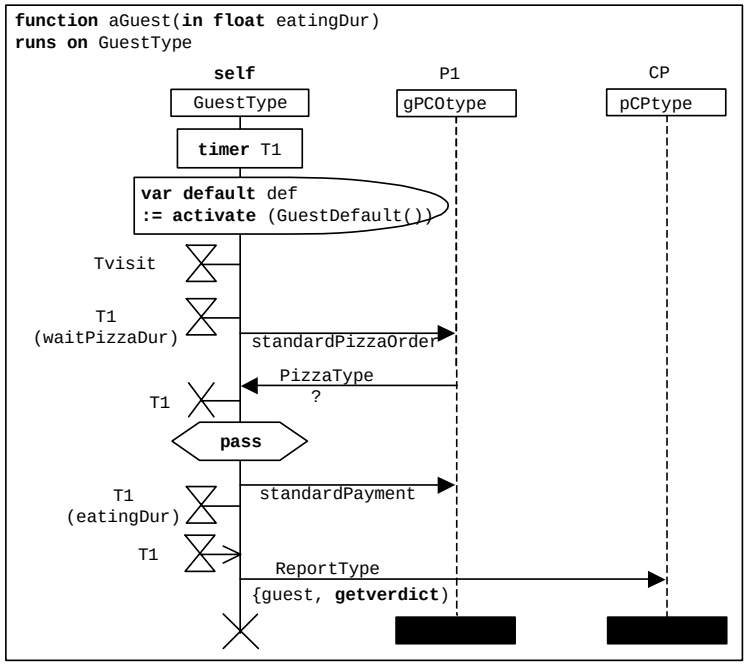



```
testcase MyTestCase (
    in boolean          internetService,     // SERVICES
    in boolean          phoneService,
    in boolean          restaurantService,
    in integer          totalNrCreatedPTCs,  // TERMINATION
    in integer          maxNrActivePTCs,     // CONTROL
    inout integer   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

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

    map(self:P1, system:mPCO);

    if (internetService)  {
        newInternetPTC();
    }

    if (totalNrCreatedPTCs != createdPTCs
        and phoneService) {

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

    while ( totalNrCreatedPTCs != createdPTCs ) {

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

                setverdict(report.lverdict);

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

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


            [else] {

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

    all component.done;
    stop;
  }
```

Figure 1: Restaurant example – MyTestCase test case

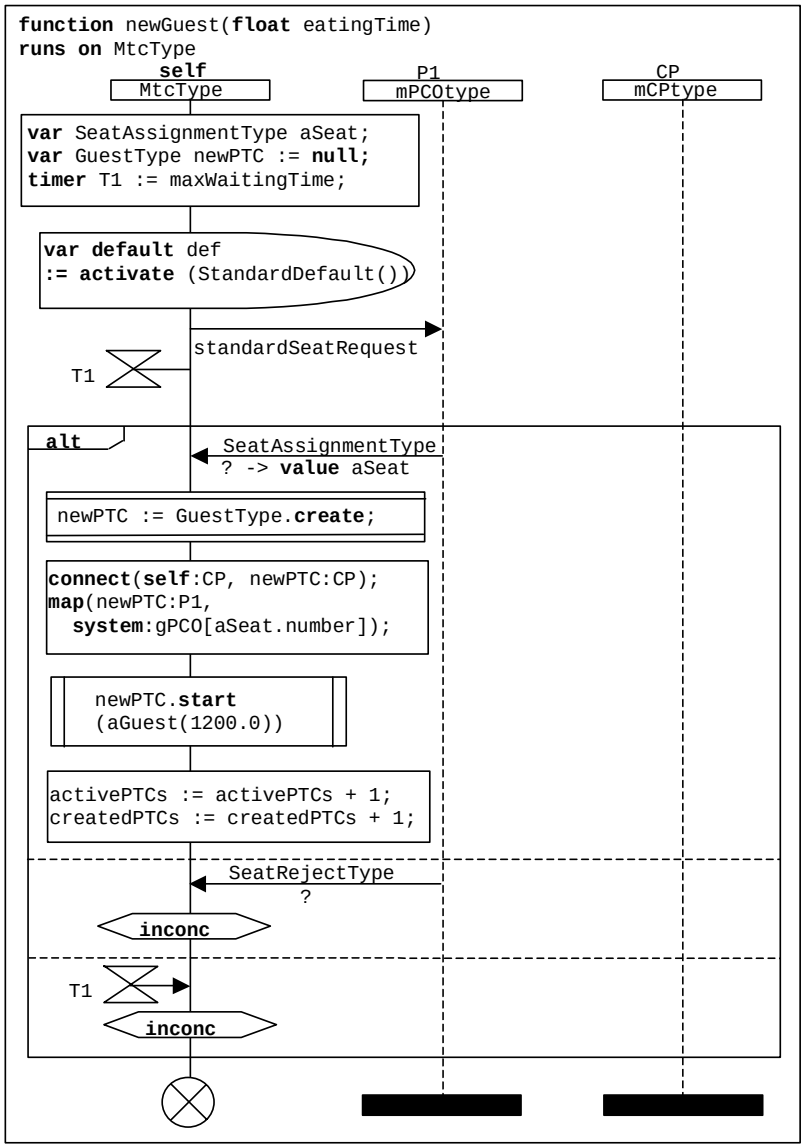



```
function newInternetPTC ()
   runs on MtcType {

   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;

}
```



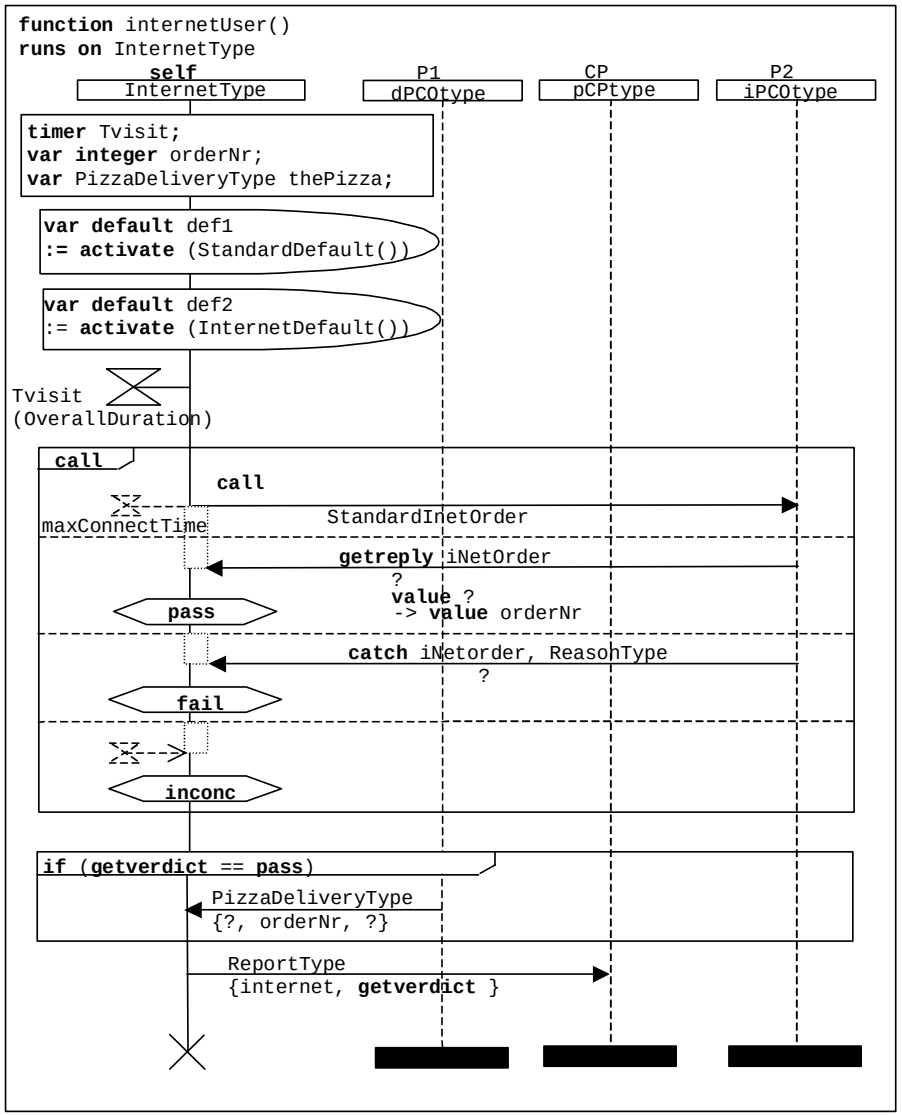

```
function aGuest (in float eatingDur) runs on GuestType {

   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 2: Restaurant example – newInternetPTC and aGuest functions



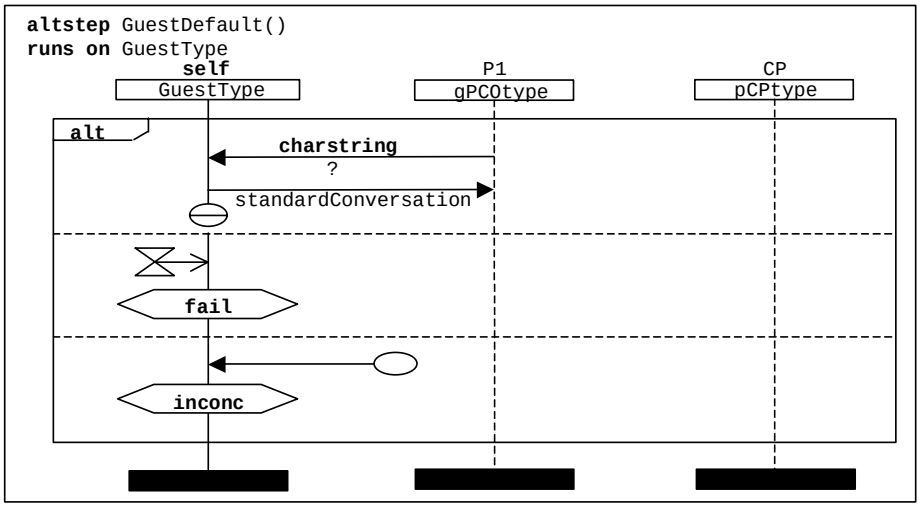

```
function newGuest (float eatingTime) runs on MtcType {

   var SeatAssignmentType aSeat;
   var GuestType newPTC := null;
   timer T1 := maxWaitingTime;

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

   // Request for a seat
   P1.send(standardSeatRequest);
   T1.start;

   alt {
      [] P1.receive(SeatAssignmentType:?) -> value aSeat {
         newPTC := GuestType.create;

         connect(self:CP, newPTC:CP);
         map(newPTC:P1, system:gPCO[aSeat.number]);

         newPTC.start(aGuest(1200.0));

         activePTCs  := activePTCs+1; // Update MTC variables
         createdPTCs := createdPTCs+1;
      }

      [] P1.receive(SeatRejectType:?) { // No seat assigned
         setverdict(inconc);
      }

      [] T1.timeout { // No answer on seat request
         setverdict(inconc);
      }
    }
    return;
}
```

Figure 3: Restaurant example – newGuest function

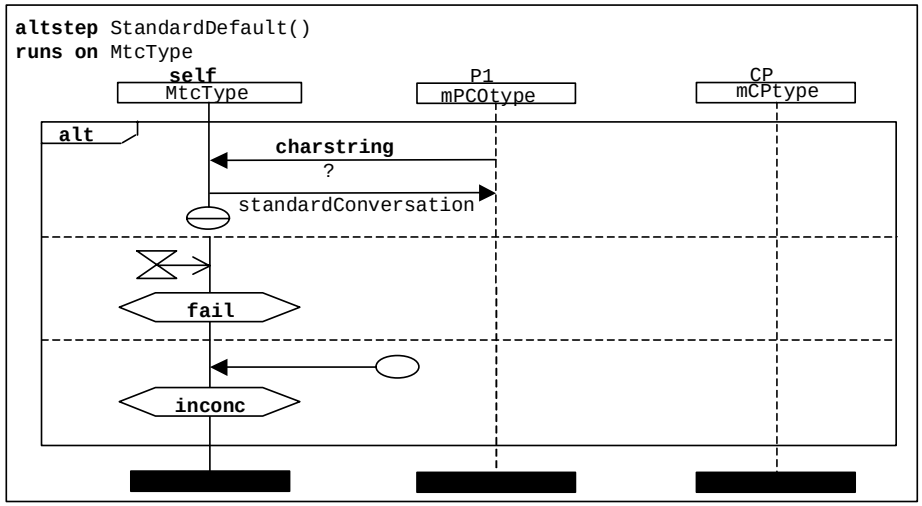



```
function internetUser () runs on InternetType {
  // ***
  // *** Purpose: Specifies the behaviour of an
  // *** internet guest
  // ***

  timer Tvisit;
  var integer orderNr;
  var PizzaDeliveryType thePizza;

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

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

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

Figure 4: Restaurant example – internetUser function

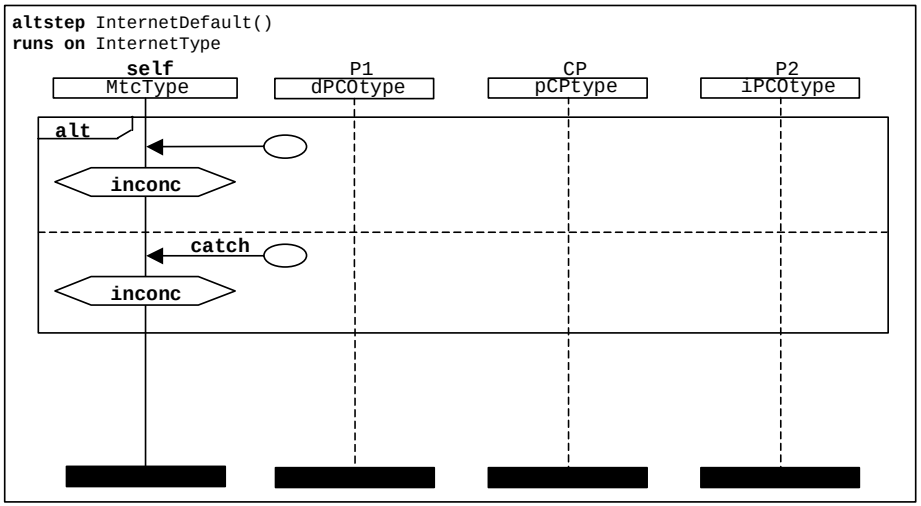



```
altstep GuestDefault() runs on GuestType {
   // ***
   // *** Purpose: Default behaviour for
   // *** message based ports
   // ***

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

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

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

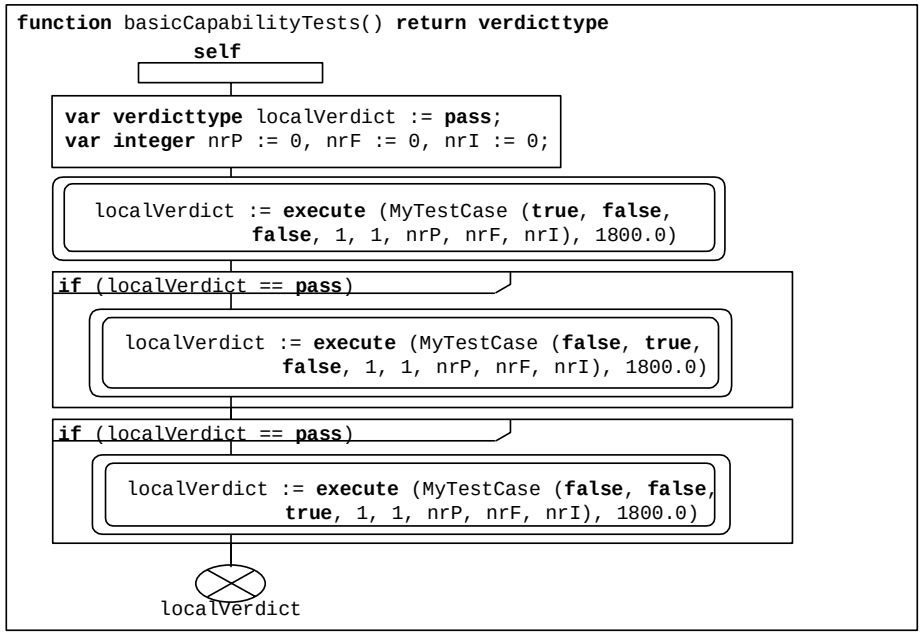



```
altstep StandardDefault() runs on MtcType {
   // ***
   // *** Purpose: Default behaviour for
   // **** message based ports
   // ***

