Methods for Testing and Specification (MTS);
The Testing and Test Control Notation version 3;
Part 4: TTCN-3 Operational Semantics
Reference
RES/MTS-201873-4 T3 ed461 OS

Keywords
language, testing, TTCN

Important notice

The present document can be downloaded from:
http://www.etsi.org/standards-search

The present document may be made available in electronic versions and/or in print. The content of any electronic and/or
print versions of the present document shall not be modified without the prior written authorization of ETSI. In case of any
existing or perceived difference in contents between such versions and/or in print, the only prevailing document is the
print of the Portable Document Format (PDF) version kept on a specific network drive within ETSI Secretariat.

Users of the present document should be aware that the document may be subject to revision or change of status.
Information on the current status of this and other ETSI documents is available at

If you find errors in the present document, please send your comment to one of the following services:
https://portal.etsi.org/People/CommitteeSupportStaff.aspx

Copyright Notification

No part may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying
and microfilm except as authorized by written permission of ETSI.
The content of the PDF version shall not be modified without the written authorization of ETSI.
The copyright and the foregoing restriction extend to reproduction in all media.

© European Telecommunications Standards Institute 2017.
All rights reserved.

DECT™, PLUGTESTS™, UMTS™ and the ETSI logo are Trade Marks of ETSI registered for the benefit of its Members.
3GPP™ and LTE™ are Trade Marks of ETSI registered for the benefit of its Members and
of the 3GPP Organizational Partners.
oneM2M logo is protected for the benefit of its Members
GSM® and the GSM logo are Trade Marks registered and owned by the GSM Association.
## Contents

Intellectual Property Rights .................................................................................................................. 8
Foreword .................................................................................................................................................. 8
Modal verbs terminology ......................................................................................................................... 8

1 Scope .................................................................................................................................................. 9
2 References .......................................................................................................................................... 9
2.1 Normative references ..................................................................................................................... 9
2.2 Informative references .................................................................................................................. 9
3 Definitions and abbreviations ............................................................................................................. 9
3.1 Definitions ...................................................................................................................................... 9
3.2 Abbreviations .............................................................................................................................. 9

4 Introduction ........................................................................................................................................ 10
5 Structure of the present document ................................................................................................... 10
6 Restrictions ......................................................................................................................................... 10

7 Replacement of short forms ................................................................................................................ 11
7.0 General .......................................................................................................................................... 11
7.1 Order of replacement steps .......................................................................................................... 12
7.2 Replacement of global constants and module parameters ............................................................ 12
7.3 Embedding single receiving operations into alt statements .......................................................... 12
7.4 Embedding stand-alone altstep calls into alt statements .............................................................. 13
7.5 Replacement of interleave statements .......................................................................................... 13
7.6 Replacement of trigger operations ............................................................................................... 26
7.7 Replacement of select-case statements ......................................................................................... 26
7.8 Replacement of simple break statements ..................................................................................... 28
7.9 Replacement of continue statements ............................................................................................ 28
7.10 Adding default parameters to disconnect and unmap operations without parameters ............... 29
7.11 Adding default values of parameters ........................................................................................... 29

