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

Intellectual Property Rights .................................................................................................................6
Foreword ..................................................................................................................................................6
Introduction ............................................................................................................................................6
1  Scope ..................................................................................................................................................8
2  References .........................................................................................................................................8
3  Abbreviations ......................................................................................................................................8
4  Overview ............................................................................................................................................9
5  GFT language concepts ....................................................................................................................10
6  Mapping between GFT and TTCN-3 Core language .........................................................................12
7  Module structure ...............................................................................................................................12
8  GFT symbols ....................................................................................................................................14
9  GFT diagrams ..................................................................................................................................17
  9.1 Common properties ......................................................................................................................17
  9.1.1 Diagram area ............................................................................................................................17
  9.1.2 Diagram heading ....................................................................................................................17
  9.1.3 Paging ......................................................................................................................................17
  9.2 Control diagram ............................................................................................................................18
  9.3 Test case diagram .........................................................................................................................18
  9.4 Function diagram ........................................................................................................................19
  9.5 Altstep diagram ...........................................................................................................................20
10 Instances in GFT diagrams .............................................................................................................21
  10.1 Control instance ........................................................................................................................21
  10.2 Test component instances .........................................................................................................21
  10.3 Port instances ............................................................................................................................22
11 Elements of GFT diagrams .............................................................................................................22
  11.1 General drawing rules ...............................................................................................................22
  11.1.1 Usage of semicolons .............................................................................................................22
  11.1.2 Usage of action symbols ......................................................................................................23
  11.1.3 Comments ............................................................................................................................23
  11.2 Invoking GFT diagrams .............................................................................................................24
  11.2.1 Execution of test cases .........................................................................................................24
  11.2.2 Invocation of functions ..........................................................................................................24
  11.2.3 Invocation of altsteps ...........................................................................................................25
  11.3 Declarations ..............................................................................................................................25
  11.3.1 Declaration of timers, constants and variables in action symbols ...........................................26
  11.3.2 Declaration of constants and variables within inline expression symbols ............................26
  11.3.3 Declaration of constants and variables within create symbols ...........................................26
  11.3.4 Declaration of constants and variables within default declarations .....................................26
  11.3.5 Declaration of constants and variables within reference symbols ......................................27
  11.3.6 Declaration of constants and variables within execute test case symbols ............................27
  11.4 Basic program statements .........................................................................................................27
  11.4.1 The Log statement .................................................................................................................28
  11.4.2 The Label statement ..............................................................................................................28
  11.4.3 The Goto statement ..............................................................................................................28
  11.4.4 The If-else statement ............................................................................................................29
  11.4.5 The For statement ................................................................................................................29
  11.4.6 The While statement .............................................................................................................30
  11.4.7 The Do-while statement .......................................................................................................30
11.5 Behavioural Program Statements .................................................................................................30
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1</td>
<td>11.11 External actions</td>
<td>59</td>
</tr>
<tr>
<td>11.10</td>
<td>Test verdict operations</td>
<td>58</td>
</tr>
<tr>
<td>11.9</td>
<td>11.9.5 Use of any and all with timers</td>
<td>58</td>
</tr>
<tr>
<td>11.8</td>
<td>11.8.6.4 Use of any and all with ports</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>11.8.6.2 The Start port operation</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>11.8.5.1 The Check any operation</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>11.8.4.6.2 Catch any exception</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>11.8.4.6.1 The Timeout exception</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>11.8.4.5 The Raise operation</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>11.8.4.4.2 Get a reply on any port</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>11.8.4.2.2 Getcall on any port</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>11.8.4.2 The Getcall operation</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>11.8.4.1.2 Calling blocking procedures</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>11.8.4.1.1 Calling non-blocking procedures</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>11.8.4.4.1 Get any reply from any call</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>11.8.4.4.2 Get a reply on any port</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>11.8.4.4 The Getreply operation</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>11.8.4.3 The Reply operation</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>11.8.4.2.1 Accepting any call</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>11.8.4.2 The Getcall operation</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>11.8.4.2.2 Getcall on any port</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>11.8.4.3 The Trigger operation</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>11.8.3.3.1 Trigger on any message</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>11.8.3.3 The Trigger operation</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>11.8.3.2.2 Receive on any port</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>11.8.3.2 The Receive operation</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>11.8.3.2.1 Receive any message</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>11.8.3.1 The Send operation</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>11.8.4 Procedure-based communication</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>11.8.4.1 The Call operation</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>11.8.4.1.2 Calling non-blocking procedures</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>11.8.4.1.1 Calling blocking procedures</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>11.8.4.2 The Getcall operation</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>11.8.4.3 The Reply operation</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>11.8.4.4 The Getreply operation</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>11.8.4.5 The Raise operation</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>11.8.4.6 The Catch operation</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>11.8.4.6.1 The Timeout exception</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>11.8.4.6.2 Catch any exception</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>11.8.4.6.3 Catch on any port</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>11.8.5 The Check operation</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>11.8.5.1 The Check any operation</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>11.8.5.2 Check on any port</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>11.8.6 Controlling communication ports</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>11.8.6.1 The Clear port operation</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>11.8.6.2 The Start port operation</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>11.8.6.3 The Stop port operation</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>11.8.6.4 Use of any and all with ports</td>
<td>56</td>
</tr>
<tr>
<td>11.7</td>
<td>Interleaved Behaviour</td>
<td>34</td>
</tr>
<tr>
<td>11.6</td>
<td>Default handling</td>
<td>35</td>
</tr>
<tr>
<td>11.5</td>
<td>Selecting/Deselecting an Alternative</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>Else branch in alternatives</td>
<td>33</td>
</tr>
<tr>
<td>11.4</td>
<td>The Repeat statement</td>
<td>33</td>
</tr>
<tr>
<td>11.3</td>
<td>The Start timer operation</td>
<td>34</td>
</tr>
<tr>
<td>11.2</td>
<td>The Return statement</td>
<td>34</td>
</tr>
<tr>
<td>11.1</td>
<td>Default references</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>The activate operation</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>The deactivate operation</td>
<td>35</td>
</tr>
<tr>
<td>11.8</td>
<td>Configuration operations</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>The Create operation</td>
<td>36</td>
</tr>
<tr>
<td>11.7</td>
<td>The Disconnect and Unmap operations</td>
<td>36</td>
</tr>
<tr>
<td>11.6</td>
<td>11.6.3 The Repeat statement</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>11.6.2 The deactivate operation</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>11.6.1 Default references</td>
<td>35</td>
</tr>
<tr>
<td>11.5</td>
<td>Interleaved Behaviour</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>The Repeat statement</td>
<td>33</td>
</tr>
<tr>
<td>11.4</td>
<td>The Start timer operation</td>
<td>34</td>
</tr>
<tr>
<td>11.3</td>
<td>The Return statement</td>
<td>34</td>
</tr>
<tr>
<td>11.2</td>
<td>The Return statement</td>
<td>34</td>
</tr>
<tr>
<td>11.1</td>
<td>Default references</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>The activate operation</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>The deactivate operation</td>
<td>35</td>
</tr>
<tr>
<td>11.8</td>
<td>Configuration operations</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>The Create operation</td>
<td>36</td>
</tr>
<tr>
<td>11.7</td>
<td>The Disconnect and Unmap operations</td>
<td>36</td>
</tr>
<tr>
<td>11.6</td>
<td>Default references</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>The activate operation</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>The deactivate operation</td>
<td>35</td>
</tr>
<tr>
<td>11.5</td>
<td>Interleaved Behaviour</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>The Repeat statement</td>
<td>33</td>
</tr>
<tr>
<td>11.4</td>
<td>The Start timer operation</td>
<td>34</td>
</tr>
<tr>
<td>11.3</td>
<td>The Return statement</td>
<td>34</td>
</tr>
<tr>
<td>11.2</td>
<td>The Return statement</td>
<td>34</td>
</tr>
<tr>
<td>11.1</td>
<td>Default references</td>
<td>35</td>
</tr>
<tr>
<td>11.8</td>
<td>Communication operations</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>General format of the sending operations</td>
<td>38</td>
</tr>
<tr>
<td>11.7</td>
<td>Configuration operations</td>
<td>35</td>
</tr>
<tr>
<td>11.6</td>
<td>Default references</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>The activate operation</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>The deactivate operation</td>
<td>35</td>
</tr>
<tr>
<td>11.5</td>
<td>Interleaved Behaviour</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>The Repeat statement</td>
<td>33</td>
</tr>
<tr>
<td>11.4</td>
<td>The Start timer operation</td>
<td>34</td>
</tr>
<tr>
<td>11.3</td>
<td>The Return statement</td>
<td>34</td>
</tr>
<tr>
<td>11.2</td>
<td>The Return statement</td>
<td>34</td>
</tr>
<tr>
<td>11.1</td>
<td>Default references</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>The activate operation</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>The deactivate operation</td>
<td>35</td>
</tr>
<tr>
<td>11.8</td>
<td>Communication operations</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>General format of the receiving operations</td>
<td>39</td>
</tr>
<tr>
<td>11.7</td>
<td>Configuration operations</td>
<td>35</td>
</tr>
<tr>
<td>11.6</td>
<td>Default references</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>The activate operation</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>The deactivate operation</td>
<td>35</td>
</tr>
<tr>
<td>11.5</td>
<td>Interleaved Behaviour</td>
<td>34</td>
</tr>
<tr>
<td>11.4</td>
<td>The Repeat statement</td>
<td>33</td>
</tr>
<tr>
<td>11.3</td>
<td>The Start timer operation</td>
<td>34</td>
</tr>
<tr>
<td>11.2</td>
<td>The Return statement</td>
<td>34</td>
</tr>
<tr>
<td>11.1</td>
<td>Default references</td>
<td>35</td>
</tr>
<tr>
<td>11.8</td>
<td>Communication operations</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>General format of the sending operations</td>
<td>38</td>
</tr>
<tr>
<td>11.7</td>
<td>Configuration operations</td>
<td>35</td>
</tr>
<tr>
<td>11.6</td>
<td>Default references</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>The activate operation</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>The deactivate operation</td>
<td>35</td>
</tr>
<tr>
<td>11.5</td>
<td>Interleaved Behaviour</td>
<td>34</td>
</tr>
<tr>
<td>11.4</td>
<td>The Repeat statement</td>
<td>33</td>
</tr>
<tr>
<td>11.3</td>
<td>The Start timer operation</td>
<td>34</td>
</tr>
<tr>
<td>11.2</td>
<td>The Return statement</td>
<td>34</td>
</tr>
<tr>
<td>11.1</td>
<td>Default references</td>
<td>35</td>
</tr>
</tbody>
</table>
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://webapp.etsi.org/IPR/home.asp).