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

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

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

Figure 5: Restaurant example – GuestDefaut and StandardDefault functions

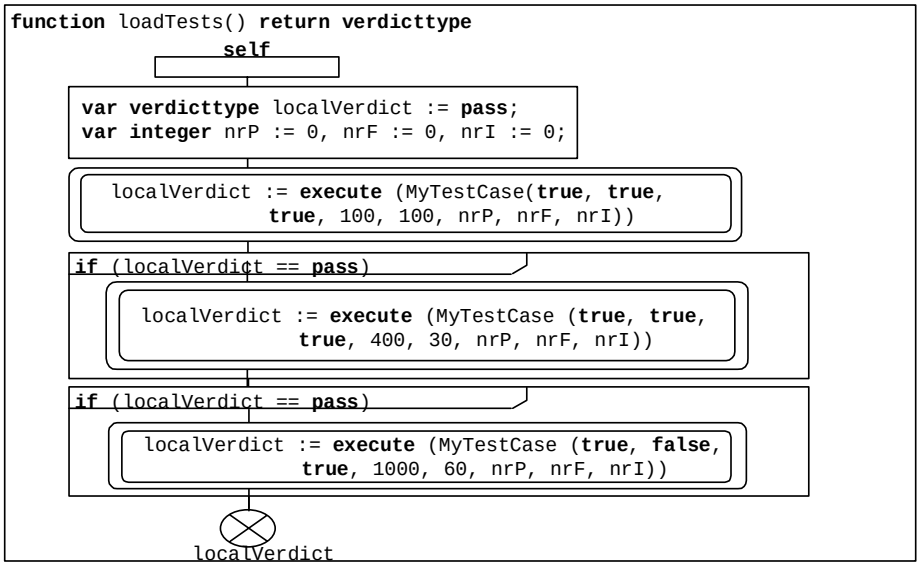



```
altstep InternetDefault()
   runs on InternetType {
   // ***
   // *** Purpose: Default behaviour for
   // **** the procedure based port
   // ***

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

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

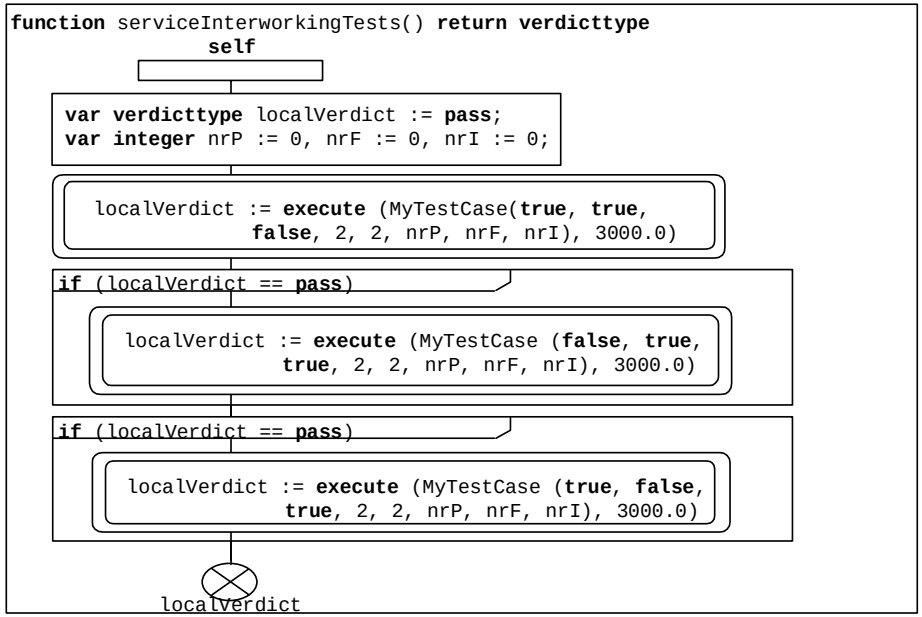



```
function basicCapabilityTests ()
   return verdicttype {
   var verdicttype localVerdict := pass;
   var integer nrP := 0, nrF := 0, nrI  := 0;

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

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

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

Figure 6: Restaurant example – internetDefault altstep and basicCapabilityTests function

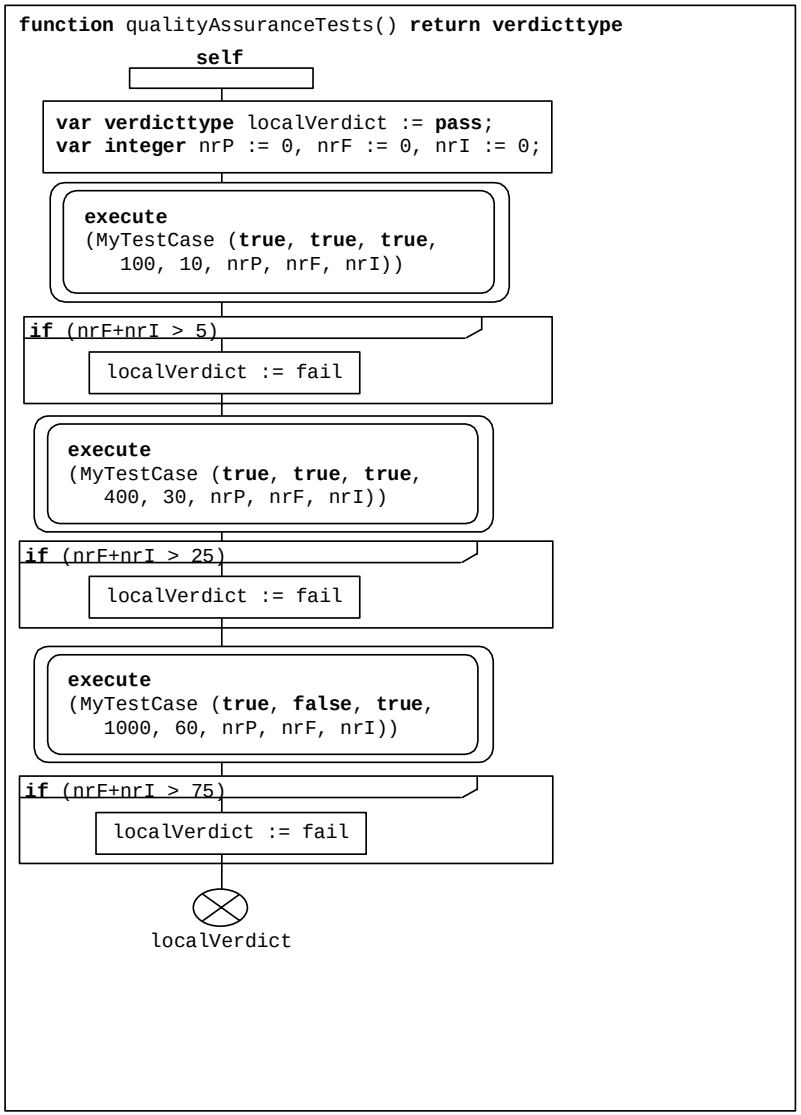



```
function loadTests () return verdicttype {
   var verdicttype localVerdict := pass;
   var integer nrP := 0, nrF := 0, nrI  := 0;

   // *** Minimal load ***
   localVerdict := execute(MyTestCase(
        true,true,true,100,10,nrP,nrF,nrI));

   // *** Medium load ***
   if (localVerdict == pass) {
      localVerdict := execute(MyTestCase(
         true,true,true,400,30,nrP,nrF,nrI));
    }

   // *** Maximal load ***
   if (localVerdict == pass) {
      localVerdict := execute(MyTestCase(
         true,false,true,1000,60,nrP,nrF,nrI));
    }
    return (localVerdict);
}
```



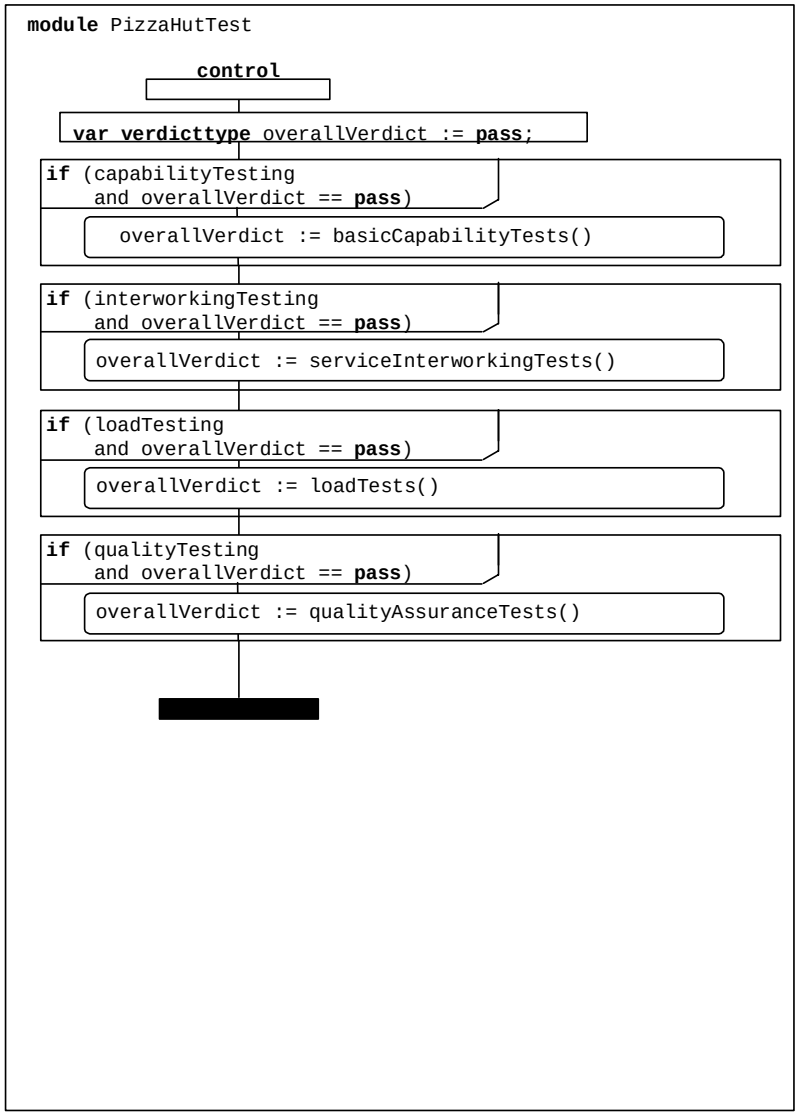

```
function serviceInterworkingTests ()
   return verdicttype {
   var verdicttype localVerdict := pass;
   var integer nrP := 0, nrF := 0, nrI  := 0;

   // *** INTERNET ORDER & PHONE ORDER ***
   localVerdict := execute(MyTestCase(
      true,true,false,2,2,nrP,nrF,nrI),3000.0);

   // *** PHONE ORDER & RESTAURANT ORDER
   if (localVerdict == pass) {
      localVerdict := execute(MyTestCase(
      false,true,true,2,2,nrP,nrF,nrI),3000.0);
   }

   // *** RESTAURANT ORDER & INTERNET ORDER***
   if (localVerdict == pass) {
      localVerdict := execute(MyTestCase(
      true,false,true,2,2,nrP,nrF,nrI),3000.0);
   }

   return (localVerdict);
}
```

Figure 7: Restaurant example – loadTests and serviceInterworkingTests functions




```
function qualityAssuranceTests ()
   return verdicttype {

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

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

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

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

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

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

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

   return (localVerdict);

} // end function qualityAssuranceTests
```

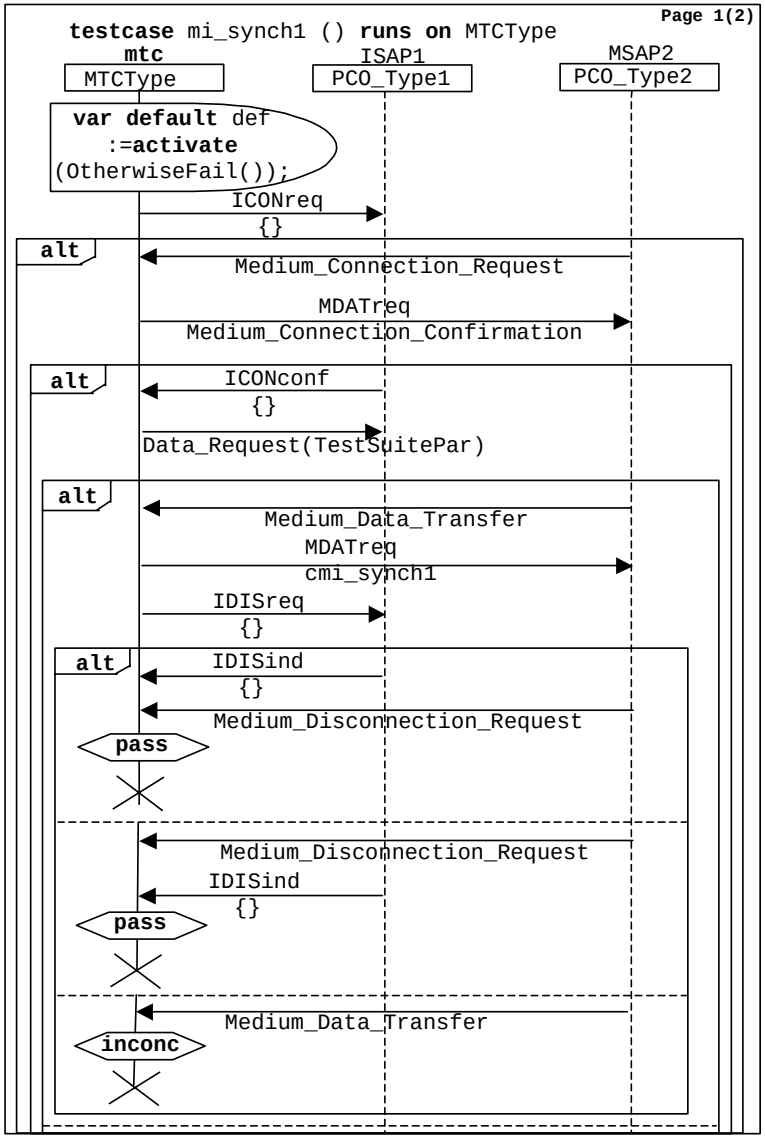
Figure 8: Restaurant example – qualityAssuranceTests function




```
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();
   }

}
```

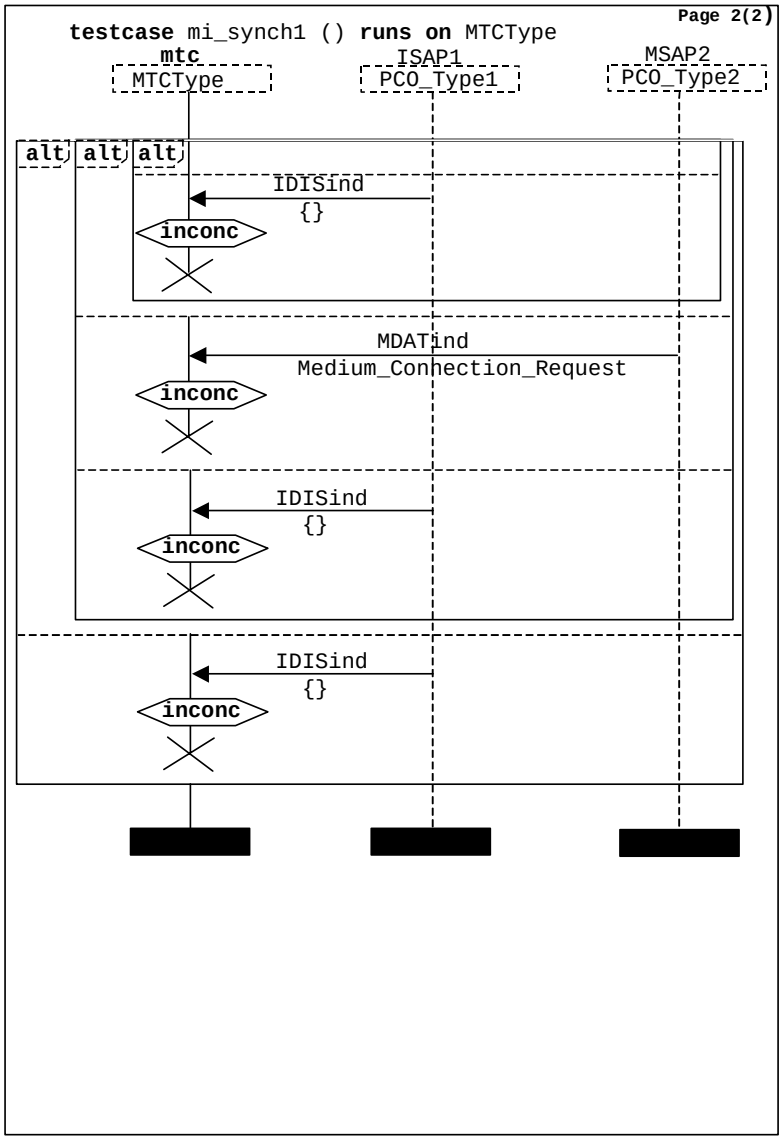
Figure 9: Restaurant example – PizzaHutTest module