8 Flow graph semantics of TTCN-3 ....................................................................................................... 30
8.0 General .......................................................................................................................................... 30
8.1 Flow graphs .................................................................................................................................... 30
8.1.0 General ...................................................................................................................................... 30
8.1.1 Flow graph frame ....................................................................................................................... 30
8.1.2 Flow graph nodes ....................................................................................................................... 30
8.1.2.0 General ................................................................................................................................. 30
8.1.2.1 Start nodes ............................................................................................................................. 30
8.1.2.2 End nodes ............................................................................................................................ 31
8.1.2.3 Basic nodes .......................................................................................................................... 31
8.1.2.4 Reference nodes .................................................................................................................. 31
8.1.2.4.0 General ............................................................................................................................ 31
8.1.2.4.1 OR combination of reference nodes ............................................................................... 31
8.1.2.4.2 Multiple occurrences of reference nodes ....................................................................... 32
8.1.3 Flow lines ................................................................................................................................... 32
8.1.4 Flow graph segments ................................................................................................................ 33
8.1.5 Comments .................................................................................................................................. 34
8.1.6 Handling of flow graph descriptions .......................................................................................... 34
8.2 Flow graph representation of TTCN-3 behaviour ......................................................................... 35
8.2.0 General ...................................................................................................................................... 35
8.2.1 Flow graph construction procedure .......................................................................................... 35
8.2.2 Flow graph representation of module control .......................................................................... 36
8.2.3 Flow graph representation of test cases .................................................................................... 36
8.2.4 Flow graph representation of functions ..................................................................................... 37
8.2.5 Flow graph representation of altsteps ...................................................................................... 38
8.2.6 Flow graph representation of component type definitions ...................................................... 39
8.2.7 Retrieval of start nodes of flow graphs .......................................................... 40
8.3 State definitions for TTCN-3 modules ............................................................................. 40
8.3.0 General .................................................................................................................. 40
8.3.1 Module state ........................................................................................................... 41
8.3.1.0 General ............................................................................................................... 41
8.3.1.1 Accessing the module state ................................................................................. 41
8.3.1a Configuration state .................................................................................................. 41
8.3.1a.0 General ............................................................................................................... 41
8.3.1a.1 Accessing the configuration state ................................................................. 41
8.3.2 Entity states .............................................................................................................. 42
8.3.2.0 General ............................................................................................................... 42
8.3.2.1 Accessing entity states ........................................................................................ 44
8.3.2.2 Data state and variable binding ............................................................................. 45
8.3.2.3 Accessing data states ............................................................................................ 46
8.3.2.4 Timer state and timer binding ............................................................................. 46
8.3.2.5 Accessing timer states ........................................................................................ 47
8.3.2.6 Port references and port binding ................................................................. 48
8.3.2.7 Accessing port references ................................................................................. 49
8.3.3 Port states ................................................................................................................. 49
8.3.3.0 General ............................................................................................................... 49
8.3.3.1 Handling of connections among ports ......................................................... 50
8.3.3.2 Handling of port states ....................................................................................... 50
8.3.3a Component verdict states ...................................................................................... 51
8.3.4 General functions for the handling of module states ............................................. 51
8.4 Messages, procedure calls, replies and exceptions .................................................... 52
8.4.0 General .................................................................................................................. 52
8.4.1 Messages ............................................................................................................... 52
8.4.2 Procedure calls and replies .................................................................................... 52
8.4.3 Exceptions ............................................................................................................. 53
8.4.4 Construction of messages, procedure calls, replies and exceptions ..................... 53
8.4.5 Matching of messages, procedure calls, replies and exceptions ......................... 53
8.4.6 Retrieval of information from received items ...................................................... 54
8.5 Call records for functions, altsteps and test cases ...................................................... 54
8.5.0 General .................................................................................................................. 54
8.5.1 Handling of call records ......................................................................................... 54
8.6 The evaluation procedure for a TTCN-3 module ....................................................... 55
8.6.1 Evaluation phases .................................................................................................. 55
8.6.1.0 General ............................................................................................................... 55
8.6.1.1 Phase I: Initialization .......................................................................................... 55
8.6.1.2 Phase II: Update ................................................................................................ 56
8.6.1.3 Phase III: Selection ............................................................................................ 56
8.6.1.4 Phase IV: Execution .......................................................................................... 56
8.6.2 Global functions ..................................................................................................... 57
9 Flow graph segments for TTCN-3 constructs ............................................................... 57
9.0 General ..................................................................................................................... 57
9.1 Action statement ....................................................................................................... 57
9.2 Activate statement .................................................................................................... 58
9.2a Alive component operation ...................................................................................... 59
9.2a.0 General ............................................................................................................... 59
9.2a.1 Flow graph segment <alive-comp-act> ............................................................. 61
9.2a.2 Flow graph segment <alive-comp-snap> ........................................................... 62
9.3 Alt statement ............................................................................................................. 62
9.3.0 General ............................................................................................................... 62
9.3.1 Flow graph segment <take-snapshot> ................................................................. 64
9.3.2 Flow graph segment <receiving-branch> ............................................................ 65
9.3.3 Flow graph segment <altstep-call-branch> ....................................................... 66
9.3.4 Flow graph segment <else-branch> ................................................................. 67
9.3.5 Flow graph segment <default-evocation> ........................................................ 68
9.4 Altstep call ............................................................................................................... 69
9.5 Assignment statement ............................................................................................... 69
9.5a Break statements in altsteps .................................................................................. 69
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.49.1</td>
<td>Void</td>
<td>155</td>
</tr>
<tr>
<td>9.49.2</td>
<td>Flow graph segment &lt;stop-alive-component&gt;</td>
<td>155</td>
</tr>
<tr>
<td>9.49.3</td>
<td>Flow graph segment &lt;stop-all-comp&gt;</td>
<td>156</td>
</tr>
<tr>
<td>9.50</td>
<td>Stop execution statement</td>
<td>157</td>
</tr>
<tr>
<td>9.51</td>
<td>Stop port operation</td>
<td>158</td>
</tr>
<tr>
<td>9.52</td>
<td>Stop timer operation</td>
<td>159</td>
</tr>
<tr>
<td>9.53</td>
<td>System operation</td>
<td>159</td>
</tr>
<tr>
<td>9.53a</td>
<td>Test case stop operation</td>
<td>160</td>
</tr>
<tr>
<td>9.54</td>
<td>Timer declaration</td>
<td>160</td>
</tr>
<tr>
<td>9.54.0</td>
<td>General</td>
<td>160</td>
</tr>
<tr>
<td>9.54.1</td>
<td>Flow graph segment &lt;timer-decl-default&gt;</td>
<td>161</td>
</tr>
<tr>
<td>9.54.2</td>
<td>Flow graph segment &lt;timer-decl-no-def&gt;</td>
<td>161</td>
</tr>
<tr>
<td>9.55</td>
<td>Timeout timer operation</td>
<td>162</td>
</tr>
<tr>
<td>9.56</td>
<td>Unmap operation</td>
<td>163</td>
</tr>
<tr>
<td>9.56.0</td>
<td>General</td>
<td>163</td>
</tr>
<tr>
<td>9.56.1</td>
<td>Flow graph segment &lt;unmap-all&gt;</td>
<td>165</td>
</tr>
<tr>
<td>9.56.2</td>
<td>Flow graph segment &lt;unmap-comp&gt;</td>
<td>166</td>
</tr>
<tr>
<td>9.56.3</td>
<td>Flow graph segment &lt;unmap-port&gt;</td>
<td>167</td>
</tr>
<tr>
<td>9.57</td>
<td>Variable declaration</td>
<td>167</td>
</tr>
<tr>
<td>9.57.0</td>
<td>General</td>
<td>167</td>
</tr>
<tr>
<td>9.57.1</td>
<td>Flow graph segment &lt;var-declaration-init&gt;</td>
<td>168</td>
</tr>
<tr>
<td>9.57.2</td>
<td>Flow graph segment &lt;var-declaration-undef&gt;</td>
<td>169</td>
</tr>
<tr>
<td>9.58</td>
<td>While statement</td>
<td>169</td>
</tr>
<tr>
<td>10</td>
<td>Lists of operational semantic components</td>
<td>170</td>
</tr>
<tr>
<td>10.1</td>
<td>Functions and states</td>
<td>170</td>
</tr>
<tr>
<td>10.2</td>
<td>Special keywords</td>
<td>171</td>
</tr>
<tr>
<td>10.3</td>
<td>Flow graphs of TTCN-3 behaviour descriptions</td>
<td>172</td>
</tr>
<tr>
<td>10.4</td>
<td>Flow graph segments</td>
<td>172</td>
</tr>
<tr>
<td>History</td>
<td></td>
<td>175</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 (https://ipr.etsi.org/).