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).

The present document is part 3 of a multi-part deliverable. Full details of the entire series can be found in part 1 [1].

Introduction

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

The core language of TTCN-3 is defined in ES 201 873-1 [1] and provides a full text-based syntax, static semantics and operational semantics as well as a definition for the use of the language with ASN.1. The GFT presentation format provides an alternative way of displaying the core language (see figure 1).

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

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

Together, these characteristics form GFT—the graphical presentation format of TTCN-3.
1 Scope

The present document defines the graphical 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 ITU-T Recommendation Z.120 [3] with test specific extensions.

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

The following documents contain provisions which, through reference in this text, constitute provisions of the present document.

- References are either specific (identified by date of publication and/or edition number or version number) or non-specific.
- For a specific reference, subsequent revisions do not apply.
- For a non-specific reference, the latest version applies.

Referenced documents which are not found to be publicly available in the expected location might be found at http://docbox.etsi.org/Reference.

NOTE: While any hyperlinks included in this clause were valid at the time of publication ETSI cannot guarantee their long term validity.


3 Abbreviations

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

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNF</td>
<td>Backus-Naur Form</td>
</tr>
<tr>
<td>CATG</td>
<td>Computer Aided Test Generation</td>
</tr>
<tr>
<td>GFT</td>
<td>Graphical presentation Format of TTCN-3</td>
</tr>
<tr>
<td>MSC</td>
<td>Message Sequence Chart</td>
</tr>
<tr>
<td>MTC</td>
<td>Main Test Component</td>
</tr>
<tr>
<td>PTC</td>
<td>Parallel Test Component</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
</tr>
<tr>
<td>TFT</td>
<td>Tabular presentation Format of TTCN-3</td>
</tr>
<tr>
<td>TTCN</td>
<td>Testing and Test Control Notation</td>
</tr>
</tbody>
</table>
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 interaction with the SUT and does not describe the actions of the test components necessary to validate the SUT behaviour, i.e. it is a test purpose description.

![MSC TestPurposeExample](image)

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

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

**NOTE:** The examples in figures 2 and 3 are only simple examples to illustrate the different usages of MSC for testing. The diagrams will be more complicated in case of a distributed SUT that consists of several processes or a distributed test configuration with several test components.
In identifying these two categories of MSC usage two distinct areas of work can be identified (see figure 4):

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

b) *Development of a Graphical presentation format for TTCN-3 (GFT) and definition of the mapping between GFT and TTCN-3.*

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

## 5 GFT language concepts

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

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

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

GFT defines no graphical representation for:

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

TTCN-3 definitions and declarations without a corresponding GFT presentation may be presented in the TTCN-3 core language or in the tabular presentation format for TTCN-3 (TFT) (ES 201 873-2 [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 (ITU-T Recommendation Z.120 [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:
• the representation of port instances;
• the creation of test components;
• the start of test components;
• the return from a function call;
• the repetition of alternatives;
• the time supervision of a procedure-based call;
• the execution of test cases;
• the activation and deactivation of defaults;
• the labelling and goto; and
• the timers within call statements.

A complete list of all symbols used in GFT is presented in clause 8.
Mapping between GFT and TTCN-3 Core language

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

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

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

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

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

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

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

Figure 6: Structure of TTCN-3 modules

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

types

datatype MyType

component MyCompType

functions

function MySpecialFunction

altsteps

altstep MyAltStep

testcases

testcase MyTestCase

control

class control

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

8 GFT symbols

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

<table>
<thead>
<tr>
<th>GFT Element</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame symbol</td>
<td></td>
<td>Used to frame GFT diagrams</td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Frame symbol" /></td>
<td></td>
</tr>
<tr>
<td>Reference symbol</td>
<td></td>
<td>Used to represent the invocation of functions and altsteps</td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Reference symbol" /></td>
<td></td>
</tr>
<tr>
<td>GFT Element</td>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Port instance symbol</td>
<td><img src="image" alt="Port instance symbol" /></td>
<td>Used to represent port instances</td>
</tr>
<tr>
<td>Component instance symbol</td>
<td><img src="image" alt="Component instance symbol" /></td>
<td>Used to represent test components and the control instance</td>
</tr>
<tr>
<td>Action box symbol</td>
<td><img src="image" alt="Action box symbol" /></td>
<td>Used for textual TTCN-3 declarations and statements, to be attached to a component symbol</td>
</tr>
<tr>
<td>Condition symbol</td>
<td><img src="image" alt="Condition symbol" /></td>
<td>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</td>
</tr>
<tr>
<td>Labelling symbol</td>
<td><img src="image" alt="Labelling symbol" /></td>
<td>Used for TTCN-3 labelling and goto, to be attached to a component symbol</td>
</tr>
<tr>
<td>Goto symbol</td>
<td><img src="image" alt="Goto symbol" /></td>
<td>Used for TTCN-3 labelling and goto, to be attached to a component symbol</td>
</tr>
<tr>
<td>Inline expression symbol</td>
<td><img src="image" alt="Inline expression symbol" /></td>
<td>Used for TTCN-3 if-else, for, while, do-while, alt, call and interleave statement, to be attached to a component symbol</td>
</tr>
<tr>
<td>Default symbol</td>
<td><img src="image" alt="Default symbol" /></td>
<td>Used for TTCN-3 activate and deactivate statement, to be attached to a component symbol</td>
</tr>
<tr>
<td>Stop symbol</td>
<td><img src="image" alt="Stop symbol" /></td>
<td>Used for TTCN-3 stop statement, to be attached to a component symbol</td>
</tr>
<tr>
<td>Return symbol</td>
<td><img src="image" alt="Return symbol" /></td>
<td>Used for TTCN-3 return statement, to be attached to a component symbol</td>
</tr>
<tr>
<td>Repeat symbol</td>
<td><img src="image" alt="Repeat symbol" /></td>
<td>Used for TTCN-3 repeat statement, to be attached to a component symbol</td>
</tr>
<tr>
<td>GFT Element</td>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Create symbol</td>
<td><img src="image" alt="Create symbol" /></td>
<td>Used for TTCN-3 create statement, to be attached to a component symbol</td>
</tr>
<tr>
<td>Start symbol</td>
<td><img src="image" alt="Start symbol" /></td>
<td>Used for TTCN-3 start statement, to be attached to a component symbol</td>
</tr>
<tr>
<td>Message symbol</td>
<td><img src="image" alt="Message symbol" /></td>
<td>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</td>
</tr>
<tr>
<td>Found symbol</td>
<td><img src="image" alt="Found symbol" /></td>
<td>Used for representing TTCN-3 receive, getcall, getreply, catch, trigger and check from any port, to be attached to a component symbol</td>
</tr>
<tr>
<td>Suspension region symbol</td>
<td><img src="image" alt="Suspension region symbol" /></td>
<td>Used in combination with a blocking call, to be within a call inline expression and attached to a component symbol</td>
</tr>
<tr>
<td>Start timer symbol</td>
<td><img src="image" alt="Start timer symbol" /></td>
<td>Used for TTCN-3 start timer operation, to be attached to a component symbol</td>
</tr>
<tr>
<td>Timeout timer symbol</td>
<td><img src="image" alt="Timeout timer symbol" /></td>
<td>Used for TTCN-3 timeout operation, to be attached to a component symbol</td>
</tr>
<tr>
<td>Stop timer symbol</td>
<td><img src="image" alt="Stop timer symbol" /></td>
<td>Used for TTCN-3 stop timer operation, to be attached to a component symbol</td>
</tr>
<tr>
<td>Start implicit timer symbol</td>
<td><img src="image" alt="Start implicit timer symbol" /></td>
<td>Used for TTCN-3 implicit timer start in blocking call, to be within a call inline expression and attached to a component symbol</td>
</tr>
<tr>
<td>Timeout implicit timer symbol</td>
<td><img src="image" alt="Timeout implicit timer symbol" /></td>
<td>Used for TTCN-3 timeout exception in blocking call, to be within a call inline expression and attached to a component symbol</td>
</tr>
<tr>
<td>Execute symbol</td>
<td><img src="image" alt="Execute symbol" /></td>
<td>Used for TTCN-3 execute test case statement, to be attached to a component instance symbol</td>
</tr>
<tr>
<td>Text symbol</td>
<td><img src="image" alt="Text symbol" /></td>
<td>Used for TTCN-3 with statement and comments, to be placed within a GFT diagram</td>
</tr>
<tr>
<td>Event comment symbol</td>
<td><img src="image" alt="Event comment symbol" /></td>
<td>Used for TTCN-3 comments associated to events, to be attached to events on component instance or port instance symbols</td>
</tr>
</tbody>
</table>
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 is therefore outside the scope of the present document. A simple numbering scheme may only assign a page number, whereas an advanced numbering scheme may support the reconstruction of a diagram only by using the numbering information on the different pages.

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

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

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

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

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

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

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

9.3 Test case diagram

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

A GFT test case diagram shall include one test component instance describing the behaviour of the `mtc` (also called mtc instance) and one port instance for each port owned by the `mtc`. The name associated with the mtc instance shall be `mhc`. 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 `mhc` 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.

![GFT Core Diagrams](image)

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

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

### 9.4 Function diagram

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

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

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

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

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

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

Attributes associated to the function presented in GFT shall be specified within a text symbol in the function diagram. The principle shape of a GFT function diagram and the corresponding TTCN-3 core description are sketched in figure 11.
A function is used to specify and structure test behaviour, define default behaviour or to structure computation in a module. A function may contain declarations, statements, communication and timer operations and invocation of function or altsteps and an optional return statement.