E.2 The INRES Example



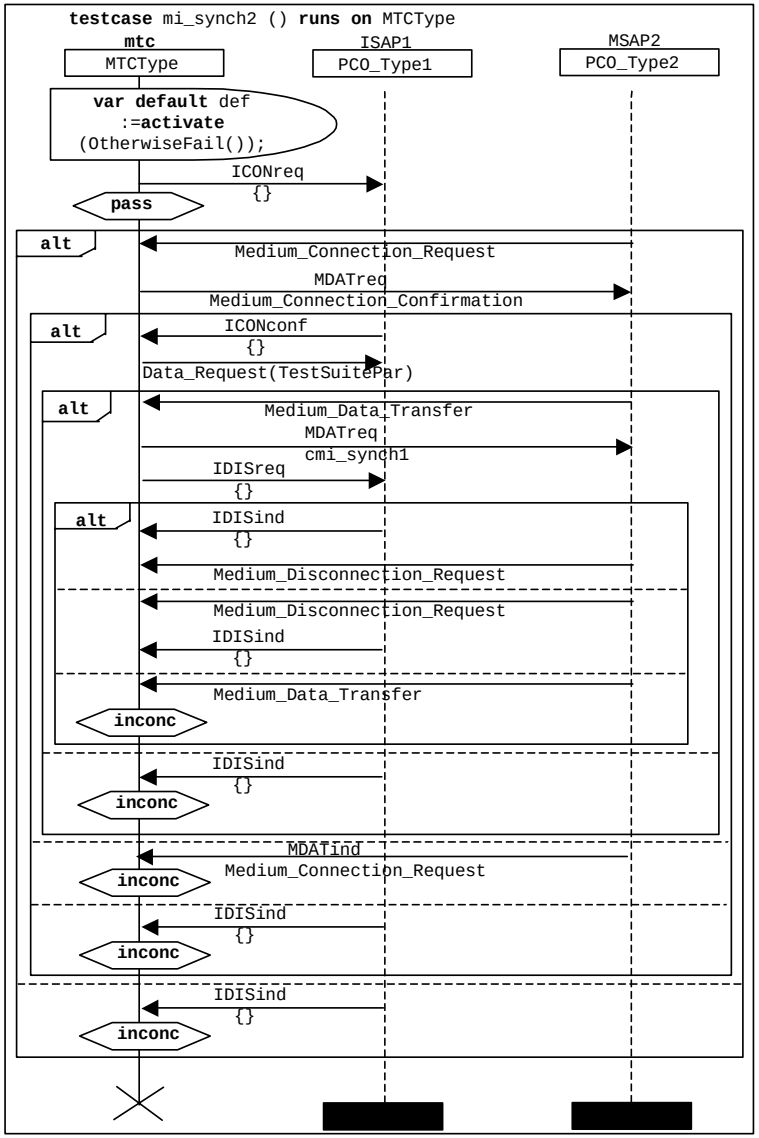

```
testcase mi_synch1 () runs on MTCType {

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

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

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

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


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



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

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

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

Figure 10: INRES example - mi_synch1 1(2) test case



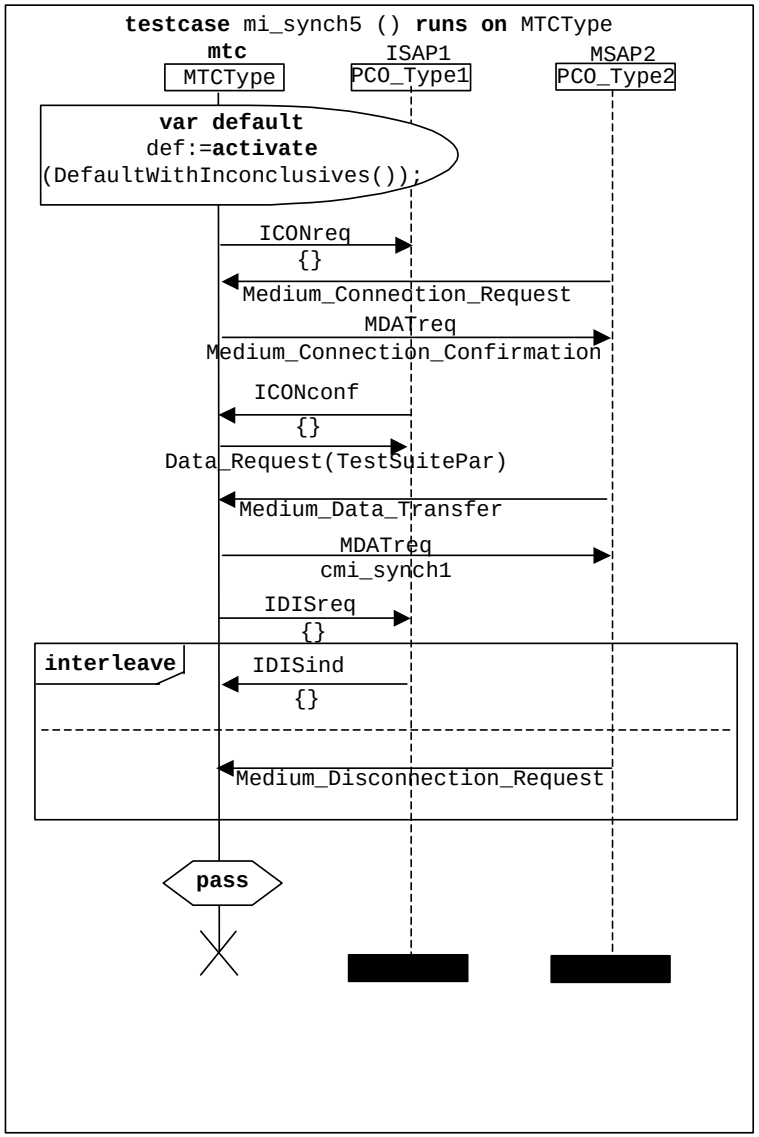

```
/* testcase mi_synch1 () continuation */




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

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

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


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

Figure 11: INRES example - mi_synch1 2(2) test case



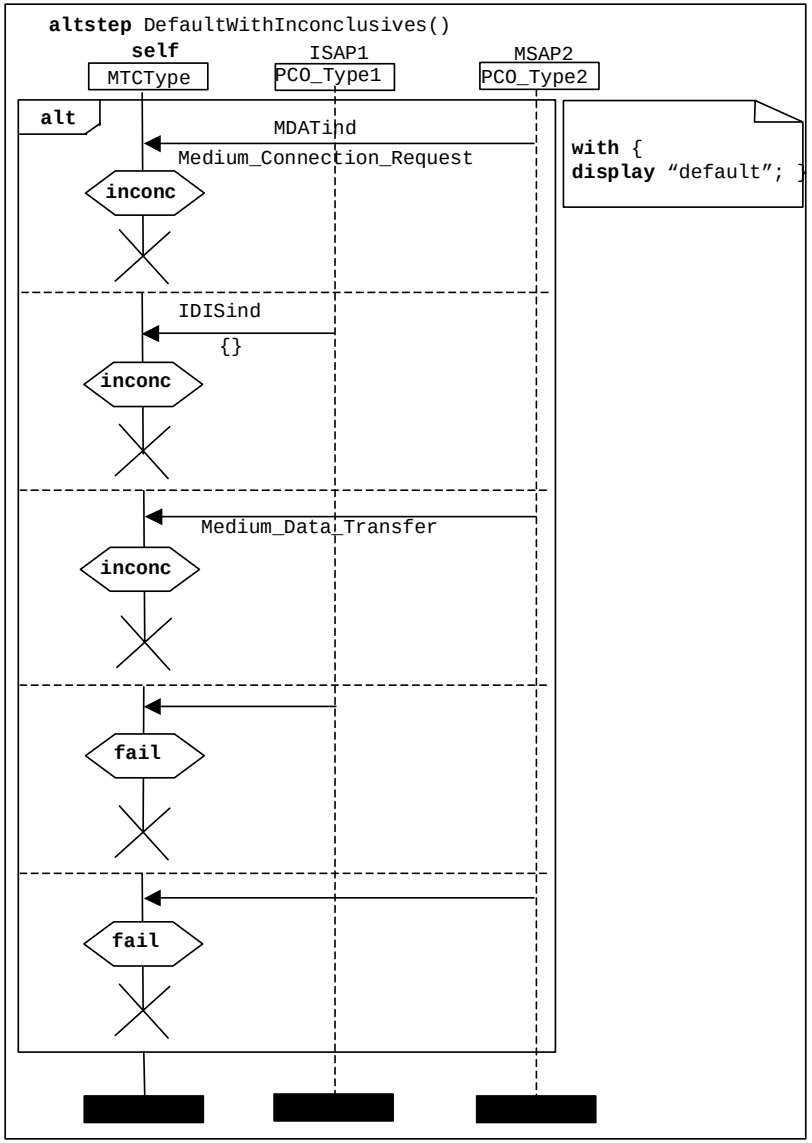

```
testcase mi_synch2 () runs on MTCType {



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

Figure 12: INRES example - mi_synch2 test case



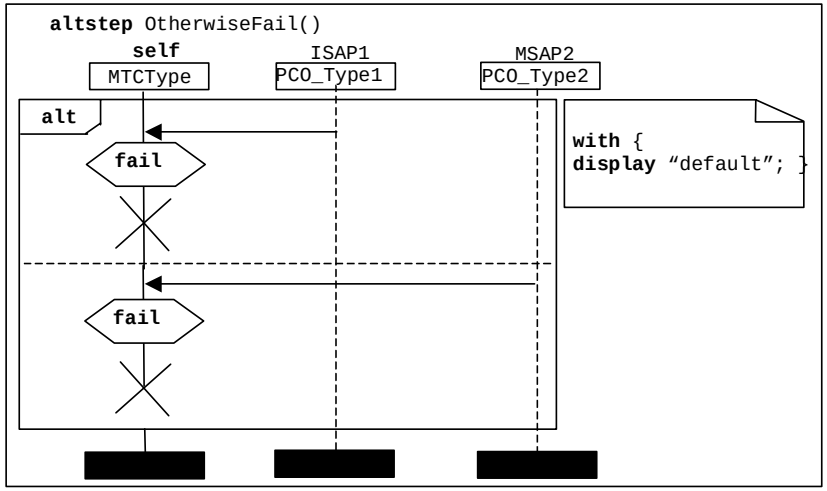

```
testcase mi_synch5 () runs on MTCType {



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

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

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

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

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

  setverdict(pass);

  stop;

} /* End testcase mi_synch5 */
```

Figure 13: INRES example - mi_synch5 test case

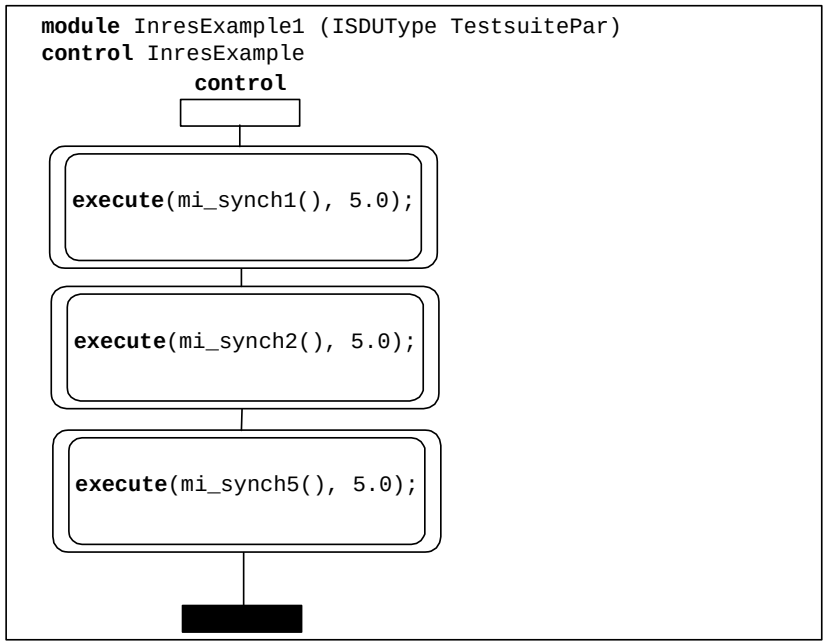



```
altstep DefaultWithInconclusives() {

   /* INCONCLUSIVE CASES */


  [] MSAP2.receive( MDATind:Medium_Connection_Request())

     setverdict(inconclusive);
     stop;
  }


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

     setverdict(inconclusive);
     stop;
  }



  [] MSAP2.receive ( Medium_Data_Transfer() ) {

     setverdict(inconclusive);
     stop;
  }


 /* FAIL CASES */

  [] ISAP1.receive {

     setverdict(fail);
     stop;

  }


  [] MSAP2.receive {

     setverdict(fail);
     stop;
  }


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

Figure 14: INRES example - DefaultWithInconclusives altstep




```
altstep OtherwiseFail() {

   [] ISAP1.receive {

      setverdict(fail);

      stop;
   }


   [] MSAP2.receive {

      setverdict(fail);

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




```
module InresExample1 ( ISDUType TestsuitePar ) {

   control InresExample {

      execute (mi_synch1(), 5.0);

      execute (mi_synch2(), 5.0);

      execute (mi_synch5(), 5.0);