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 4 of a multi-part deliverable. Full details of the entire series can be found in part 1 [1].

NOTE: All formatting in the present document has been done intentionally. Underlined words denote special elements of the formal semantics. Their meaning is described in clauses 7 and 8.

Modal verbs terminology

In the present document "shall", "shall not", "should", "should not", "may", "need not", "will", "will not", "can" and "cannot" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"must" and "must not" are NOT allowed in ETSI deliverables except when used in direct citation.
1 Scope
The present document defines the operational semantics of TTCN-3. The present document is based on the TTCN-3 core language defined in ETSI ES 201 873-1 [1].

2 References

2.1 Normative references
References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found at https://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.

The following referenced documents are necessary for the application of the present document.


2.2 Informative references
References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

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

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

Not applicable.

3 Definitions and abbreviations

3.1 Definitions
For the purposes of the present document, the terms and definitions given in ETSI ES 201 873-1 [1] apply.

3.2 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>MTC</td>
<td>Master Test Component</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
</tr>
<tr>
<td>TTCN</td>
<td>Testing and Test Control Notation</td>
</tr>
</tbody>
</table>
4 Introduction

This clause defines the meaning of TTCN-3 behaviour in an intuitive and unambiguous manner. The operational semantics is not meant to be formal and therefore the ability to perform mathematical proofs based on this semantics is very limited.