### 9.5 Altstep diagram

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

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

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

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

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

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

Attributes associated to the altstep shall be specified within a text symbol in the GFT altstep diagram. The principle shape of a GFT altstep diagram and the corresponding TTCN-3 core language are sketched in figure 12.
An altstep is used to specify default behaviour or to structure the alternatives of an alt statement. An altstep may contain statements, communication and timer operations and invocation of function or altsteps.

10 Instances in GFT diagrams

GFT diagrams include the following kinds of instances:

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

10.1 Control instance

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

10.2 Test component instances

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

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

The optional test component type may be provided within the instance head symbol. It has to be consistent with the test component type given after the runs on keyword in the heading of the GFT diagram.
The principle shape of a test component instance in a test case diagram is shown in figure 13 b). The principle shape of a test component instance in a function or altstep diagram is shown in figure 13 c).

10.3 Port instances

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

11 Elements of GFT diagrams

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

11.1 General drawing rules

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

11.1.1 Usage of semicolons

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

The semicolon is optional if a GFT symbol includes only one statement in TTCN-3 core language (see figure 14 a) and figure 14 b)).
Semicolons shall separate the statements in a sequence of statements within an action symbol. The semicolon is optional for the last statement in the sequence (figure 14 c)).

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

### 11.1.2 Usage of action symbols

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

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

### 11.1.3 Comments

GFT provide three possibilities to put comments into GFT diagrams:

- Comments may be put into GFT symbols following the symbol inscription and using the syntax for comments of the TTCN-3 core language (figure 14 d)).

- Comments in the syntax for comments of the TTCN-3 core language can be put into text symbols and freely placed in the GFT diagram area (figure 14 e)).

- The comment symbol can be used to associate comments to GFT symbols. A comment in a comment symbol can be provided in form of free text, i.e. the comment delimiter "/*", "*/" and "/" of the core language need not to be used (figure 14 f)).

![Figure 14: Examples for the effects of the general drawing rules](image-url)
11.2 Invoking GFT diagrams

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

11.2.1 Execution of test cases

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

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

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

Figure 15: Test case execution

11.2.2 Invocation of functions

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

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

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

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

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

Figure 16: Invocation of user defined function
Functions called inside a TTCN-3 construct with an associated GFT symbol are represented as text within that 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).

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

11.3 Declarations

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

The following declarations shall be made within action symbols:

- timer declarations;
- declarations of variables without initialization;
- declarations of variables and constants with initialization;
  - if the initialization is not made by means of functions that include communication functions; or
  - if a declaration is:
    - of a component type that is not initialized by means of a create operation;
    - of type default that is not initialized by means of an activate operation;
    - of type verdicttype that is not initialized by means of an execute statement;
    - of a simple basic type;
    - of a basic string type;
    - of the type anytype;
    - of a port type;
    - of the type address; or
    - of a user-defined structured type with fields that fulfil all restrictions mentioned in this bullet for "declarations of variables and constants with initialization".

NOTE: Please refer to ES 201 873-1 [1], table 3 for an overview on TTCN-3 types.

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

11.3.2 Declaration of constants and variables within inline expression symbols

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

11.3.3 Declaration of constants and variables within create symbols

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

11.3.4 Declaration of constants and variables within default symbols

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

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

11.3.6 Declaration of constants and variables within execute test case symbols

Constants and variable declarations of type \texttt{verdicttype} that are initialized by means of \texttt{execute} statements shall be made within execute test case symbols. In contrast to declarations within action symbols, each declaration that is initialized by means of an \texttt{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).

![Diagram showing variable declarations in GFT](image)

\textbf{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 \texttt{log, label, goto, if-else, for, while} and \texttt{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")
```

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

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;
```

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;
```

Figure 23: Goto Statement
11.4.4 The If-else statement

The **if-else** statement shall be represented by an inline expression symbol labelled with the **if** keyword and a Boolean expression as defined in ES 201 873-1 [1], clause 19.6. The if-else inline expression symbol may contain one or two operands, separated by a dashed line. Figure 24 illustrates an **if** statement with a single operand, which is executed when the Boolean expression \( x>1 \) evaluates to true. Figure 25 illustrates an **if-else** statement in which the top operand is executed when the Boolean expression \( x>1 \) evaluates to true, and the bottom operand is executed if the Boolean expression evaluates to false.

![Figure 24: If-Statement](image)

![Figure 25: If-else Statement](image)

11.4.5 The For statement

The **for** statement shall be represented by an inline expression symbol labelled with a **for** definition as defined in 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 initialized within the **for** statement.

![Figure 26: For Statement](image)
11.4.6 The While statement

The `while` symbol shall be represented by an inline expression symbol labelled with a `while` definition as defined in 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)
}
```

Figure 27: While Statement

11.4.7 The Do-while statement

The `do-while` statement shall be represented by an inline expression symbol labelled with a `do-while` definition as defined in 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);
```

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`. 
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.

11.5.2 Alternative Behaviour

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

Note that an alternative inline expression should always cover all port instances, if communication operators are involved. Figure 31 illustrates an alternative behaviour in which either a message event is received with the value defined by Template1, or a message event is received with the value defined by Template2. The invocation of an altstep in an alternative inline expression is shown in figure 32.
In addition, it is possible to call an altstep as the only event within an alternative operand. This shall be drawn using a reference symbol (see clause 11.2.3).

11.5.2.1 Selecting/Deselecting an Alternative

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

```
Template1
alt
x>1
else
MyErrorHandler()
```

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

**Figure 34: Else within an alternative**

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

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

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

11.5.3 The Repeat statement

The `repeat` statement shall be represented by a repeat symbol. This symbol shall only be used as last event of an alternative operand in an `alt` statement or as last event of an operand of the top alternative in an `altstep` definition. Figure 35 illustrates an alternative statement in which the second operand, having successfully received a message with a value matching `Template2`, causes the alternative to be repeated.

```
Template1
alt
MyPort
```

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

**Figure 35: Repeat within an alternative**
11.5.4 Interleaved Behaviour

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

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

NOTE: An interleave inline expression should always cover all port instances if communication operators are involved.

![Figure 36: Interleave statement](image)

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;
```

![Figure 37: Return symbol without a return value](image)

```
return ReturnValue;
```

![Figure 38: Return symbol with a return value](image)
11.6 Default handling

GFT provides graphical representation for the activation and deactivation of defaults (see ES 201 873-1 [1], clause 21).

11.6.1 Default references

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

11.6.2 The activate operation

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

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

Figure 39: Default activation

11.6.3 The deactivate operation

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

```
deactivate(MyDefaultVar);
```

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;
```

![Figure 41: Create operation](image)

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

![Figure 42: Connect and map operation](image)

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

![Figure 43: Disconnect and unmap operation](image)

11.7.4 The Start test component operation

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

```
```

![Figure 44: Start operation](image)
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.

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

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

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.

\[ \text{MyVariable + YourVariable - 2} \]

\[ \text{MyPeer} \]

Figure 49: General format of sending operations

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

11.8.2 General format of the receiving operations

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

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

The receive part:

- specifies the port at which the operation shall take place;
- defines a matching part consisting of an optional type information and the matching value which specifies the acceptable input which will match the statement;
- 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).
11.8.3 Message-based communication

11.8.3.1 The Send operation

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

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

```
MyPort

5

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

---

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

```
MyPort

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

---

```
MyPort

MyType

MyTemplate(5, MyVar)

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

---

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

```
MyPort

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

---

**Figure 55: Receive operation with inline template**
11.8.3.2.1 Receive any message

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

```
MyPort
```

```
MyPort.receive;
```

Figure 56: Receive any message

11.8.3.2.2 Receive on any port

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

```
MyMessage
```

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

Figure 57: Receive on any port

11.8.3.3 The Trigger operation

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

```
trigger MyType
MyTemplate
```

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

Figure 58: Trigger operation with template reference
### 11.8.3.3.1 Trigger on any message

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

### 11.8.3.3.2 Trigger on any port

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

### 11.8.4 Procedure-based communication

#### 11.8.4.1 The Call operation

#### 11.8.4.1.1 Calling blocking procedures

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

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).

11.8.4.1.2 Calling non-blocking procedures

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

11.8.4.2 The Getcall operation

The getcall operation shall be represented by an incoming message arrow from the port instance to the test component and the keyword \texttt{getcall} above the message arrow preceding the signature. The signature is placed above the message arrow subsequent to the keyword \texttt{getcall}. The (inline) template is placed underneath the message arrow (see figures 68 and 69).
11.8.4.2.1 Accepting any call

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

11.8.4.2.2 Getcall on any port

The getcall on any port operation is represented by a found symbol representing any port to the test component and the keyword `getcall` above the message arrow followed by the signature if present. The (inline) template if present shall be placed underneath the message arrow (see figure 71).
11.8.4.3 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 figures 72 and 73).