   } // end control part

}
```

Figure 15: INRES example - OtherwiseFail altstep and InresExample1 module definitions

Annex F (informative): GFT to MSC mapping

Although GFT is based on MSC, 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 (see also Section 8). For the acceptance of GFT in various user communities as well as in standardization committees like ITU and OMG, the convergence between GFT and MSC appears to be crucial. Apart from few test specific extensions, MSC-2000 diagrams can represent GFT. Although a mapping from GFT to MSC-2000 can be achieved, the new symbols and inscription conventions in GFT essentially facilitate the use of the graphical test format in conjunction with TTCN-3. Obviously, after mapping GFT into MSC the obtained diagrams often appear quite clumsy and difficult to handle. As such, GFT can be looked at simply as a shorthand notation of an MSC representation of TTCN-3. At present, GFT is restricted to a component-oriented representation. The MSC representation reflects this view though it also provides the bridge to a connection-oriented view. It demonstrates that MSC is capable to provide both a component oriented and a connection oriented view.

F.2 GFT diagrams

F.2.1 Control diagram

The control diagram is translated into an MSC with a control instance that creates an mtc instance for the execution of test cases (see Figure 1). The verdict value is returned from the mtc instance to the control instance by means of a result message. The time supervision is mapped into the setting of a timer and a subsequent alternative whereby in case of timeout, the verdict is set to **error**.

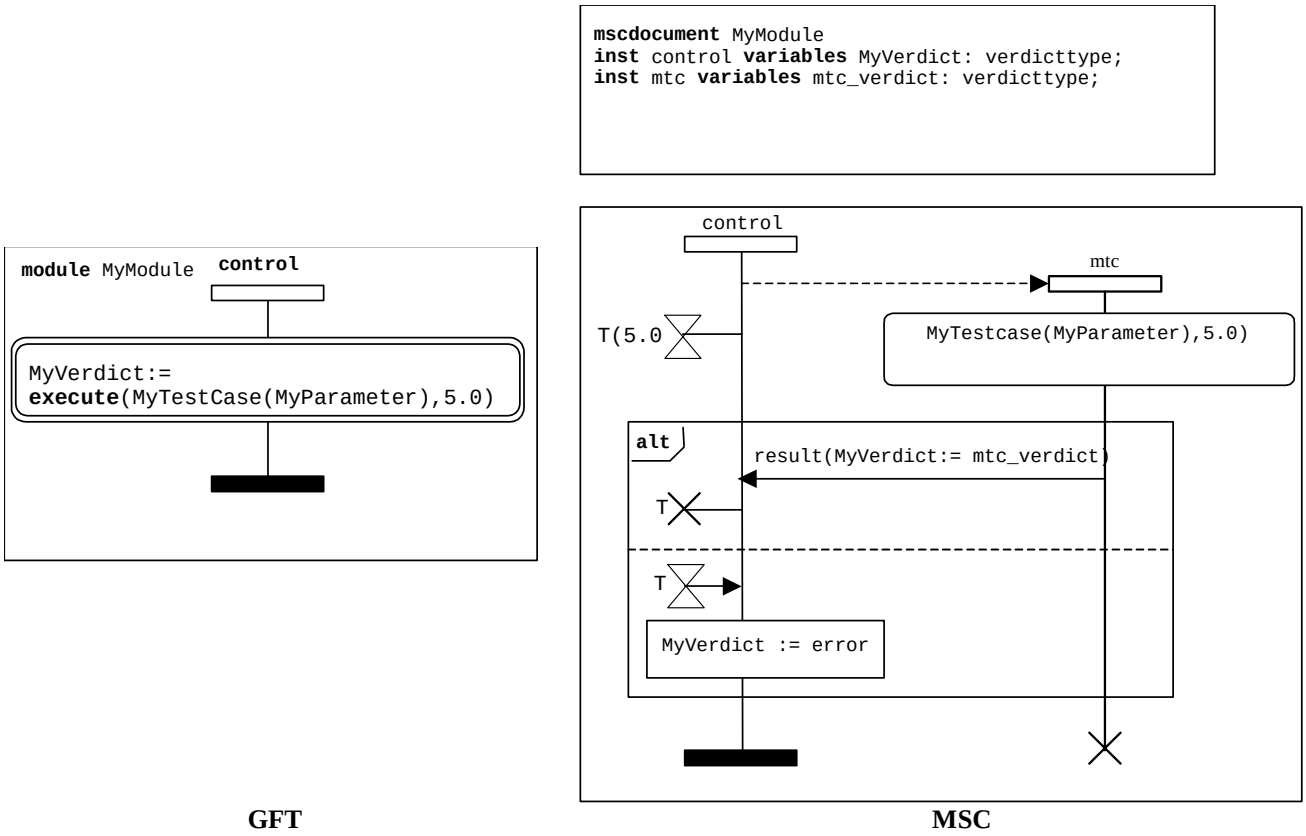


Figure 1: GFT-MSC: Control diagram

F.2.2 Test case diagram

A test case is mapped into an MSC with an mtc instance and communication events from and to port instances (see Figure 2). The declaration part is moved to the MSC document level. In/out parameters have to be expressed in form of variables owned by the mtc instance that can be modified within the test case. Parameter values have to be assigned to the variables before calling the reference.

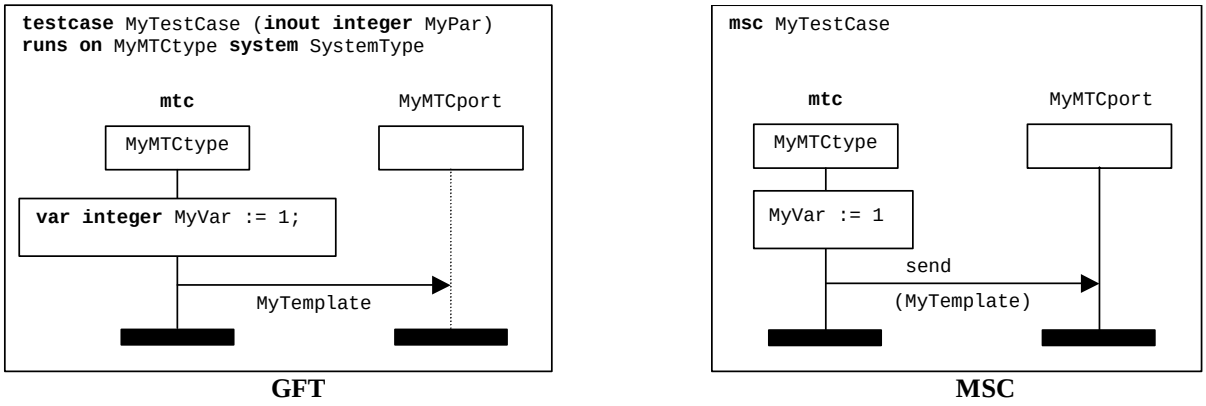


Figure 2: GFT-MSC: Test case diagram

F.2.3 Function diagram

A function diagram is mapped into an MSC with a test component instance and communication events from and to port instances (see Figure 3). The declaration part is moved to the MSC document level. In/out parameters and return variables have to be expressed in form of variables owned by the test component instance that can be modified within the test case. The name of the test component is self. The MSC heading has to contain "self" as an instance name parameter. The GFT return symbol is represented in MSC by an instance end symbol.

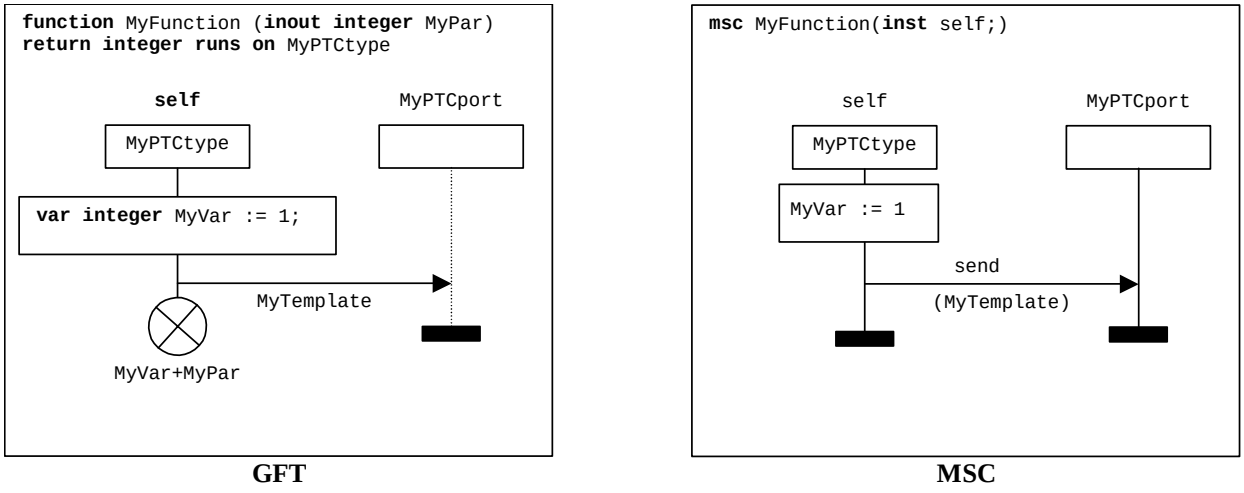


Figure 3: GFT-MSC: Function diagram

F.2.4 Altstep diagram

An altstep diagram is mapped into an MSC with a test component instance and communication events from and to port instances. The declaration part is moved to the MSC document level. In/out parameters have to be expressed in form of variables owned by the test component instance that can be modified within the test case. The name of the test component is self. The MSC heading has to contain "self" as an instance name parameter. In the else branch of the alternative inline expression, a sequence of guards is used which is an extension to MSC.

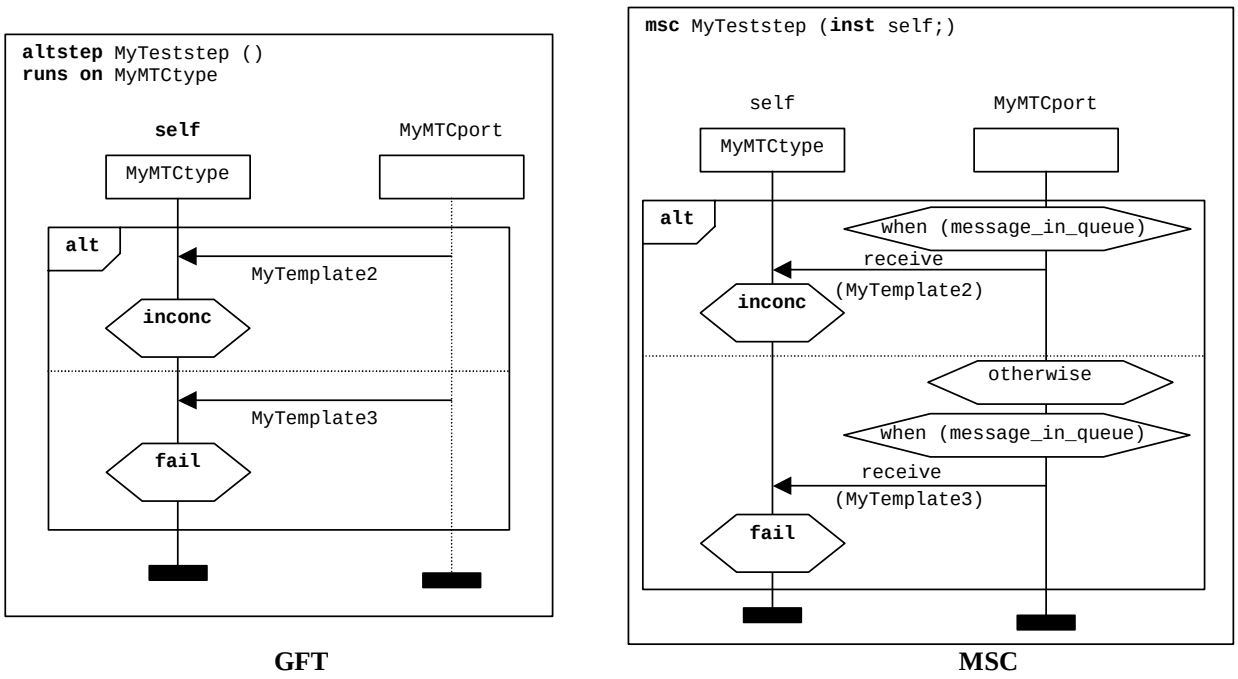


Figure 4: GFT-MSC: Altstep diagram

F.3 Instances in GFT diagrams

F.3.1 Control instance

The GFT control instance is represented in MSC by an instance with the name control.

F.3.2 Test component instances

The GFT test component instance is represented in MSC by an instance with name “mtc” inside of a testcase diagram and with name “self“ inside of a function or altstep diagram. The optional test component type may be placed within the test component heading in the presence of both, component name and component type, in MSC the component type name has to be put above the instance head and the instance name has to be put inside the instance head, contrary to GFT.

F.3.3 Port instances

GFT port instances are mapped in MSC into normal instances with solid instance axes. For the placement of port name and port type again the MSC rules for instance name and instance type have to be obeyed.

F.4 Invoking GFT diagrams

F.4.1 Execution of test cases

The GFT execution of test cases is mapped in MSC into an MSC reference on an mtc instance, which is created by the control instance.

F.4.2 Invocation of functions

The GFT invocation of functions is mapped in MSC into MSC references with inserting the instance name of the calling component instance for the instance name parameter (see Figure 5).

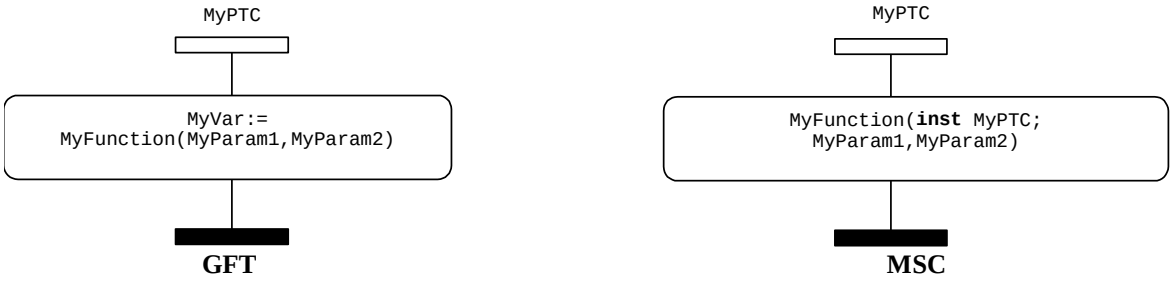


Figure 5: GFT-MSC: Function invocation

F.4.3 Invocation of altsteps

GFT invocation of altsteps is mapped in MSC into MSC references with inserting the instance name of the calling component instance for the instance name parameter (see Figure 6).

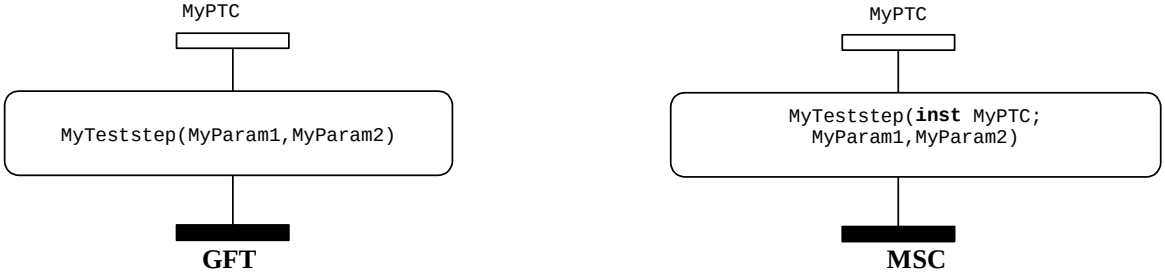


Figure 6: GFT-MSC: Altstep invocation

F.5 Basic program statements

F.5.1 The Log statement

The log statement can be mapped onto an action box with text (see Figure 7).

Figure 7: GFT-MSC: The Log Statement

F.5.2 The Label statement and Goto statement

Since Goto exists only in HMSCs the basic MSCs corresponding to the GFT diagrams are translated into HMSCs with a corresponding connection to a node representing the label (see Figure 8).

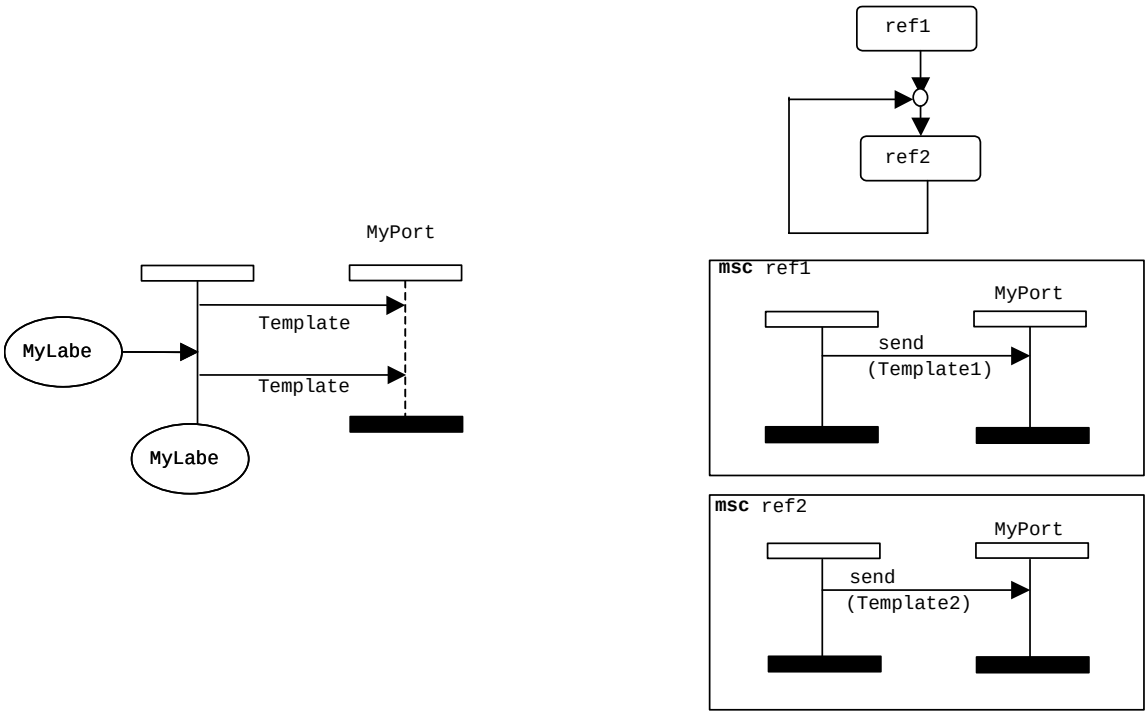


Figure 8: GFT-MSC: The Label and Goto Statement

F.5.3 The if-else statement

The if statement is translated into an optional inline expression with a guard on the sending instance (see Figure 9). Note that if the port is the sending instance, then the guard has to be placed on the port.

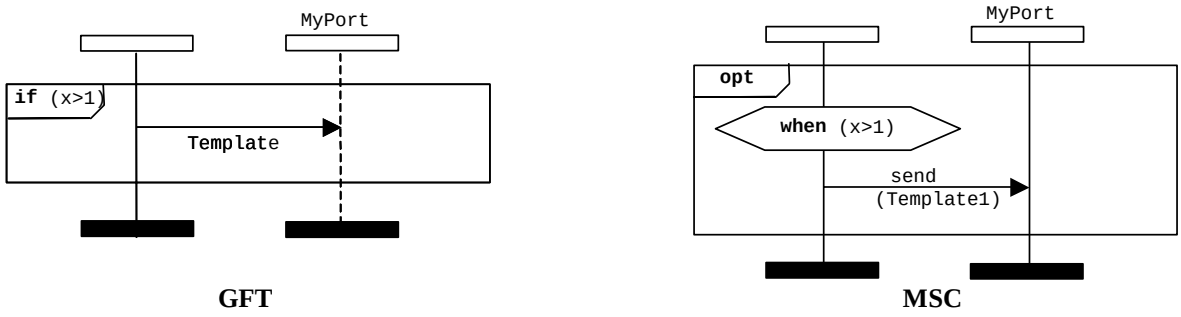
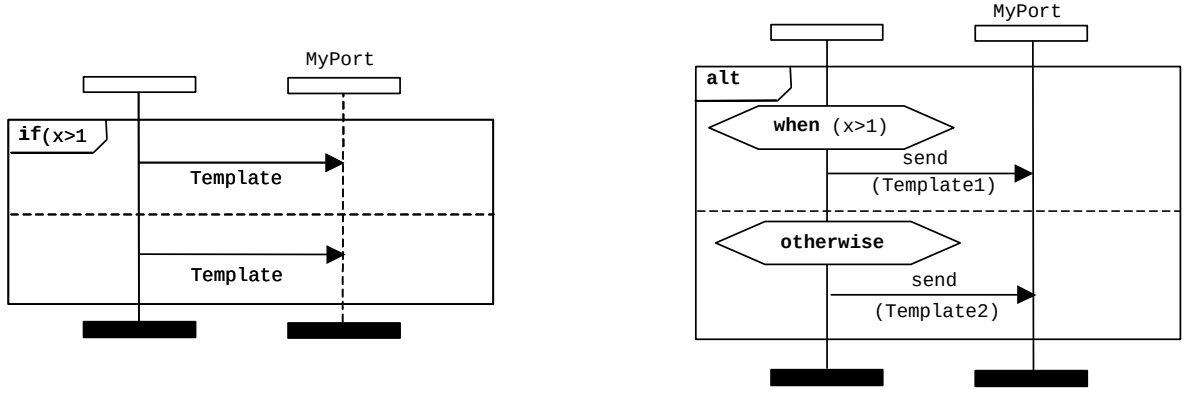


Figure 9: GFT-MSC: If Statement

The if else statement translates into an alternative inline expression with an else branch (see Figure 10).

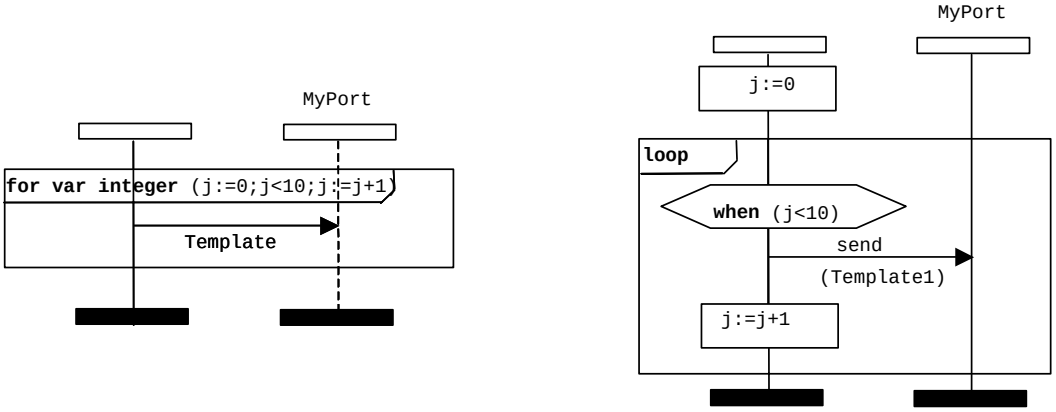


GFT **MSC**

Figure 10: GFT-MSC: If-elseStatement

F.5.4 The For statement

The for statement is represented by a loop with guarding condition and an index variable which is increased within the loop (e.g., j := j+1). The initial value is assigned to the index variable before the loop (see Figure 11).

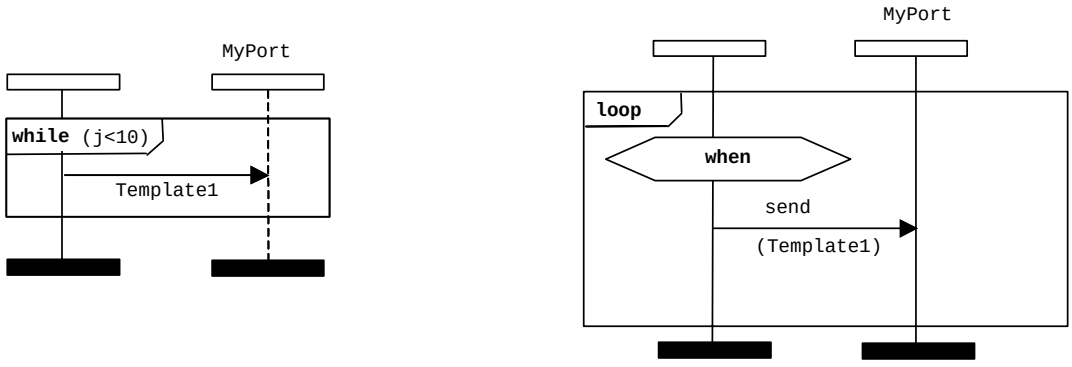


GFT **MSC**

Figure 11: GFT-MSC: For Statement

F.5.5 The While statement

The while statement is represented by a loop with guarding condition in the beginning (see Figure 12).



GFT **MSC**

Figure 12: GFT-MSC: While Statement

F.5.6 The Do-while statement

The do-while statement in GFT is represented within MSC by a loop with guarding condition in the beginning and one preceding execution of the enclosed behaviour (see Figure 13).

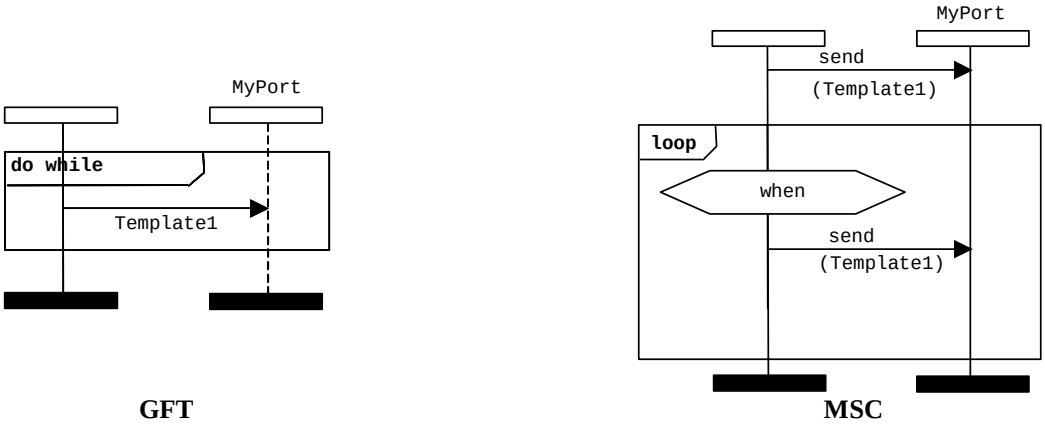


Figure 13: GFT-MSC: Do-while Statement

F.6 Behavioural Program Statements

F.6.1 Sequential Behaviour

Sequential behaviour in GFT is translated into a corresponding sequence of events or references in MSC (see Figure 14).

Figure 14: GFT-MSC: Sequential behaviour

F.6.2 Alternative Behaviour

The GFT alternative behaviour is translated in MSC into nested alternative inline expressions with else-branches (see Figure 15). The prioritisation from top to bottom in GFT is modelled by expressing each subsequent alternative as part of the else branch of the preceding alternative in MSC. The reception of messages has to be guarded by an MSC guarding condition with the guard: “message_in_queue”. In the else branch of the alternative inline expression, a sequence of guards is used which is an extension to MSC.

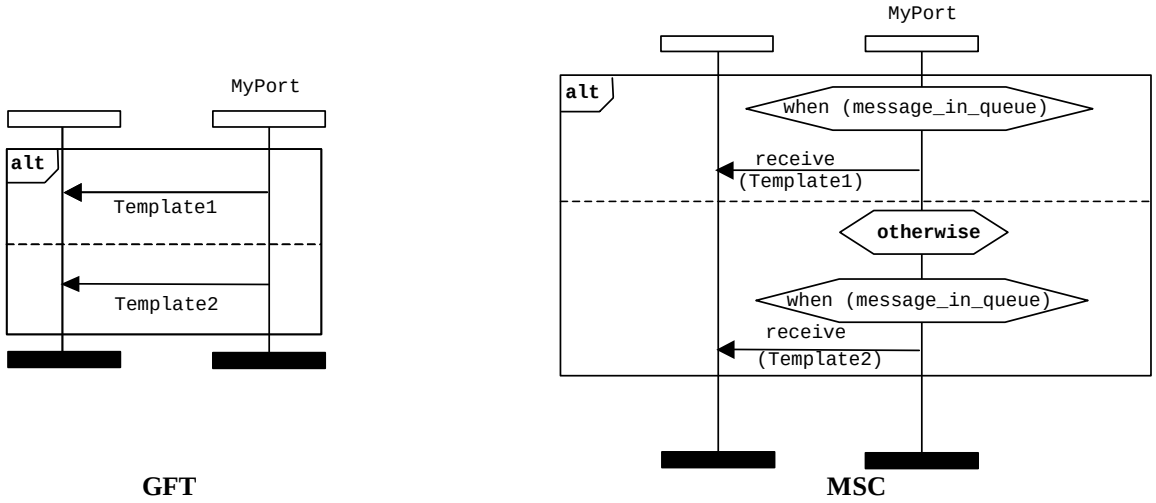


Figure 15: GFT-MSC: Alternative behaviour statement

F.6.3 Selecting/Deselecting an Alternative

The GFT selection of an alternative is mapped in MSC into nested alternative inline expressions with else-branches and guards for the reception of messages together with guards for the selection of alternatives (see Figure 16). The guards for the selection of alternatives in GFT are attached to the component instance whereas in MSC they have to be shifted to the port instance.

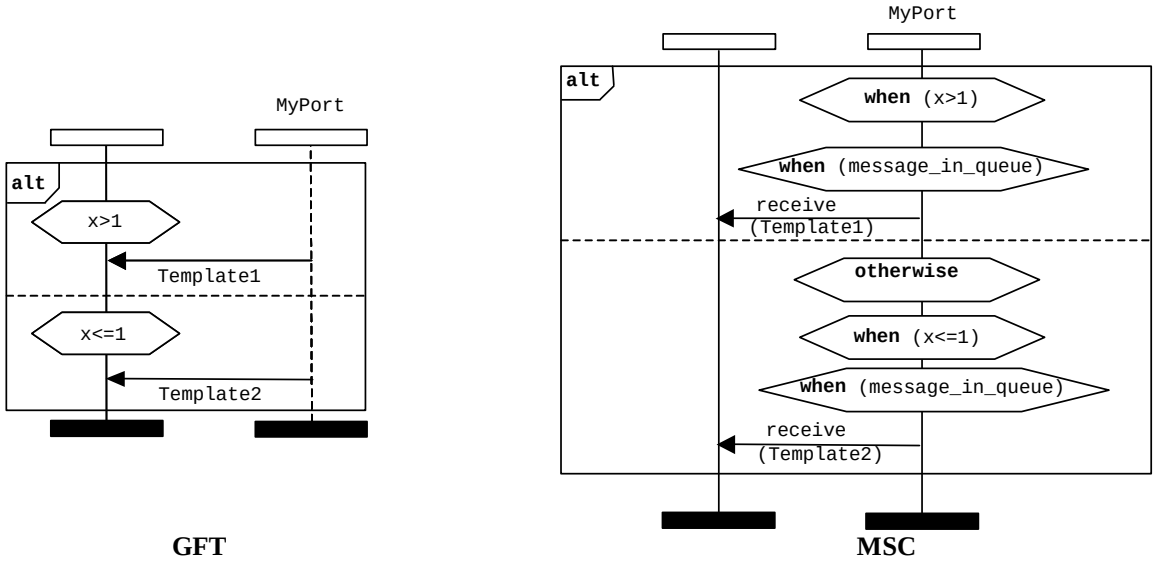


Figure 16: GFT-MSC: Selecting/Deselecting an alternative

F.6.4 Else branch in Alternative

An else branch in GFT is mapped in MSC into an otherwise branch with respect to all preceding alternatives (see Figure 17). The preceding alternatives are treated in the same manner as the alternative behaviour without else branch. The else condition on the component instance in GFT is shifted to the port instance in MSC with the keyword otherwise. Note, that according to MSC-2000 the placement of the 'otherwise' guard is depending on the first sending event in the MSC reference.

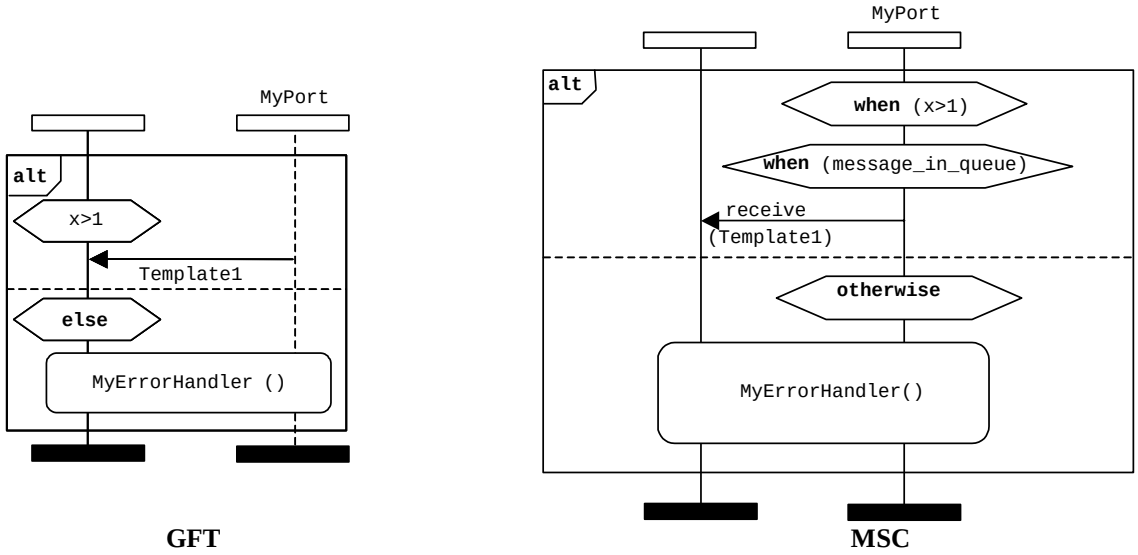


Figure 17: GFT-MSC: Else within an alternative

F.6.5 The Repeat statement

The GFT repeat statement is mapped in MSC into a loop inline expression enclosing the alternatives, whereby the loop is guarded by a guard condition which is initially true and which is set false within the alternatives without a repeat statement (see Figure 18). In general, the situation may be more complicated and needs further research.

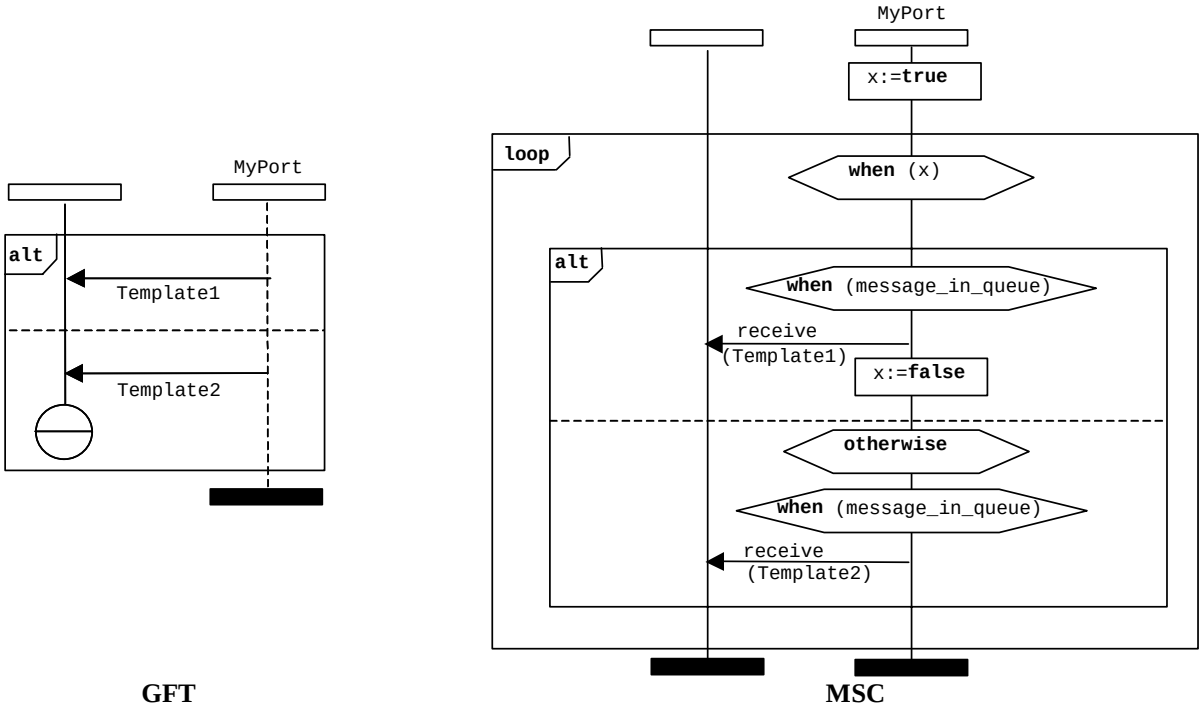
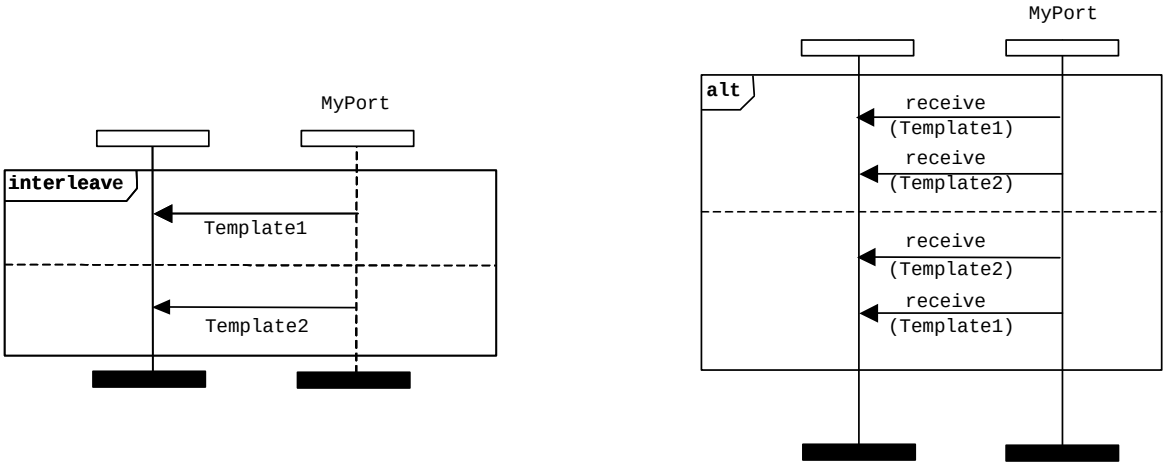


Figure 18: GFT-MSC: Repeat within an alternative

F.6.6 Interleaved Behaviour

The interleaved behaviour in GFT is mapped in MSC into an alternative inline expression of all allowed interleavings according to the expanded form in TTCN-3 (see Figure 19).