```
MyPort
reply MyProc
MyTemplateRef
value 20
```

MyPort.reply(MyProc: MyTemplateRef value 20);

Figure 72: Reply operation with template reference

```
MyPort
reply MyProc
[5, MyVar2]
value 20
```

MyPort.reply(MyProc: [5, MyVar2] value 20);

Figure 73: Reply operation with inline template

11.8.4.4 The Getreply operation

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

The signature shall be placed above the message arrow subsequent to the keyword **getreply**. The (inline) template shall be placed underneath the message arrow.
Figure 74: Getreply operation with template reference (within a call symbol)

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

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

Figure 77: Getreply operation with inline template (outside a call symbol)
11.8.4.4.1 Get any reply from any call

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

```
getreply
from MyPeer:
[] MyPort.getreply from MyPeer { ... }
```

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

```
getreply
MyPort:
MyPort.getreply;
```

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

11.8.4.4.2 Get a reply on any port

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

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

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

**Figure 80: Get a reply on any port (within a call symbol)**
11.8.4.5 The Raise operation

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

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

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

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

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

11.8.4.6 The Catch operation

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

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

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

Figure 88: Catch operation with template reference (within a call symbol)

Figure 89: Catch operation with inline template (within a call symbol)

Figure 90: Catch operation with template reference (outside a call symbol)

Figure 91: Catch operation with inline template (outside a call symbol)
11.8.4.6.2 Catch any exception

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

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

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

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

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

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

11.8.5 The Check operation

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

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

Figure 93: Check a receive with inline template
11.8.5.1 The Check any operation

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

11.8.5.2 Check on any port

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

11.8.6.1 The Clear port operation

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

11.8.6.2 The Start port operation

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

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

```
stop
```

```
MyPort.stop;
```

![Figure 100: Stop port operation](image)

11.8.6.4 Use of any and all with ports

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

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

11.9 Timer operations

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

```
MyTimer
```

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

![Figure 101: Identified timer and call timers](image)

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

11.9.1 The Start timer operation

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

```
MyTimer(10.0)
```

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

![Figure 102: The start timer operation](image)
11.9.2 The Stop timer operation

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

![Figure 103: The stop timer operation](image)

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

![Figure 104: Connected start and stop timer symbols](image)

11.9.3 The Timeout operation

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

![Figure 105: The timeout operation](image)

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

![Figure 106: Connected start and timeout timer symbols](image)
11.9.4 The Read timer operation

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

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

Figure 107: The read timer operation

11.9.5 Use of any and all with timers

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

```
all timer.stop;  
```

Figure 108: Stopping all timers

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

```
any timer.timeout;  
```

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).

```
setverdict(pass);  
```

Figure 110: Set local verdict

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

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

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

![Figure 111: External actions](image)

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()'
    }
```

![Figure 112: Specifying attributes](image)
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 (ITU-T Recommendation Z.120 [3]). The graphical syntax definition uses a meta-language, which is explained in clause 1.3.4 of ITU-T Recommendation Z.120 [3]:

"The graphical syntax is not precise enough to describe the graphics such that there are no graphical variations. Small variations on the actual shapes of the graphical terminal symbols are allowed. These include, for instance, shading of the filled symbols, the shape of an arrow head and the relative size of graphical elements. Whenever necessary the graphical syntax will be supplemented with informal explanation of the appearance of the constructions.

The meta-language consists of a BNF-like notation with the special meta-constructions: contains, is followed by, is associated with, is attached to, above and set. These constructs behave like normal BNF production rules, but additionally they imply some logical or geometrical relation between the arguments. The is attached to construct behaves somewhat differently as explained below. The left-hand side of all constructs except above must be a symbol. A symbol is a non-terminal that produces in every production sequence exactly one graphical terminal. We will consider a symbol that is attached to other areas or that is associated with a text string as a symbol too. The explanation is informal and the meta-language does not precisely describe the geometrical dependencies."


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>
<thead>
<tr>
<th>Meaning</th>
<th>TTCN-3</th>
<th>GFT</th>
<th>MSC</th>
<th>Differences</th>
</tr>
</thead>
<tbody>
<tr>
<td>is defined to be</td>
<td>::=</td>
<td>::=</td>
<td>::=</td>
<td></td>
</tr>
<tr>
<td>abc followed by xyz</td>
<td>abc xyz</td>
<td>abc xyz</td>
<td>abc xyz</td>
<td></td>
</tr>
<tr>
<td>Alternative</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 or 1 instances of abc</td>
<td>[abc]</td>
<td>[abc]</td>
<td>[abc]</td>
<td></td>
</tr>
<tr>
<td>0 or more instances of abc</td>
<td>{abc}</td>
<td>{abc}</td>
<td>(abc)+</td>
<td>X</td>
</tr>
<tr>
<td>1 or more instances of abc</td>
<td>(abc)+</td>
<td>(abc)+</td>
<td>(abc)+</td>
<td></td>
</tr>
<tr>
<td>Textual grouping</td>
<td>(...)</td>
<td>(...)</td>
<td>(...)</td>
<td>X</td>
</tr>
<tr>
<td>the non-terminal symbol abc</td>
<td>abc</td>
<td>abc (for a GFT non-terminal) or abc (for a TTCN non-terminal)</td>
<td>&lt;abc&gt;</td>
<td>X</td>
</tr>
<tr>
<td>a terminal symbol abc</td>
<td>abc</td>
<td>abc</td>
<td>abc or &lt;name&gt; or &lt;character string&gt;</td>
<td>X</td>
</tr>
</tbody>
</table>
A.3 The GFT grammar

A.3.1 Diagrams

A.3.1.1 Control diagram

ControlDiagram ::= Frame contains ( ControlHeading ControlBodyArea )

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

ControlBodyArea ::= { ControlInstanceArea TextLayer ControlEventLayer } set

TextLayer ::= [ TextArea ] set

ControlEventLayer ::= ControlEventArea ControlEventArea above ControlEventLayer

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

A.3.1.2 Testcase diagram

TestcaseDiagram ::= Frame contains ( TestcaseHeading TestcaseBodyArea )

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

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

InstanceLayer ::= { InstanceArea } set

InstanceEventLayer ::= InstanceEventArea InstanceEventArea above InstanceEventLayer

InstanceEventArea ::= ( InstanceSendEventArea InstanceReceiveEventArea InstanceCallEventArea InstanceReplyEventArea InstanceGetreplyWithinCallEventArea InstanceGetreplyOutsideCallEventArea InstanceRaiseEventArea InstanceCatchWithinCallEventArea InstanceCatchTimeoutWithinCallEventArea InstanceCatchOutsideCallEventArea InstanceTriggerEventArea InstanceCheckEventArea InstanceFoundEventArea InstanceTimerEventArea InstanceActionArea InstanceLabellingArea)
/* STATIC SEMANTICS - a condition area containing a boolean expression shall be used within all inline expression, i.e. AltArea, and call inline expression, i.e. CallArea, only */

InstanceCallEventArea ::= 
| InstanceBlockingCallEventArea
| InstanceNonBlockingCallEventArea

PortEventLayer ::= 
| 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
                  'I' (AltstepFormalParList ] 'I' )
                  [ 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 { PortArea } 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 ::=  
\[\text{Diagram}
\]

TimerStartSymbol2 ::=  
\[\text{Diagram}
\]

InstanceTimerStopArea ::=  
TimerStopArea1 | TimerStopArea2  

TimerStopArea1 ::=  
TimerStopSymbol1  
\text{is associated with} \text{TimerRef}  
\text{is attached to} \text{InstanceAxisSymbol}  

TimerStopArea2 ::=  
TimerStopSymbol2  
\text{is attached to} \text{InstanceAxisSymbol}  
\text{is attached to} \text{TimerStartSymbol}  

TimerStopSymbol1 ::=  
\[\text{Diagram}
\]

TimerStopSymbol2 ::=  
\[\text{Diagram}
\]

InstanceTimeoutArea ::=  
TimeoutArea1 | TimeoutArea2  

TimeoutArea1 ::=  
TimeoutSymbol  
\text{is associated with} \text{TimerRef}  
\text{is attached to} \text{InstanceAxisSymbol}  

TimeoutArea2 ::=  
TimeoutSymbol3  
\text{is attached to} \text{InstanceAxisSymbol}  
\text{is attached to} \text{TimerStartSymbol}  

TimeoutSymbol ::=  
TimeoutSymbol1 | TimeoutSymbol2  

TimeoutSymbol1 ::=  
\[\text{Diagram}
\]

TimeoutSymbol2 ::=  
\[\text{Diagram}
\]
A.3.4  Action

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

ActionSymbol ::=  

ActionStatement ::=  
  SUTStatements
  | ConnectStatement
  | DisconnectStatement
  | MapStatement
  | 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 */
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 */

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 */

A.3.7.2 Starting test components

InstanceComponentStartArea ::=  
  StartSymbol  
  contains StartTCStatement  
  is attached to InstanceAxisSymbol
A.3.7.3 Stopping test components

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

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

A.3.8 Inline expressions

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

IfArea ::= IfInlineExpressionArea
   is attached to InstanceInlineExpressionBeginSymbol
   [ is attached to InstanceInlineExpressionSeparatorSymbol ]
   is attached to InstanceInlineExpressionEndSymbol
   [ is attached to ( PortInlineExpressionBeginSymbol ) set
   [ is attached to ( PortInlineExpressionSeparatorSymbol ) set ]
   is attached to ( PortInlineExpressionEndSymbol ) set ]

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

InstanceInlineExpressionBeginSymbol ::= VoidSymbol

InstanceInlineExpressionSeparatorSymbol ::= VoidSymbol

InstanceInlineExpressionEndSymbol ::= VoidSymbol

VoidSymbol ::= .