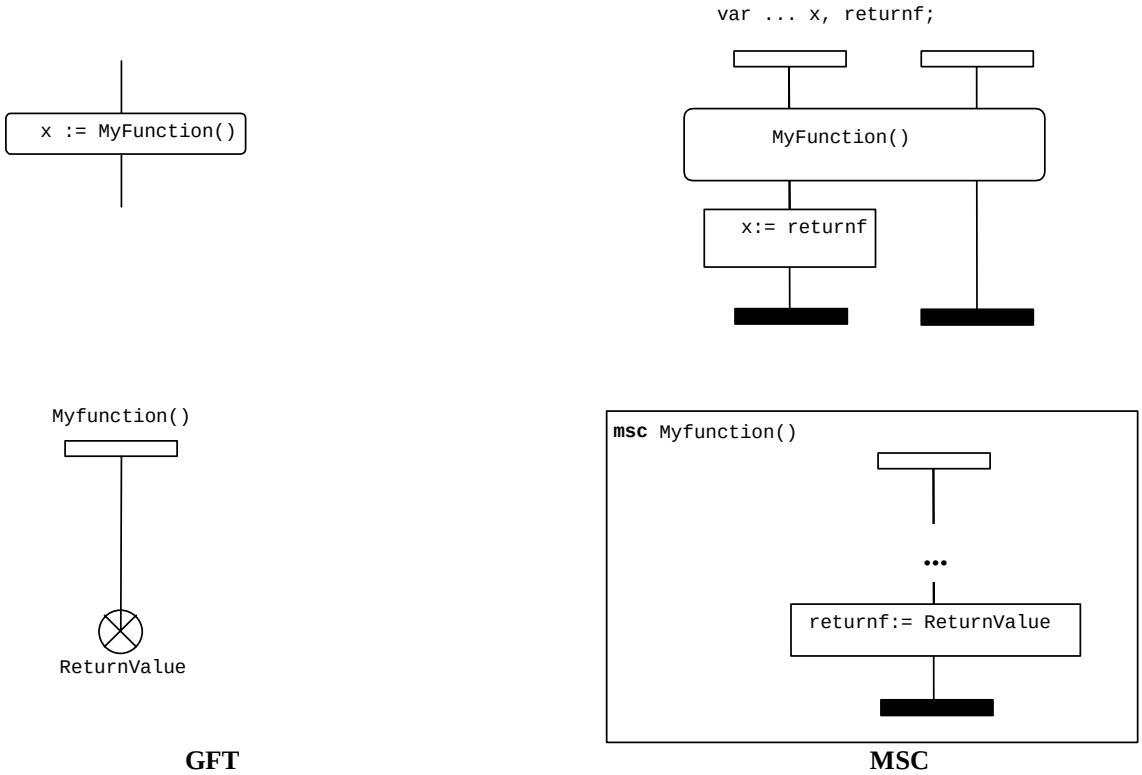


GFT **MSC**

Figure 19: GFT-MSC: Interleave Statement

F.6.7 The Return statement

The GFT return statement is mapped in MSC to the end of an MSC reference (see Figure 20). If in GFT a return value is attached to the return statement then in MSC this value has to be assigned within the MSC reference definition to a return variable owned by the calling instance.



GFT **MSC**

Figure 20: GFT-MSC: Return statement with a return value

F.7 Default Handling

The GFT default handling is represented in MSC in an expanded form. After activation (before deactivation) the default behaviour has to be added to all alternatives or to single receive events. In detail, a set of transformation rules is demanded.

F.8 Configuration operations

F.8.1 The Create operation

The GFT create operation is mapped in MSC into a create message sent to the environment with the component name and component type attached as parameter list to the create arrow (see Figure 21).

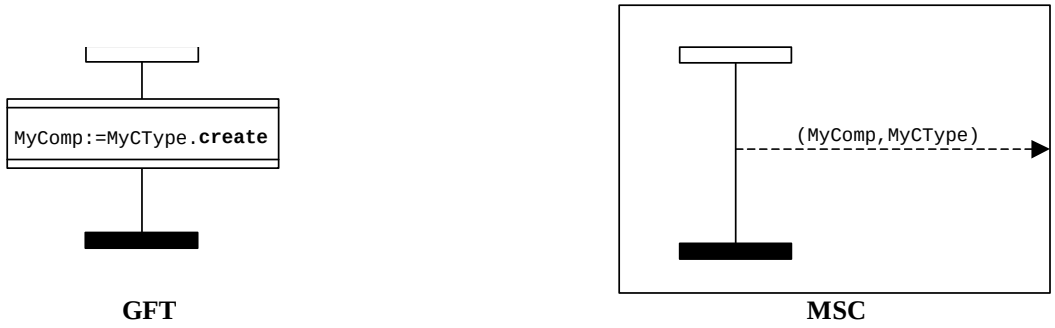


Figure 21: GFT-MSC: Create operation

F.8.2 The Connect (Disconnect) and Map (Unmap) operations

Connect (disconnect) and map (unmap) operations which in GFT are represented within an action box are mapped to several action boxes in MSC in form of informal text whereby sequential statements separated by semicolons are mapped to sequences of action boxes (see Figure 22).

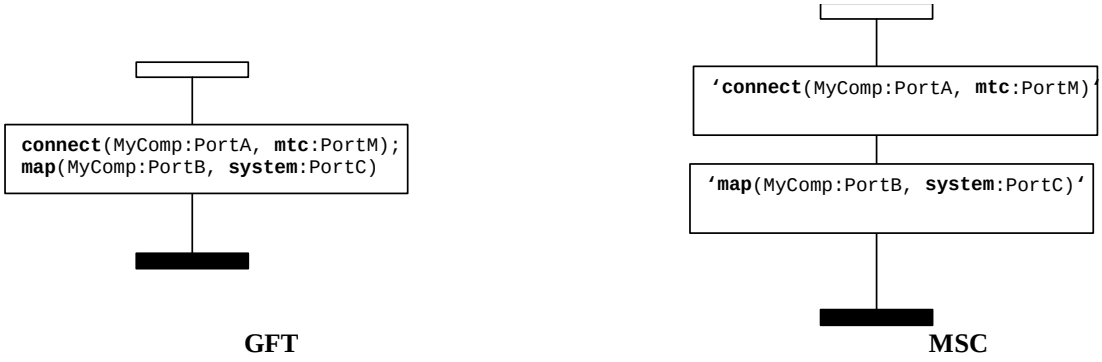
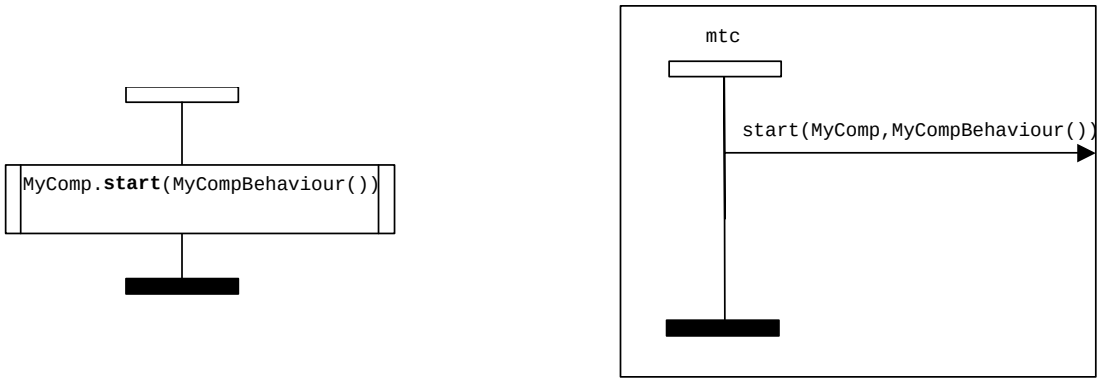


Figure 22: GFT-MSC: Connect and map operation

F.8.3 The Start test component operation

The GFT start test component operation is mapped in MSC into a start message sent to the environment with the message name "start" followed by a parameter list containing the component name and the test case name (see Figure 23).



GFT **MSC**

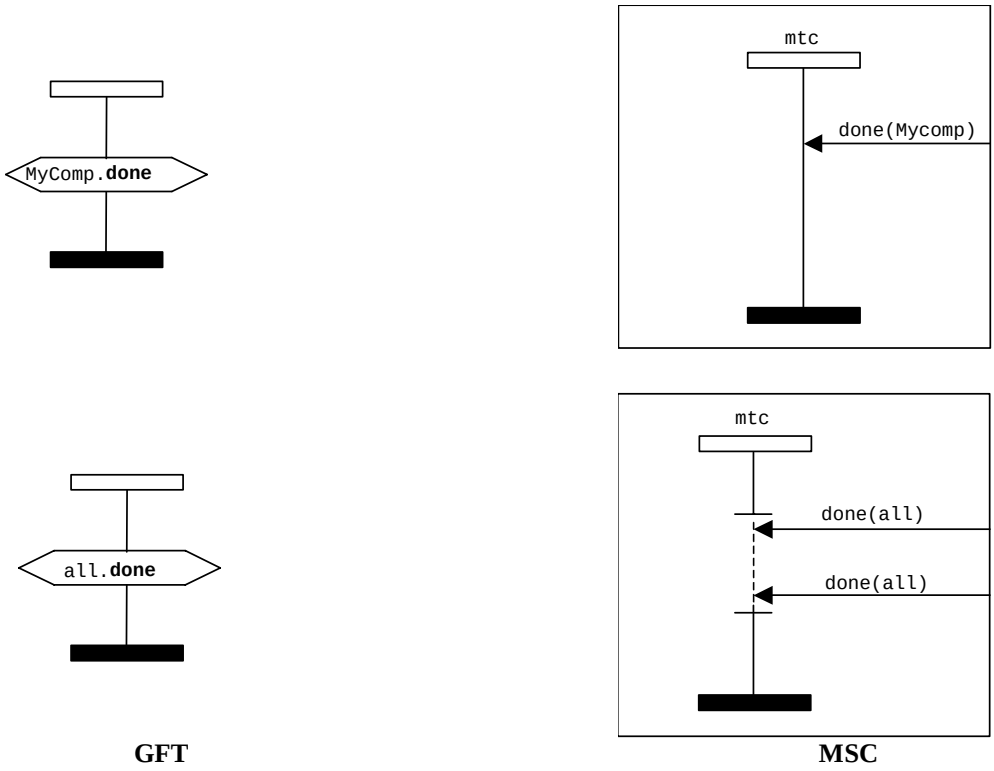
Figure 23: GFT-MSC: Start operation

F.8.4 The Stop execution and stop test component operation

The GFT stop execution operation is mapped in MSC into an instance stop. The semantics of the GFT stop execution operation differs from the standard MSC semantics: When the MTC terminates also all other test components have to terminate.The GFT stop test component operation is mapped in MSC into an action box with the corresponding statement for the stop test component operation.

F.8.5 The Done operation

The GFT done operation is mapped in MSC into a message sent from the environment to the test component that is performing the done operation with message name “done” and a parameter list containing the name of the stopped test component or in case of any.done or all.done, to a set of messages sent from the environment to the test component that is performing the done operation with message name “done” and a parameter list containing any or all (see Figure 24).



GFT **MSC**

Figure 24: GFT-MSC: Done operation

F.9 Communication operations

F.9.1 General format of the sending operations

The GFT sending operation using a message symbol, which is drawn from the component instance to the port instance, is mapped into the same message symbol in MSC. The GFT port instance is mapped into a standard MSC instance. The message name is send, call, reply or raise followed by a parameter list containing message type or signature and template.

F.9.2 General format of the receiving operations

The GFT receiving operation using a message symbol, which is drawn from the port instance to the component instance, is mapped into the same message symbol in MSC. The GFT port instance is mapped into a standard MSC instance. The message name is receive, getcall, getreply and catch followed by a parameter list containing message type or signature and template.

F.9.3 Message based communication

F.9.3.1 Send and receive operations

The GFT send and receive operations are mapped into corresponding messages in MSC with "send" and "receive" as message names followed by a parameter list containing the message type and template (see Figure 25 and Figure 26). If the type information were not present the parameter list would contain an empty parameter, which is omitted in this document for simplicity. As an extension, messages without any name are allowed. Receive on any port is mapped onto a found message (using the same graphical symbol).

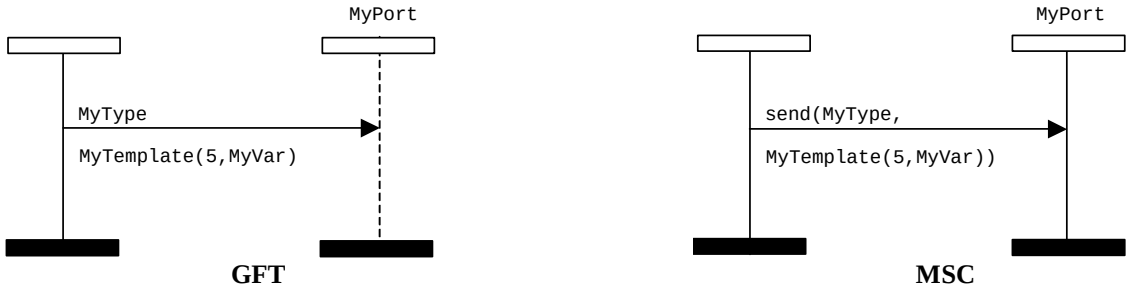


Figure 25: GFT-MSC: Send operation

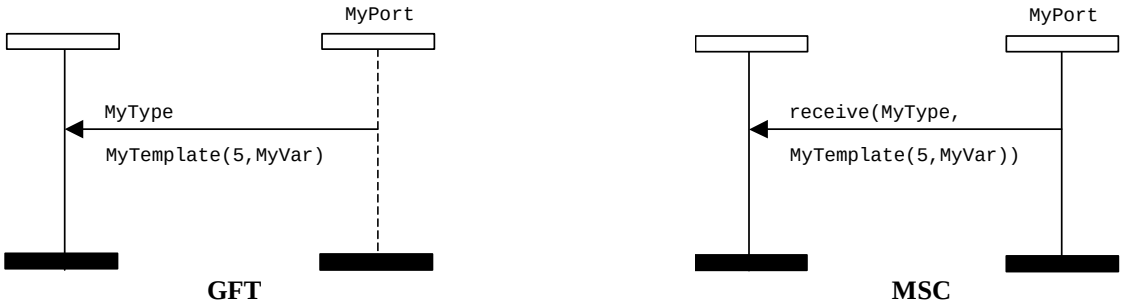


Figure 26: GFT-MSC: Receive operation

F.9.3.2 The Trigger operation