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

OperandArea ::= ConnectorLayer

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

ForArea ::= ForInlineExpressionArea
   is attached to InstanceInlineExpressionBeginSymbol
   is attached to InstanceInlineExpressionEndSymbol
   [ is attached to ( PortInlineExpressionBeginSymbol ) set
   is attached to ( PortInlineExpressionEndSymbol ) set ]

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

WhileArea ::= WhileInlineExpressionArea
   is attached to InstanceInlineExpressionBeginSymbol
   is attached to InstanceInlineExpressionEndSymbol
   [ is attached to ( PortInlineExpressionBeginSymbol ) set
   is attached to ( PortInlineExpressionEndSymbol ) set ]
```
WhileInlineExpressionArea ::= InlineExpressionSymbol
    contains ( WhileKeyword '(' BooleanExpression ')' 
        is followed by OperandArea )

DoWhileArea ::= DoWhileInlineExpressionArea
    is attached to InstanceInlineExpressionBeginSymbol
    is attached to InstanceInlineExpressionEndSymbol
    [ is attached to ( PortInlineExpressionBeginSymbol ) set ]
    [ is attached to ( PortInlineExpressionEndSymbol ) set ]

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

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

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 InstanceEventArea | 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 InstanceInterleaveGuardArea
 is followed by InstanceInterleaveGuardArea
 is followed by InstanceInlineExpressionEndSymbol
 is attached to InstanceAxisSymbol
 is attached to InterleaveInlineExpressionArea

InstanceInterleaveGuardArea ::= InstanceGuardOpArea

InstanceInlineExpressionBeginSymbol
 { is followed by InstanceEventArea | 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

InstanceInlineExpressionEndSymbol
 is attached to InstanceAxisSymbol
 is attached to InstanceInlineExpressionArea
/* 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 ::=  
  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 followed by 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

PortEventArea ::=  
  VoidSymbol

PortAxisSymbol ::=  
  VoidSymbol
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 PortInlineExpressionEndSymbol } |
| 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 PortInlineExpressionEndSymbol } |
| 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 InstanceInlineExpressionEndSymbol ]
| is followed by ControlInlineExpressionEndSymbol )
| is attached to InstanceAxisSymbol
| is attached to ElseIfInlineExpressionArea

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:

/* 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.
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 

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

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

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
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 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 */
/* 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 */
/* STATIC SEMANTICS - a port redirect, if existent, shall be put underneath the message symbol */

CatchWithinCallArea ::= MessageSymbol
   is associated with 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 shall be put underneath the message symbol */
/* STATIC SEMANTICS - a to 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 CallTimeoutSymbol13 ]

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
A.3.14 Labelling

InstanceLabellingArea ::= LabellingSymbol
  contains LabelIdentifier
  is attached to InstanceAxisSymbol

/* STATIC SEMANTICS - the label identifier shall be placed inside the circle of the labelling symbol */
Annex B (informative):
Reference Guide for GFT

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

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

\[
\text{const integer} \\
\text{MyConst} := 5;
\]
<table>
<thead>
<tr>
<th>Language Element</th>
<th>Associated Keyword</th>
<th>GFT symbols, if existent, and typical usage</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control definitions</td>
<td>control</td>
<td><img src="image" alt="Control Diagram" /></td>
<td>GFT control diagram represents the control part of a TTCN-3 module.</td>
</tr>
<tr>
<td>Function definitions</td>
<td>function</td>
<td><img src="image" alt="Function Diagram" /></td>
<td>GFT function diagrams are used to represent functions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><img src="image" alt="Function Diagram" /></td>
<td>GFT function diagrams may be defined to structure the behaviour of the control part of a TTCN-3 module.</td>
</tr>
<tr>
<td>Altstep definitions</td>
<td>altstep</td>
<td><img src="image" alt="Altstep Diagram" /></td>
<td>GFT altstep diagrams are used to represent altsteps.</td>
</tr>
<tr>
<td>Language Element</td>
<td>Associated Keyword</td>
<td>GFT symbols, if existent, and typical usage</td>
<td>Explanation</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>--------------------</td>
<td>--------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Test case definitions</td>
<td>testcase</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><img src="image" alt="GFT test case diagram" /></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>GFT test case diagrams are used to represent test cases.</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Usage of Component Instances and Ports</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Port instance</td>
<td></td>
<td><img src="image" alt="Port instance diagram" /></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>A Port in a test case, test step and function diagram is represented by an instance with a dashed instance line. The port name is specified above and the (optional) port type is described within the instance header.</strong></td>
<td></td>
</tr>
<tr>
<td>Test component instance</td>
<td></td>
<td><img src="image" alt="Test component instance diagram" /></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>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 diagram. A control instance represents the instance that executes the module control part in a control diagram.</strong></td>
<td></td>
</tr>
<tr>
<td>Declarations</td>
<td></td>
<td><img src="image" alt="Variable declarations diagram" /></td>
<td></td>
</tr>
<tr>
<td>Variable declarations</td>
<td>var</td>
<td><strong>var integer MyVar := 5</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Textual variable declaration in the header of a control, test case, test step or function diagram.</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><img src="image" alt="Variable declaration in an action box" /></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Variable declaration in an action box.</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><img src="image" alt="Variable declaration within a test case execution symbol" /></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Variable declaration within a test case execution symbol.</strong></td>
<td></td>
</tr>
<tr>
<td>Language Element</td>
<td>Associated Keyword</td>
<td>GFT symbols, if existent, and typical usage</td>
<td>Explanation</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------</td>
<td>---------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Variable declaration within a test component creation symbol.</td>
<td>var MyCType</td>
<td>c:= MyCType.create</td>
<td></td>
</tr>
<tr>
<td>Variable declaration within a default activation symbol.</td>
<td>var default d:= activate(TStep())</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable declaration within a reference symbol.</td>
<td>var integer</td>
<td>v := MyFunction()</td>
<td></td>
</tr>
<tr>
<td>Textual timer declaration in the header of a control, test case, test step or function diagram.</td>
<td>timer</td>
<td>MyTimer</td>
<td></td>
</tr>
<tr>
<td>Timer declaration in an action box.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assignment in an action box.</td>
<td>:</td>
<td>MyVar := 5</td>
<td></td>
</tr>
<tr>
<td>Assignment within a test case execution symbol.</td>
<td>v:=execute(MyTC())</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Timer declarations

Expressions

Assignments
<table>
<thead>
<tr>
<th>Language Element</th>
<th>Associated Keyword</th>
<th>GFT symbols, if existent, and typical usage</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment within a test component creation symbol.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assignment within a default activation symbol.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assignment within a reference symbol.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The log statement is put into an action box.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Definition of a label.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Go to label.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Log**

```
c:= MyCType.Create
```

**Label and Goto**

```
d:= activate(TStep())
v := MyFunction()
```

**If-else**

```
if (j<10)
```

```
<table>
<thead>
<tr>
<th>Language Element</th>
<th>Associated Keyword</th>
<th>GFT symbols, if existent, and typical usage</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>For loop</td>
<td>for (…)(…)</td>
<td>![for(j:=1; j&lt;9; j:=j+1)]</td>
<td></td>
</tr>
<tr>
<td>While loop</td>
<td>while (…) (….)</td>
<td>![while (j&lt;10)]</td>
<td></td>
</tr>
<tr>
<td>Do while loop</td>
<td>do (…)(…)</td>
<td>![do while (j&lt;10)]</td>
<td></td>
</tr>
<tr>
<td>Alternative behaviour</td>
<td>alt (….)</td>
<td>![alt [x:=1] MyTestStep()]</td>
<td></td>
</tr>
<tr>
<td>Repeat</td>
<td>repeat</td>
<td>![repeat]</td>
<td>To be used within alternative behaviour and test steps.</td>
</tr>
<tr>
<td>Interleaved behaviour</td>
<td>intertwine (….)</td>
<td>![intertwine]</td>
<td></td>
</tr>
<tr>
<td>Language Element</td>
<td>Associated Keyword</td>
<td>GFT symbols, if existent, and typical usage</td>
<td>Explanation</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------</td>
<td>---------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Activate a default</td>
<td>activate</td>
<td>MyDefault := activate(TStep())</td>
<td>The activate statement is put into a default symbol.</td>
</tr>
<tr>
<td>Deactivate a default</td>
<td>deactivate</td>
<td>deactivate(MyDefault)</td>
<td>The deactivate statement is put into a default symbol.</td>
</tr>
<tr>
<td>Returning control</td>
<td>return</td>
<td>MyValue</td>
<td>The optional return value is attached to the return symbol.</td>
</tr>
</tbody>
</table>

**Configuration operations**

<p>| Create parallel test component | create | c:= MyCType.create | The create statement is put into a test component creation symbol. |
| Connect component to component | connect | connect(...)       | The connect statement is put into an action box. |
| Disconnect two components   | disconnect | disconnect(...)    | The disconnect statement is put into an action box. |</p>
<table>
<thead>
<tr>
<th>Language Element</th>
<th>Associated Keyword</th>
<th>GFT symbols, if existent, and typical usage</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map port to test system interface</td>
<td>map</td>
<td></td>
<td>The map statement is put into an action box.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>![map (...,...)]</td>
<td></td>
</tr>
<tr>
<td>Unmap port from test system interface</td>
<td>unmap</td>
<td></td>
<td>The unmap statement is put into an action box.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>![unmap (...,...)]</td>
<td></td>
</tr>
<tr>
<td>Get MTC address</td>
<td>mtc</td>
<td></td>
<td>No special GFT symbol, used within statements, expressions or as test component identifier.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>![mtc]</td>
<td></td>
</tr>
<tr>
<td>Get test system interface address</td>
<td>system</td>
<td></td>
<td>No special GFT symbol, used within statements or expressions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>![system]</td>
<td></td>
</tr>
<tr>
<td>Get own address</td>
<td>self</td>
<td></td>
<td>No special GFT symbol, used within statements, expressions or as test component identifier.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>![self]</td>
<td></td>
</tr>
<tr>
<td>Start execution of test component</td>
<td>start</td>
<td></td>
<td>The start statement is put into a start symbol.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>![start (MyFunc())]</td>
<td></td>
</tr>
<tr>
<td>Stop execution of a test component by itself</td>
<td>stop</td>
<td></td>
<td>The termination of mtc terminates also all the other test components. Port instances cannot be stopped.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>![stop]</td>
<td></td>
</tr>
<tr>
<td>Of another test component</td>
<td></td>
<td>![componentId]</td>
<td>The component identifier is put near to the stop symbol.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check termination of a PTC</td>
<td>running</td>
<td></td>
<td>No special GFT symbol, used within expressions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Language Element</td>
<td>Associated Keyword</td>
<td>GFT symbols, if existent, and typical usage</td>
<td>Explanation</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------------------</td>
<td>---------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Wait for termination of a PTC</td>
<td>done</td>
<td>MyPTC.done</td>
<td>The done statement is put into a condition symbol.</td>
</tr>
<tr>
<td>Communication operations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send message</td>
<td>send</td>
<td>MyTemplateRef to MyPeer</td>
<td>Send a message defined by a template reference but without type information. The receiver is identified uniquely by the (optional) to-directive.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MyTemplateRef MyType</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MyTemplateRef (...)</td>
<td>Send a message defined by an inline template definition. An (optional) to-directive may be present to identify the peer entity uniquely.</td>
</tr>
<tr>
<td>Receive message</td>
<td>receive</td>
<td>MyTemplateRef from MyPeer -&gt; value MyVar sender ASender</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Diagram**:
- **Wait for termination of a PTC**
- **Send message**
- **Receive message**
<table>
<thead>
<tr>
<th>Language Element</th>
<th>Associated Keyword</th>
<th>GFT symbols, if existent, and typical usage</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive a message with a value defined by a template reference and with type information. Optional <code>from</code>, <code>value</code>- and <code>sender</code>-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.</td>
<td>MyTemplateRef MyType</td>
<td><img src="image1" alt="Diagram" /></td>
<td>Receive a message with a value defined by a template reference and with type information. Optional <code>from</code>, <code>value</code>- and <code>sender</code>-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.</td>
</tr>
<tr>
<td>Receive a message with a value defined by an inline template definition. Optional <code>from</code>, <code>value</code>- and <code>sender</code>-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.</td>
<td>MyType {…}</td>
<td><img src="image2" alt="Diagram" /></td>
<td>Receive a message with a value defined by an inline template definition. Optional <code>from</code>, <code>value</code>- and <code>sender</code>-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.</td>
</tr>
<tr>
<td>Receive any message (no value and no type is specified). Optional <code>from</code>, <code>value</code>- and <code>sender</code>-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.</td>
<td></td>
<td><img src="image3" alt="Diagram" /></td>
<td>Receive any message (no value and no type is specified). Optional <code>from</code>, <code>value</code>- and <code>sender</code>-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.</td>
</tr>
<tr>
<td>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 <code>from</code>, <code>value</code>- and <code>sender</code>-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.</td>
<td></td>
<td><img src="image4" alt="Diagram" /></td>
<td>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 <code>from</code>, <code>value</code>- and <code>sender</code>-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.</td>
</tr>
<tr>
<td>Trigger message trigger</td>
<td>trigger MyTemplateRef from MyPeer -&gt; value MyVar sender ASender</td>
<td><img src="image5" alt="Diagram" /></td>
<td>Trigger on a message with a value defined by a template reference but without type information. The (optional) <code>from</code>-directive denotes that the sender of the message shall be identified by variable MyPeer. The (optional) <code>value</code>-directive assigns received message to variable MyVar. The (optional) <code>sender</code>-directive retrieves the identifier of the sender and stores it in variable ASender.</td>
</tr>
<tr>
<td>Language Element</td>
<td>Associated Keyword</td>
<td>GFT symbols, if existent, and typical usage</td>
<td>Explanation</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------</td>
<td>---------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>trigger MyType</td>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>trigger MyType</td>
<td>{...}</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>trigger</td>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>trigger</td>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>invoke blocking procedure call</td>
<td>call</td>
<td></td>
<td>Invoking a blocking procedure by using a signature template. The receiver is identified uniquely by the (optional) to-directive. The call body, i.e. possible getreply and catch operations, is shown schematically only.</td>
</tr>
<tr>
<td>Language Element</td>
<td>Associated Keyword</td>
<td>GFT symbols, if existent, and typical usage</td>
<td>Explanation</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------</td>
<td>---------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>Call</strong></td>
<td></td>
<td><img src="call-diagram.png" alt="Call Diagram" /></td>
<td>Invoking a blocking procedure by using a signature template and signature information. An (optional) <code>to</code>-directive may be present to identify the peer entity uniquely. The call body, i.e. possible <code>getreply</code> and <code>catch</code> operations, is shown schematically only.</td>
</tr>
<tr>
<td><strong>Call</strong></td>
<td></td>
<td><img src="call-diagram.png" alt="Call Diagram" /></td>
<td>Invoking a blocking procedure by using an inline template. An (optional) <code>to</code>-directive may be present to identify the peer entity uniquely. The call body, i.e. possible <code>getreply</code> and <code>catch</code> operations, is shown schematically only.</td>
</tr>
<tr>
<td><strong>Invoke non-blocking procedure call</strong></td>
<td><strong>Call</strong></td>
<td><img src="invoke-diagram.png" alt="Invoke non-blocking procedure call Diagram" /></td>
<td>Call a remote procedure, the call is defined by a template reference but without signature information. The receiver is identified uniquely by the (optional) <code>to</code>-directive.</td>
</tr>
<tr>
<td><strong>Call</strong></td>
<td></td>
<td><img src="call-diagram.png" alt="Call Diagram" /></td>
<td>Call the remote procedure <code>MyProc</code>. The call is defined by a template reference. An (optional) <code>to</code>-directive may be present to identify the peer entity uniquely.</td>
</tr>
<tr>
<td><strong>Call</strong></td>
<td></td>
<td><img src="call-diagram.png" alt="Call Diagram" /></td>
<td>Call the remote procedure <code>MyProc</code>. The call is defined by an inline template. An (optional) <code>to</code>-directive may be present to identify the peer entity uniquely.</td>
</tr>
<tr>
<td>Language Element</td>
<td>Associated Keyword</td>
<td>GFT symbols, if existent, and typical usage</td>
<td>Explanation</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------</td>
<td>-------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Reply to procedure call from remote entity</td>
<td>reply</td>
<td>MyTemplateRef value 20 to MyPeer</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>reply MyProc</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MyTemplateRef value 20</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>reply MyProc</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{...} value 20</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>reply MyProc</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>raise MyProc</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MyTemplateRef to MyPeer</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>raise MyProc</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ExceptionType</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MyTemplateRef</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>raise MyProc</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ExceptionType</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{...}</td>
<td></td>
</tr>
</tbody>
</table>