The GFT trigger operation in MSC is represented by a guarded loop containing an alternative between the reception of the matching message and the rest (see Figure 27). The guarding variable of the loop that is initially true is set false if the matching message (represented by a message with the name "receive" followed by a parameter list containing the type information and template) is received, otherwise the non-matching messages (in form of any receive messages) are sent to a black hole. The priorities for the alternatives are handled as described above for the alternative behaviour.

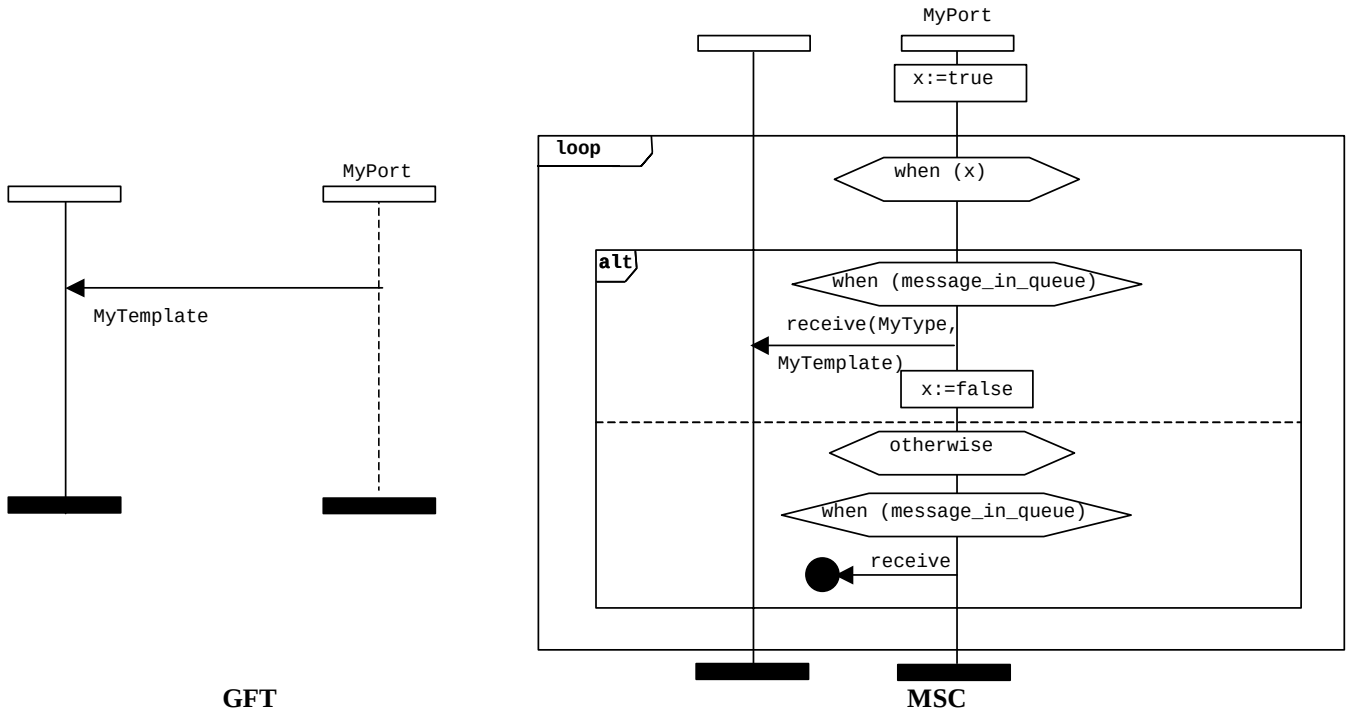


Figure 27: GFT-MSC: Trigger operation

F.9.4 Procedure-based communication

F.9.4.1 The Call operation

F.9.4.1.1 Calling blocking procedures

The GFT blocking call operations are mapped into control flow messages in MSC.

The GFT blocking call is mapped into a method call in MSC with the message name “call” followed by a parameter list containing the signature and template (see Figure 28). No method region is attached to the port instance since the port instance describes only the interface.

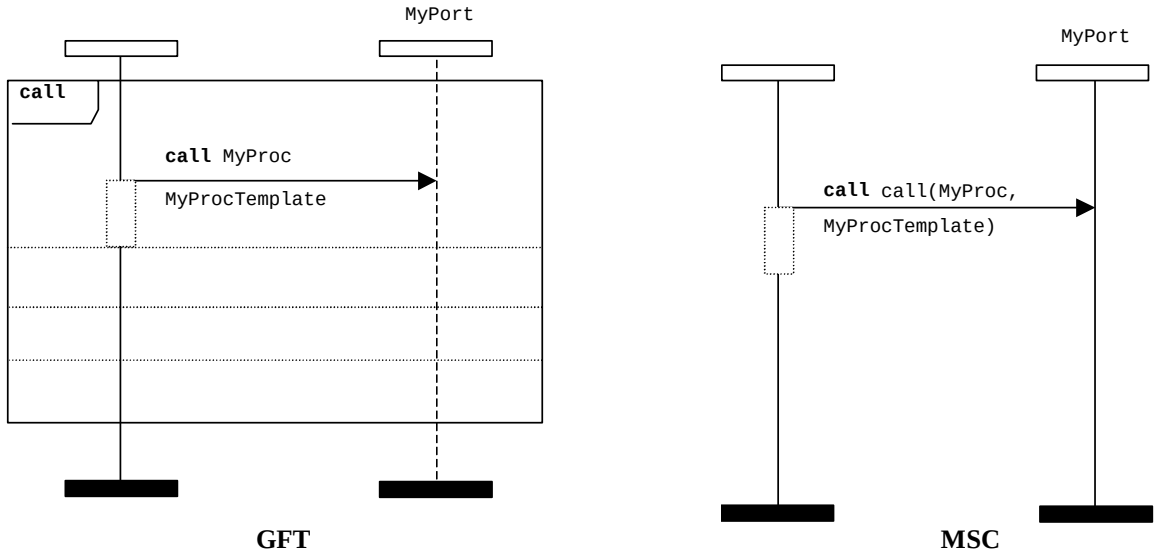


Figure 28: GFT-MSC: Blocking call operation

The call inline expression in GFT is mapped in MSC into an alternative inline expression containing the alternatives of the possible responses (see Figure 29). The response messages (getreply, catch) in MSC are represented by dashed

message arrows with the message names getreply and catch. The splitting of suspension regions used in the alternative inline expression is not allowed in MSC-2000, which is a deficiency of the MSC language to be removed.

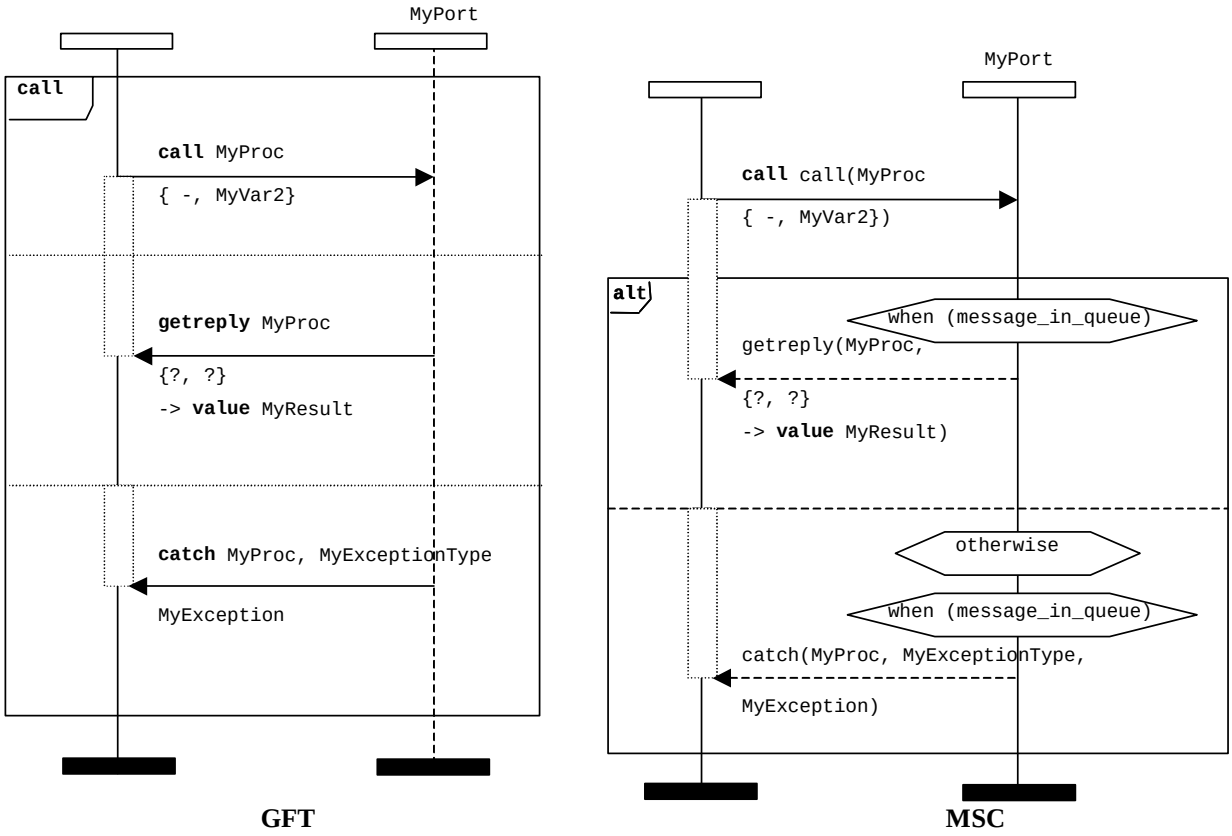


Figure 29: GFT-MSC: Blocking call operation followed by alternatives of getreply and catch

F.9.4.1.2 Calling non-blocking procedures

The call of GFT non-blocking procedures in MSC is mapped into call messages (with the keyword call) without suspension and method symbols and with the message name "call" followed by a parameter list containing the signature and template (see Figure 30).

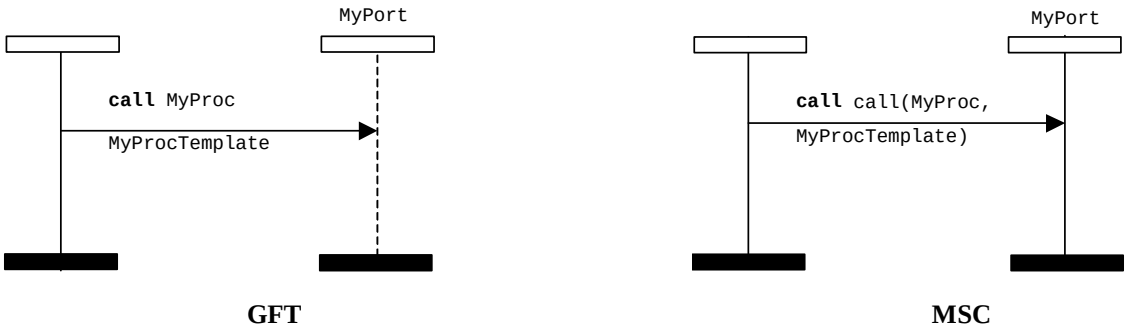


Figure 30: GFT-MSC: Non-blocking call operation

F.9.4.2 The Getcall operation

The GFT getcall operation is mapped into a method call in MSC from the port instance to the test component instance with the message name "getcall" followed by a parameter list containing the signature and template and with a method symbol on the test component and no suspension region on the port instance (see Figure 31).

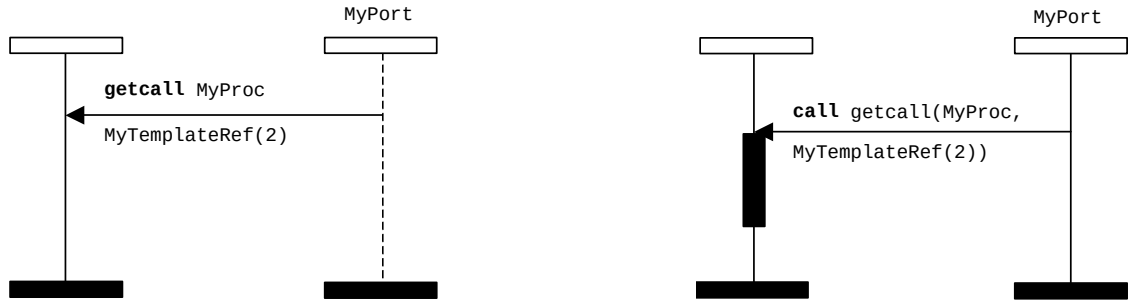


Figure 31: GFT-MSC: Getcall operation

In case of getcall on any call operation, a call message without the name getcall and no parameter list is used in MSC (see Figure 32).

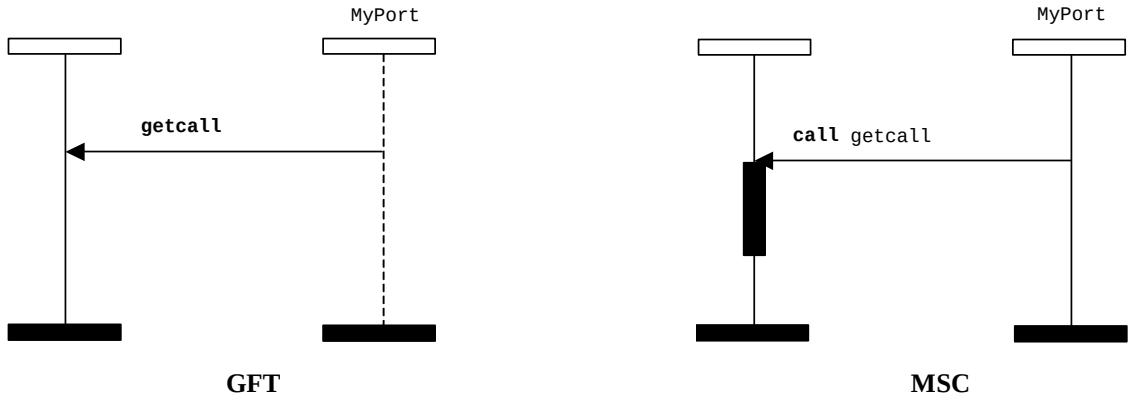


Figure 32: GFT-MSC: Getcall on any call operation

In case of getcall on any port, a found call message is used in MSC (see Figure 33).

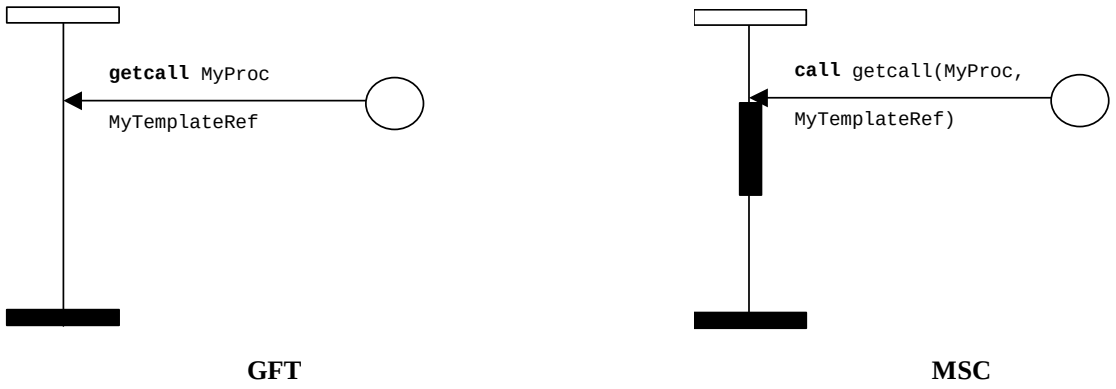


Figure 33: GFT-MSC: Getcall on any port operation

F.9.4.3 The Reply operation

The GFT reply operation is mapped in MSC to a dashed message arrow with the message name "reply" followed by a parameter list containing the signature, template and reply value and with a method region on the test component preceding the reply message (see Figure 34).

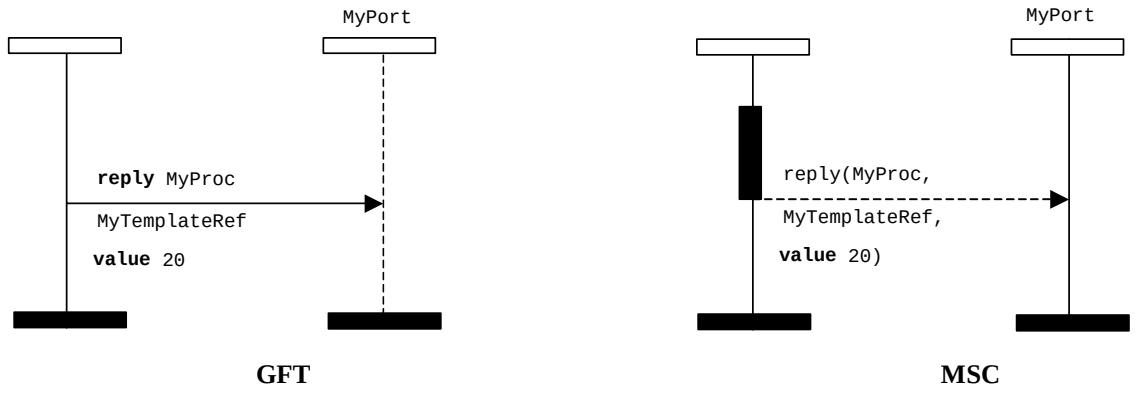


Figure 34: GFT-MSC: Reply operation

F.9.4.4 The Getreply operation

The GFT getreply operation within a call is represented in MSC by a dashed message arrow with the message name "getreply" followed by a parameter list containing the signature and template whereby the message arrowhead is attached to a preceding suspension region on the test component (see Figure 35).

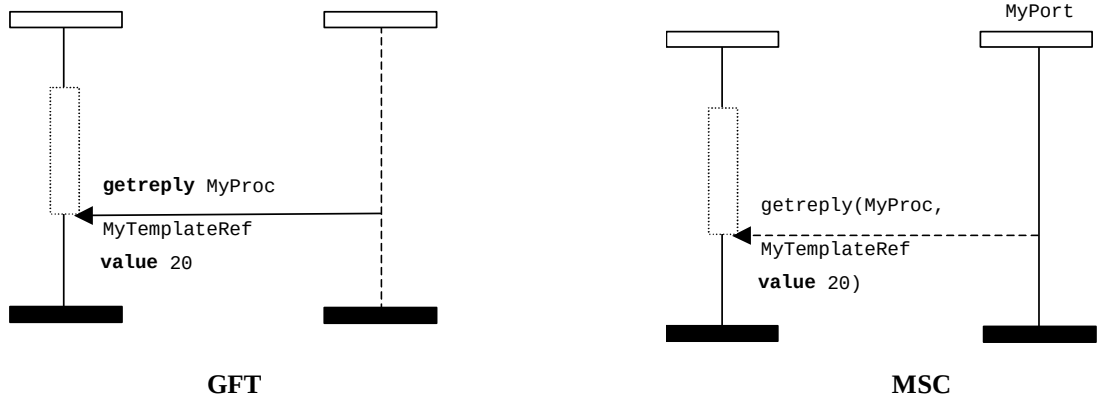


Figure 35: GFT-MSC: Getreply operation (within a call symbol)

In case of a GFT getreply operation outside the call, also in MSC the suspension region is omitted (see Figure 36).

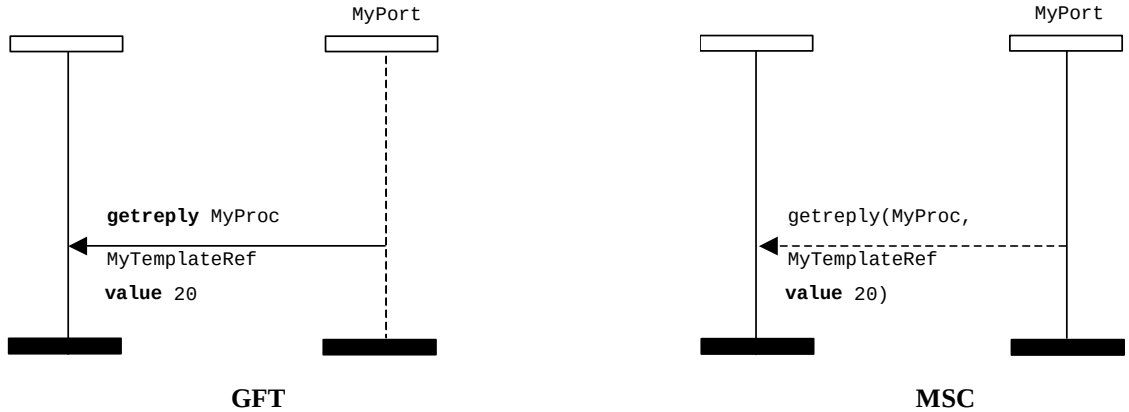


Figure 36: GFT-MSC: Getreply operation (outside a call symbol)

The GFT any reply from any call is represented in MSC by a dashed message arrow from the port instance to the component instance with the name getreply and no parameter list and the GFT get a reply on any port operation is represented in MSC by a dashed found message arrow.

F.9.4.5 The Raise operation

The GFT raise operation is represented in MSC by a dashed message arrow with the message name "raise" followed by a parameter list containing the signature, exception type and template and with a method region on the test component preceding the raise message (see Figure 37).

.

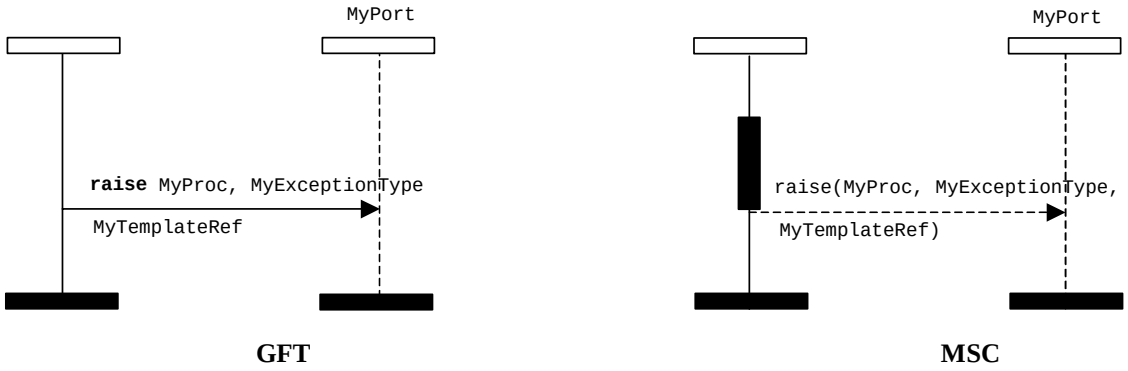


Figure 37: GFT-MSC: Raise operation

F.9.4.6 The Catch operation

The GFT catch operation within a call is represented in MSC by a dashed message arrow with the message name "catch" followed by a parameter list containing the signature and the exception type whereby the message arrowhead is attached to a preceding suspension region on the test component (see Figure 38).