- Reply to a remote procedure call. The reply is defined by a template reference and the possible return value (value-directive).
- NOTE 1: The signature information is part of the template definition. The receiver is identified uniquely by the (optional) to-directive.
- Reply to a remote procedure call of MyProc. The reply is defined by a template reference and the possible return value (value-directive). An (optional) to-directive may be present to identify the peer entity uniquely.
- Reply to a remote procedure call of MyProc. The reply is defined by an inline template and the possible return value (value-directive). An (optional) to-directive may be present to identify the peer entity uniquely.
- Raise an exception to an accepted call of MyProc. The exception is defined by a template reference. NOTE 2: The type of the exception is defined within the template definition. The receiver is identified uniquely by the (optional) to-directive.
- Raise an exception to an accepted call of MyProc. The exception is defined by its (optional) type and a template reference. An (optional) to-directive may be present to identify the peer entity uniquely.
- Raise an exception to an accepted call of MyProc. The exception is defined by its type and an inline template. An (optional) to-directive may be present to identify the peer entity uniquely.
<table>
<thead>
<tr>
<th>Language Element</th>
<th>Associated Keyword</th>
<th>GFT symbols, if existent, and typical usage</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accept procedure call from remote entity</td>
<td>getcall</td>
<td>MyTemplateRef from MyPeer -&gt; param (...) sender ASender</td>
<td>Accept a procedure call from a remote entity. The call signature has to match the conditions defined by the template reference. NOTE 3: The signature information is part of the template definition. The (optional) from-directive denotes that the sender of the call shall be identified by variable MyPeer. The (optional) param-directive assigns in-parameter values to Variables. The (optional) sender-directive retrieves the identifier of the sender and stores it in variable ASender.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MyTemplateRef MyProc</td>
<td>Accept a procedure call from a remote entity. The call signature has to match the conditions defined by signature reference and the template reference. Optional from-, param- and sender-directives may be present to identify the sender of the call, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MyProc (...)</td>
<td>Accept a procedure call from a remote entity. The call signature has to match the conditions defined by signature reference and the inline template definition. Optional from-, param- and sender-directives may be present to identify the sender of the call, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>getcall</td>
<td>Accept any procedure call from any remote entity. Optional from- and sender-directives may be present to identify the sender of the call or to retrieve the identifier of the peer entity.</td>
</tr>
<tr>
<td>Language Element</td>
<td>Associated Keyword</td>
<td>GFT symbols, if existent, and typical usage</td>
<td>Explanation</td>
</tr>
<tr>
<td>------------------------------------------------------</td>
<td>--------------------</td>
<td>---------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>getcall</strong></td>
<td></td>
<td></td>
<td>Accept any procedure call from any remote entity at any port. The call to be received from any port may be restricted by means referring to templates or by using inline templates. Optional <em>from</em>, <em>param</em> and <em>sender</em>-directives may be present to identify the sender of the call, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</td>
</tr>
<tr>
<td><strong>Handle response from a previous blocking call</strong></td>
<td>getreply</td>
<td></td>
<td>Receive a response from a blocking call. The reply has to match the conditions defined by the template reference. NOTE 4: The signature information is part of the template definition. The (optional) <em>from</em>-directive denotes that the sender of the call shall be identified by variable <em>MyPeer</em>. The (optional) <em>value</em>-directive assigns the possible return value of the procedure to variable <em>MyVal</em>. The (optional) <em>param</em>-directive assigns out-parameter values to Variables. The (optional) <em>sender</em>-directive retrieves the identifier of the sender and stores it in variable <em>ASender</em>.</td>
</tr>
<tr>
<td><strong>getreply</strong></td>
<td></td>
<td></td>
<td>Receive a response from a blocking call. The reply has to match the conditions defined by signature reference and the template reference. Optional <em>from</em>, <em>value</em>, <em>param</em> and <em>sender</em>-directives may be present to identify the sender of the reply, to retrieve the return value of the procedure, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</td>
</tr>
<tr>
<td>Language Element</td>
<td>Associated Keyword</td>
<td>GFT symbols, if existent, and typical usage</td>
<td>Explanation</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------</td>
<td>---------------------------------------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| **Receive a response from a**  
blocking call. The reply has to  
match the conditions defined  
by signature reference and the  
inline template definition.  
Optional from-, value-,  
param- and  
sender-directives may be  
present to identify the sender  
of the reply, to retrieve the  
return value of the procedure,  
to assign the in-parameters to  
variables or to retrieve the  
identifier of the peer entity. |  
```plaintext
MyProc
{ ... }
``` |  
```plaintext
call
getreply
```
```plaintext
catch ...
```
| **Accept any response from a**  
blocking call. |  
```plaintext
MyTemplateRef
from MyPeer
-> value MyVal
param (...) 
sender ASender
``` |  
```plaintext
call
getreply
```
```plaintext
catch ...
```
| **Handle response**  
from a previous non-  
blocking call or  
independent from a  
call. | getreply |  
```plaintext
getreply
MyTemplateRef
from MyPeer
-> value MyVal
param (...) 
sender ASender
``` |  
```plaintext
MyProc
{ ... }
``` |  
```plaintext
call
getreply
```
```plaintext
catch ...
```
| **Receive a response from a**  
previous call. The reply has to  
match the conditions defined  
by the template reference.  
NOTE 5: The signature  
information is part of the  
template definition.  
The (optional)  
from-directive denotes that the  
sender of the call shall be  
identified by variable MyPeer.  
The (optional)  
value-directive assigns the  
possible return value of the  
procedure to variable MyVal.  
The (optional)  
param-directive assigns out-  
parameter values to Variables.  
The (optional)  
sender-directive retrieves the  
identifier of the sender and  
stores it in variable ASender. |  
```plaintext
``` |  
```plaintext
handle response
getreply
``` |  
```plaintext
call
getreply
```
```plaintext
catch ...
```
<table>
<thead>
<tr>
<th>Language Element</th>
<th>Associated Keyword</th>
<th>GFT symbols, if existent, and typical usage</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><img src="https://via.placeholder.com/150" alt="image" /></td>
<td>Receive a response from a previous call. The reply has to match the conditions defined by signature reference and the template reference. Optional <code>from</code>, <code>value</code>, <code>param</code>- and <code>sender</code>-directives may be present to identify the sender of the reply, to retrieve the return value of the procedure, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><img src="https://via.placeholder.com/150" alt="image" /></td>
<td>Receive a response from a previous call. The reply has to match the conditions defined by signature reference and the inline template definition. Optional <code>from</code>, <code>value</code>, <code>param</code>- and <code>sender</code>-directives may be present to identify the sender of the reply, to retrieve the return value of the procedure, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><img src="https://via.placeholder.com/150" alt="image" /></td>
<td>Accept any response from any previous call. Optional <code>from</code>- and <code>sender</code>-directives may be present to identify the sender of the reply or to retrieve the identifier of the peer entity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><img src="https://via.placeholder.com/150" alt="image" /></td>
<td>Accept any response from any previous call at any port. The reply to be received from any port may be restricted by means referring to templates or by using inline templates. Optional <code>from</code>, <code>value</code>, <code>param</code>- and <code>sender</code>-directives may be present to identify the sender of the reply, to retrieve the return value of the procedure, to assign the in-parameters to variables or to retrieve the identifier of the peer entity.</td>
</tr>
<tr>
<td>Language Element</td>
<td>Associated Keyword</td>
<td>GFT symbols, if existent, and typical usage</td>
<td>Explanation</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------</td>
<td>------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Catch exception from a previous blocking call</td>
<td>catch</td>
<td><img src="catch_call_diagram.png" alt="Diagram" /></td>
<td>Catch an exception from a previous call. The exception has to match the conditions defined by the template reference. NOTE 6: The type information is part of the template definition. The (optional) from-directive denotes that the sender of the exception shall be identified by variable MyPeer. The (optional) value-directive assigns the value of the exception to variable MyVal. The (optional) sender-directive retrieves the identifier of the sender and stores it in variable ASender.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><img src="catch_call_diagram.png" alt="Diagram" /></td>
<td>Catch an exception from a previous call. The exception has to match the conditions defined by the exception type and the template reference. Optional from-, value-, and sender-directives may be present to identify the sender of the exception, to retrieve the exception value or to retrieve the identifier of the peer entity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><img src="catch_call_diagram.png" alt="Diagram" /></td>
<td>Catch an exception from a previous call. The exception has to match the conditions defined by the exception type and the inline template definition. Optional from-, value-, and sender-directives may be present to identify the sender of the exception, to retrieve the exception value or to retrieve the identifier of the peer entity.</td>
</tr>
<tr>
<td>Language Element</td>
<td>Associated Keyword</td>
<td>GFT symbols, if existent, and typical usage</td>
<td>Explanation</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------</td>
<td>------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Catch exception from a previous non-blocking call or independent from a call</td>
<td>catch</td>
<td><img src="image1.png" alt="Diagram" /></td>
<td>Catch an exception from a previous call. The exception has to match the conditions defined by the template reference. <strong>NOTE 7:</strong> The type information is part of the template definition. The (optional) <code>from</code>-directive denotes that the sender of the exception shall be identified by variable <code>MyPeer</code>. The (optional) <code>value</code>-directive assigns the value of the exception to variable <code>MyVal</code>. The (optional) <code>sender</code>-directive retrieves the identifier of the sender and stores it in variable <code>ASender</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><img src="image2.png" alt="Diagram" /></td>
<td>Catch an exception from a previous call. The exception has to match the conditions defined by the exception type and the inline template definition. Optional <code>from</code>, <code>value</code>, and <code>sender</code>-directives may be present to identify the sender of the exception, to retrieve the exception value or to retrieve the identifier of the peer entity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><img src="image3.png" alt="Diagram" /></td>
<td>Catch an exception from a previous call. The exception has to match the conditions defined by the exception type and the template reference. Optional <code>from</code>, <code>value</code>, and <code>sender</code>-directives may be present to identify the sender of the exception, to retrieve the exception value or to retrieve the identifier of the peer entity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><img src="image4.png" alt="Diagram" /></td>
<td>Catch an exception from a previous call. The exception has to match the conditions defined by the exception type and the inline template definition. Optional <code>from</code>, <code>value</code>, and <code>sender</code>-directives may be present to identify the sender of the exception, to retrieve the exception value or to retrieve the identifier of the peer entity.</td>
</tr>
<tr>
<td>Language Element</td>
<td>Associated Keyword</td>
<td>GFT symbols, if existent, and typical usage</td>
<td>Explanation</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------</td>
<td>-------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Catch any exception from any previous call.</td>
<td></td>
<td></td>
<td>Catch any exception from any previous call. Optional <code>from</code>, <code>value</code>- and <code>sender</code>-directives may be present to identify the sender of the exception, to retrieve the exception value (and assign it to a variable of type <code>anytype</code>) or to retrieve the identifier of the peer entity.</td>
</tr>
<tr>
<td>Catch any exception from any previous call at any port.</td>
<td></td>
<td></td>
<td>Catch any exception from any previous call at any port. The exception to be received from any port may be restricted by means referring to templates or by using inline templates. Optional <code>from</code>, <code>value</code>, and <code>sender</code>-directives may be present to identify the sender of the exception, to retrieve the exception value or to retrieve the identifier of the peer entity.</td>
</tr>
<tr>
<td>Check (current) message/call received</td>
<td><code>check</code></td>
<td></td>
<td>Can be used also in combination with <code>getcall</code>, <code>getreply</code>, and <code>catch</code></td>
</tr>
<tr>
<td><code>check</code> MyTemplateRef</td>
<td></td>
<td></td>
<td>with template, without type.</td>
</tr>
<tr>
<td><code>check</code> MyType MyTemplateRef</td>
<td></td>
<td></td>
<td>with template, with type.</td>
</tr>
<tr>
<td><code>check</code> MyTemplateRef</td>
<td></td>
<td></td>
<td>without template, without type (any message from that port).</td>
</tr>
<tr>
<td><code>check</code> MyTemplateRef</td>
<td></td>
<td></td>
<td>with template, without type, without port (this message from that port).</td>
</tr>
<tr>
<td>Language Element</td>
<td>Associated Keyword</td>
<td>GFT symbols, if existent, and typical usage</td>
<td>Explanation</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>--------------------</td>
<td>---------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Check current message, call, reply or exception</td>
<td>check</td>
<td><img src="image" alt="Check current message" /></td>
<td>Check if a message with a value defined by a template reference has been received. The syntax follows the syntax for the reception of messages. NOTE 8: Check may also be used in combination with <code>getcall</code>, <code>getreply</code> and <code>catch</code>.</td>
</tr>
<tr>
<td>Check MyTemplateRef from MyPeer -&gt; value MyVar</td>
<td>check MyType</td>
<td><img src="image" alt="Check MyTemplateRef" /></td>
<td>Check if a message with a value defined by a template reference has been received. The syntax follows the syntax for the reception of messages. NOTE 9: Check may also be used in combination with <code>getcall</code>, <code>getreply</code> and <code>catch</code>.</td>
</tr>
<tr>
<td>Check MyTemplateRef</td>
<td>check MyType</td>
<td><img src="image" alt="Check MyTemplateRef" /></td>
<td>Check if a message with a value defined by an inline template definition has been received. The syntax follows the syntax for the reception of messages. NOTE 10: Check may also be used in combination with <code>getcall</code>, <code>getreply</code> and <code>catch</code>.</td>
</tr>
<tr>
<td>Check MyVar</td>
<td>check</td>
<td><img src="image" alt="Check MyVar" /></td>
<td>Check if any message (no value and no type is specified) has been received. The syntax follows the syntax for the reception of messages. NOTE 11: Check may also be used in combination with <code>getcall</code>, <code>getreply</code> and <code>catch</code>.</td>
</tr>
<tr>
<td>Check ASender</td>
<td>check</td>
<td><img src="image" alt="Check ASender" /></td>
<td>Check if any message (no value and no type is specified) has been received at any port. The syntax follows the syntax for the reception of messages. NOTE 12: Check may also be used in combination with <code>getcall</code>, <code>getreply</code> and <code>catch</code>.</td>
</tr>
<tr>
<td>Clear port</td>
<td>clear</td>
<td><img src="image" alt="Clear port" /></td>
<td>The clear port statement is put into a condition symbol. The condition shall cover the instance of the port to be cleared only.</td>
</tr>
<tr>
<td>Language Element</td>
<td>Associated Keyword</td>
<td>GFT symbols, if existent, and typical usage</td>
<td>Explanation</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------------------</td>
<td>---------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Clear and give access to port</td>
<td>start</td>
<td>![Diagram of start port statement]</td>
<td>The <code>start</code> port statement is put into a condition symbol. The condition shall cover the instance of the port to be started only.</td>
</tr>
<tr>
<td>Stop access (receiving &amp; sending) at port</td>
<td>stop</td>
<td>![Diagram of stop port statement]</td>
<td>The <code>stop</code> statement is put into a condition symbol. The condition shall cover the instance of the port to be stopped only.</td>
</tr>
<tr>
<td>Timer operations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start timer</td>
<td>start</td>
<td>![Diagram of start timer operation]</td>
<td></td>
</tr>
<tr>
<td>Stop timer</td>
<td>stop</td>
<td>![Diagram of stop timer operation]</td>
<td></td>
</tr>
<tr>
<td>Read elapsed time</td>
<td>read</td>
<td>![Diagram of read elapsed time operation]</td>
<td>No special GFT symbol, used within statements or expressions.</td>
</tr>
<tr>
<td>Check if timer running</td>
<td>running</td>
<td>![Diagram of check if timer running operation]</td>
<td>No special GFT symbol, used within statements or expressions.</td>
</tr>
<tr>
<td>Timeout operation</td>
<td>timeout</td>
<td>![Diagram of timeout operation]</td>
<td></td>
</tr>
<tr>
<td>Set local verdict</td>
<td>verdict.set</td>
<td>![Diagram of set local verdict operation]</td>
<td>The verdict is put into a condition symbol.</td>
</tr>
<tr>
<td>Language Element</td>
<td>Associated Keyword</td>
<td>GFT symbols, if existent, and typical usage</td>
<td>Explanation</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>--------------------</td>
<td>---------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Get local verdict</td>
<td>verdict.get</td>
<td></td>
<td>No special GFT symbol, used within statements or expressions.</td>
</tr>
<tr>
<td>SUT operations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remote action to be done by the SUT</td>
<td>sut.action</td>
<td></td>
<td>The action statement is put into an action box.</td>
</tr>
<tr>
<td>Execution of test cases</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute test case</td>
<td>execute</td>
<td></td>
<td>The execute statement is put into a testcase execution symbol.</td>
</tr>
<tr>
<td>Attributes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Definition of attributes for control, testcases, teststeps and functions</td>
<td>with</td>
<td>with { display &quot;...&quot;; extension &quot;...&quot;; }</td>
<td>The with statement is put into a text symbol.</td>
</tr>
<tr>
<td>Comments</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comments within text</td>
<td>/* My several lines comment */</td>
<td>Can be used wherever text can be placed.</td>
<td></td>
</tr>
<tr>
<td>Comments for instance events</td>
<td>/* My instance event comment */</td>
<td>Shall be attached to events on a control, test component or port instance.</td>
<td></td>
</tr>
<tr>
<td>Comments control, test case, function or test step diagrams</td>
<td>/* My Comment explains a little bit more */</td>
<td>Shall be attached to events on a control, test component or port instance.</td>
<td></td>
</tr>
</tbody>
</table>
Annex C (informative):
Examples