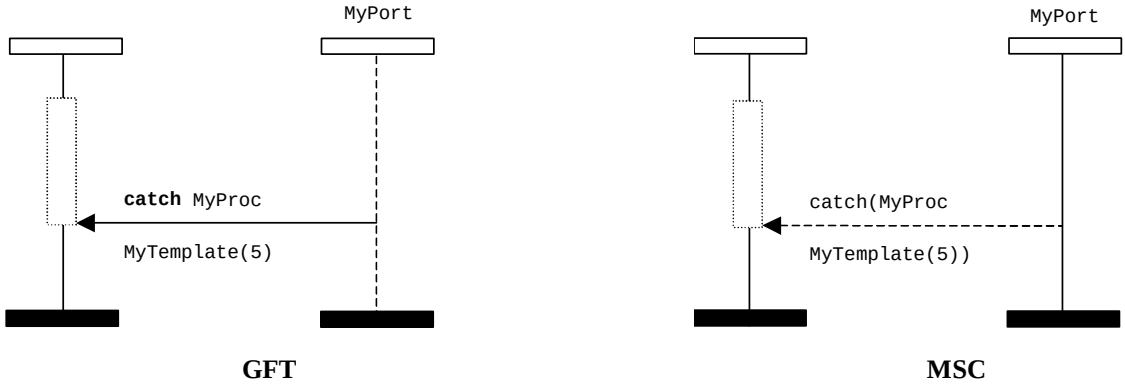


Figure 38: GFT-MSC: Catch operation (within a call symbol)

In case of a GFT catch operation outside the call, also in MSC the suspension region is omitted (see Figure 39).

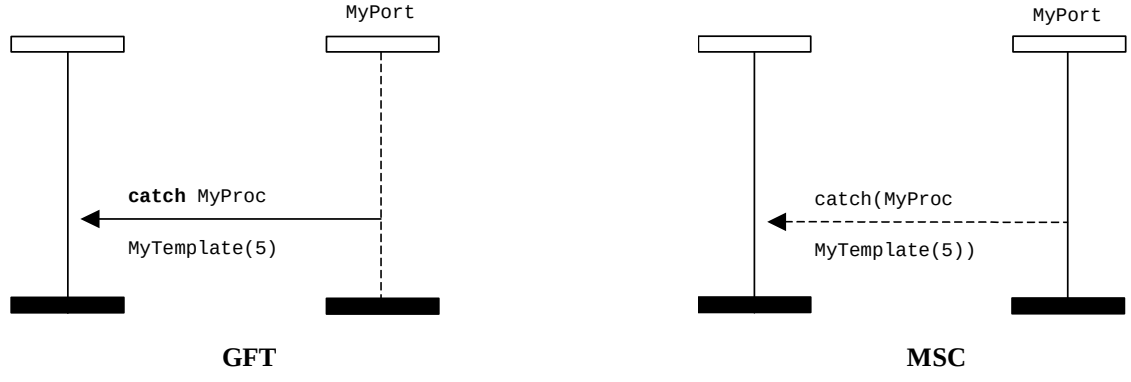


Figure 39: GFT-MSC: Catch operation (outside a call symbol)

The GFT catch any exception is represented in MSC by a dashed message arrow from the port instance to the component instance with the name catch and no parameter list and the GFT catch on any port operation is represented in MSC by a dashed found message arrow.

The GFT timeout exception for the call operation is represented in MSC by a separate timeout symbol without Timer name (see Figure 40). That is an extension to MSC. The GFT timeout exception inside a call is translated into a MSC timeout symbol with the message arrowhead attached to a preceding suspension region that is also an extension to MSC.

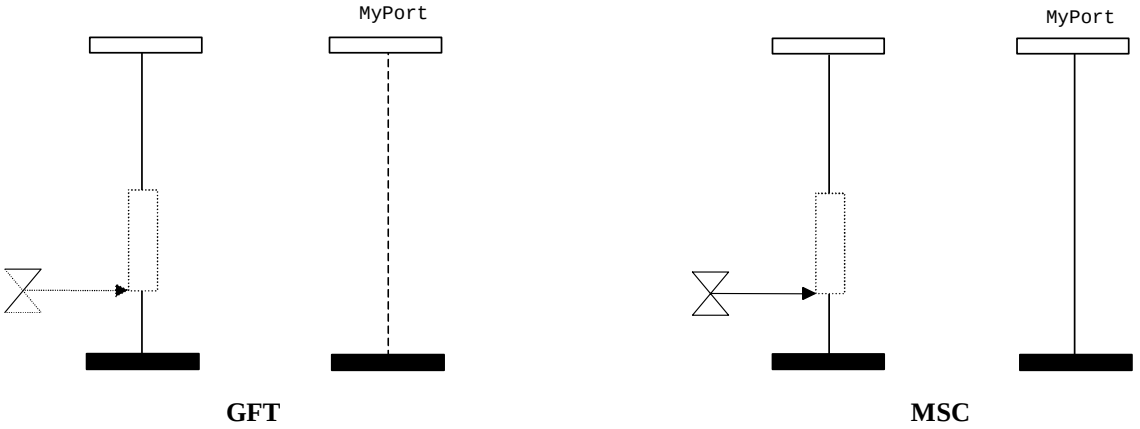


Figure 40: GFT-MSC: Timeout exception (within a call symbol)

F.9.5 The Check operation

The GFT check a receive operation is mapped in MSC into a message arrow from the port instance to the component instance with the message name "check receive" (see Figure 41).

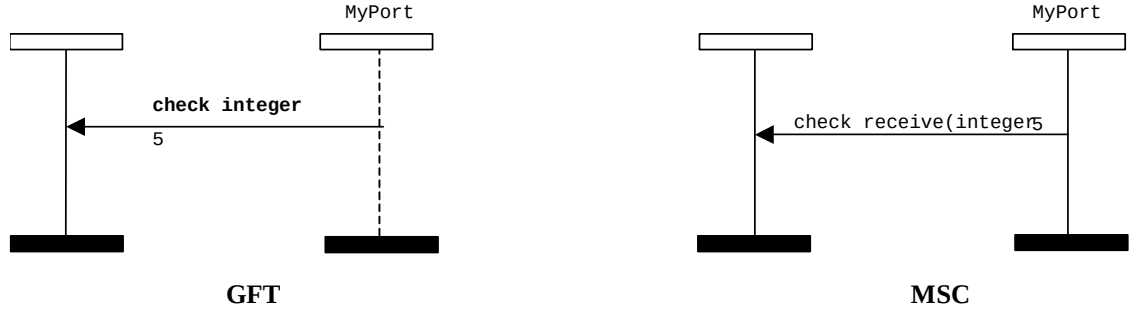


Figure 41: GFT-MSC: Check a receive

The GFT check a getcall operation is mapped in MSC into a message arrow from the port instance to the component instance with the message name "check getcall".

The GFT check a getreply operation is mapped in MSC into a dashed message arrow from the port instance to the component instance with the message name "check getreply" (see Figure 42).

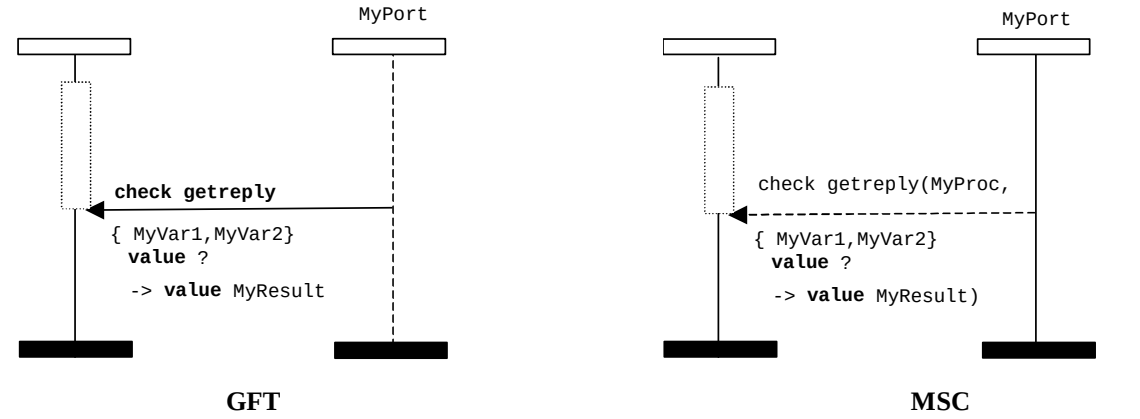


Figure 42: GFT-MSC: Check a getreply

The GFT check a catch operation is mapped in MSC into a dashed message arrow from the port instance to the component instance with the message name "check getreply".

The various check message names are followed by a parameter list containing the information attached to the specific check operation.

The GFT catch any is represented in MSC by a message arrow from the port instance to the component instance with the name check and no parameter list. The GFT catch on any port operation is represented in MSC by a dashed found message arrow.

F.10 Controlling communication ports

F.10.1 The Clear port operation

The clear operation is represented by a guarded loop of receive any messages which are sent to a black hole (see Figure 43). The guarding condition is true as long the port is not empty.

Figure 43: GFT-MSC:Clear port operation

F.10.2 The Start port operation

The GFT start port operation is represented in MSC by a message arrow from the test component instance to the port instance with the message name "start" (see Figure 44).

Figure 44: GFT-MSC: Start port operation

F.10.3 The Stop port operation

The GFT stop port operation is represented in MSC by a message arrow from the test component instance to the port instance with the message name "stop" (see Figure 45).

GFT **MSC**

Figure 45: GFT-MSC: Stop port operation

F.11 Timer operationsF.11.1 The Start timer operation

The GFT start timer operation is represented in MSC by the same start timer construct.

F.11.2 The Stop timer operation

The GFT stop timer operation is represented in MSC by the same stop timer construct.

F.11.3 The Timeout operation

The GFT timeout operation is represented in MSC by the same timeout construct.

F.11.4 The Read timer operation

The GFT read timer operation is represented also in MSC by an action box with the same text.

F.11.5 Use of any and all with timers

The GFT stop timer operation applied to all timers is mapped in MSC into a stop timer construct without timer name and the GFT timeout operation applied to any timer is mapped in MSC into a timeout construct without timer name. Both are extensions to MSC.

F.12 Setverdict operation

The GFT verdict **set** operation with a condition symbol within which the values **`pass`**, **`fail`**, **`inconc`** or **`none`** are denoted is mapped in MSC into a corresponding setting condition interpreting pass, fail, inconc and none as states.

F.13 External action

The GFT external action within action box symbols is mapped into MSC actions with the same the external action syntax inside. Since MSC actions may only contain informal text or data expressions/assignments, as an extension to MSC also general statements have to be admitted.

F.14 Specifying attributes

The GFT attributes represented within text symbols are mapped into MSC text symbols containing the same syntax description. As an extension to MSC, text symbols may contain statements in addition to comment text.

History

Document history		
V1.1.1	January 2001	Publication
V2.2.0	March 2002	Revision distributed to MTS for approval at MTS#34.