C.1 The Restaurant example

```plaintext
package Restaurant;

var boolean internetService;
var boolean phoneService;
var boolean restaurantService;
var integer totalCreatedPTCs;
var integer maxNrActivePTCs;
var TestSystemType system:

var ReportType report;
var default def := activate (StandardDefault());
map(self:P1, system:mPCO);
if (internetService) {
    newInternetPTC();
} if (totalCreatedPTCs != createdPTCs and phoneService) {
   if (maxNrActivePTCs == 1) {
        all component.done;
   } newPhonePTC();
} while (totalCreatedPTCs != 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 C.1: Restaurant example - MyTestCase test case
function newInternetPTC()
  runs on MtcType

  self
  MtcType
  MtcType
  mPCOtype
  mPCOtype

  var InternetType newPTC :=
  InternetType.create;

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

  newPTC.start
  (internetUser());

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

  newPTC.start(internetUser());

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

  return;
}

function aGuest (in float eatingDur) runs on GuestType {
  timer T1;

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

Figure C.2: Restaurant example - newInternetPTC and aGuest functions
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 C.3: Restaurant example - newGuest function
function internetUser() runs on InternetType

self
InternetType

P1
dPCOtype

cPType

P2

iPCOtype

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) {
  if (getverdict == pass) {
    P1.receive(PizzaDeliveryType {?, orderNr, ?});
  }
  CP.send(ReportType : {internet, getverdict}); stop;
}

Figure C.4: Restaurant example - internetUser function
altstep GuestDefault() runs on GuestType {
    // *** Purpose: Default behaviour for
    // *** message based ports
    // ***
    [] P1.receive(charstring : ?) {
        P1.send(standardConversation);
        repeat;
    }
    [] any timer.timeout {
        setverdict(fail);
    }
    [] any port.receive {
        setverdict(inconc);
    }
}

Figure C.5: Restaurant example - GuestDefault and StandardDefault functions
function basicCapabilityTests() return verdicttype

localVerdict := execute (MyTestCase (true, false, false, 1, 1, nrP, nrF, nrI), 1800.0)

if (localVerdict == pass)
    localVerdict := execute (MyTestCase (false, true, false, 1, 1, nrP, nrF, nrI), 1800.0)

if (localVerdict == pass)
    localVerdict := execute (MyTestCase (false, false, true, 1, 1, nrP, nrF, nrI), 1800.0)
function loadTests() return verdicttype

self

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

if (localVerdict == pass)
localVerdict := execute (MyTestCase(true, true, true, 100, 10, nrP, nrF, nrI));

if (localVerdict == pass)
localVerdict := execute (MyTestCase(true, true, true, 400, 30, nrP, nrF, nrI));

if (localVerdict == pass)
localVerdict := execute (MyTestCase(true, true, true, 1000, 60, nrP, nrF, nrI));

return (localVerdict);

function serviceInterworkingTests() return verdicttype

self

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

if (localVerdict == pass)
localVerdict := execute (MyTestCase(true, true, false, 2, 2, nrP, nrF, nrI), 3000.0);

if (localVerdict == pass)
localVerdict := execute (MyTestCase(false, true, true, 2, 2, nrP, nrF, nrI), 3000.0);

if (localVerdict == pass)
localVerdict := execute (MyTestCase(true, false, true, 2, 2, nrP, nrF, nrI), 3000.0);

return (localVerdict);

Figure C.7: Restaurant example - loadTests and serviceInterworkingTests functions
```java
function qualityAssuranceTests() return verdicttype {
    self
    var verdicttype localVerdict := pass;
    var integer nrP := 0, nrF := 0, nrI := 0;

    execute (MyTestCase (true, true, true, 100, 10, nrP, nrF, nrI));
    if (nrF + nrI > 5) {
        localVerdict := fail;
    }

    execute (MyTestCase (true, true, true, 400, 30, nrP, nrF, nrI));
    if (nrF + nrI > 25) {
        localVerdict := fail;
    }

    execute (MyTestCase (true, false, true, 1000, 60, nrP, nrF, nrI));
    if (nrF + nrI > 75) {
        localVerdict := fail;
    }

    return (localVerdict);
}
```

Figure C.8: Restaurant example - qualityAssuranceTests

```java
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);
}
```
module PizzaHutTest {
  boolean capabilityTesting, interworkingTesting, loadTesting, qualityTesting;

  control {
    var verdicttype overallVerdict := pass;

    if (capabilityTesting and overallVerdict == pass) {
      overallVerdict := basicCapabilityTests();
    }

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

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

    if (qualityTesting and overallVerdict == pass) {
      overallVerdict := qualityAssuranceTests();
    }
  }
}

Figure C.9: Restaurant example - PizzaHutTest module
C.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:{} ) {

alt { /* end alt4 */
[ ] MSAP2.receive (Medium_Data_Transfer ) {
setverdict(inconclusive);stop;
}
[ ] MSAP2.receive( Medium_Disconnection_Request ){
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 alt3 */
}
}

Figure C.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 */
testcase mi_synch2 () runs on MTCType

alt ISAP1 MSAP2

PCO_Type1 PCO_Type2

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:IDISInd ) ; /* PASS */

}

}

}

}

}

}

ISAP1.receive ( IDISind:{}) {

setverdict(inconclusive);

}

}

med ( ISAP1.receive ( IDISind:IDISInd ) ) {

setverdict(inconclusive);

}

)

}

)

)

MSAP2.receive ( MDATInd:Medium_Connection_Request ) {

setverdict(inconclusive);

}

)

)

ISAP1.receive ( IDISInd:IDISInd ) {

setverdict(inconclusive);

}

)

)

)

)

)

}

}

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

)

) 

stop; /* End testcase mi_synch2 */
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 */
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 C.14: INRES example - DefaultWithInconclusives altstep
altstep OtherwiseFail() {
    with { display "default"; }
    ISAP1.receive {
        setverdict(fail); stop;
    }
    MSAP2.receive {
        setverdict(fail); stop;
    }
}

module InresExample1 {
    control InresExample {
        execute (mi_synch1(), 5.0);
        execute (mi_synch2(), 5.0);
        execute (mi_synch5(), 5.0);
    } // end control part
}

module InresExample1 {
    ...
    control InresExample {
        execute (mi_synch1(), 5.0);
        execute (mi_synch2(), 5.0);
        execute (mi_synch5(), 5.0);
    } // end control part

Figure C.15: INRES example - OtherwiseFail altstep and InresExample1 module definitions
## History

<table>
<thead>
<tr>
<th>Document history</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>V1.1.1</strong></td>
</tr>
<tr>
<td><strong>V1.1.2</strong></td>
</tr>
<tr>
<td><strong>V1.2.1</strong></td>
</tr>
<tr>
<td><strong>V2.2.1</strong></td>
</tr>
<tr>
<td><strong>V2.2.2</strong></td>
</tr>
<tr>
<td><strong>V3.1.1</strong></td>
</tr>
<tr>
<td><strong>V3.2.1</strong></td>
</tr>
</tbody>
</table>