This operational semantics provides a state oriented view on the execution of a TTCN module. Different kinds of states are introduced and the meaning of the different TTCN-3 constructs is described by:

1) using state information to define the preconditions for the execution of a construct; and

2) defining how the execution of a construct will change a state.

The operational semantics is restricted to the meaning of behaviour in TTCN-3, i.e. functions, altsteps, test cases, module control and language constructs for defining test behaviour, e.g. `send` and `receive` operations, `if-else-`, or `while`- statements. The meaning of some TTCN-3 constructs is explained by replacing them with other language constructs. For example, `interleave` statements are short forms for series of nested `alt` statements and the meaning of each `interleave` statement is explained by its replacement with a corresponding series of nested `alt` statements.

In most cases, the definition of the semantics of a language is based on an abstract syntax tree of the code that may be described. This semantics does not work on an abstract syntax tree but requires a graphical representation of TTCN-3 behaviour descriptions in form of flow graphs. A flow graph describes the flow of control in a function, alt step, test case or the module control. The mapping of TTCN-3 behaviour descriptions onto flow graphs is straightforward.

**NOTE:** The mapping of TTCN-3 statements onto flow graphs is an informal step and is not defined by using the BNF rules in ETSI ES 201 873-1 [1]. The reason for this is that the BNF rules are not optimal for an intuitive mapping because several static semantic rules are coded into BNF rules in order to allow static semantic checks during the syntax check.

5 Structure of the present document

The present document is structured into four parts:

1) The first part (see clause 6) describes restrictions of the operational semantics, i.e. issues related to the semantics, which are not covered by the present document.

2) The second part (see clause 8) defines the meaning of TTCN-3 short cut and macro notations by their replacement with other TTCN-3 language constructs. These replacements in a TTCN-3 module can be seen as pre-processing step before the module can be interpreted according to the following operational semantics description.

3) The third part (see clause 9) describes the operational semantics of TTCN-3 by means of flow graph interpretation and state modification.

4) The fourth part (see clause 10) specifies the mapping of the different TTCN-3 statements onto flow graph segments, which provide the building blocks for flow graphs representing functions, alt steps, test cases and module control.

6 Restrictions

The operational semantics only covers behavioural aspects of TTCN-3, i.e. it describes the meaning of statements and operations. It does not provide:

a) A semantics for the data aspects of TTCN-3. This includes aspects like encoding, decoding and the usage of data imported from non-TTCN-3 specifications.

b) A semantics for the grouping mechanism. Grouping is related to the definitions part of a TTCN-3 module and has no behavioural aspects.
c) A semantics for the **import** statement. The import of definitions has to be done in the definitions part of a TTCN-3 module. The operational semantics handles imported definitions as if they are defined in the importing module.

d) A semantics for the visibility of definitions. The correct usage of imported definitions declared with **public**, **private** and **friend** visibility has to be checked by other means.

e) A semantics for fuzzy and lazy evaluation of variables and parameters. However, notes in the appropriate clauses of this standard refer to places where fuzzy and lazy evaluation has to be considered.

### 7 Replacement of short forms

#### 7.0 General

Short forms have to be expanded by the corresponding complete definitions on a textual level before this operational semantics can be used for the explanation of TTCN-3 behaviour.

TTCN-3 short forms are:

- lists of module parameter, constant and variable declarations of the same type and lists of timer declarations;
- stand-alone receiving operations;
- stand-alone altsteps calls;
- **trigger** operations;
- missing **return** and **stop** statements at the end of function and test case definitions;
- missing **stop** execution statements;
- **interleave** statements;
- **select-case** statements;
- **break** and **continue** statements;
- **disconnect** and **unmap** operations without parameters; and
- default values of missing actual parameters.

In addition to the handling of short forms, the operational semantics requires a special handling for module parameters, global constants, i.e. constants that are defined in the module definitions part, and pre-processing macros. All references to module parameters, global constants and pre-processing macros shall be replaced by concrete values. This means, it is assumed that the value of module parameters, global constants and pre-processing macros can be determined before the operational semantics becomes relevant.

NOTE 1: The handling of module parameters and global constants in the operational semantics will be different from their handling in a TTCN-3 compiler. The operational semantics describes the meaning of TTCN-3 behaviour and is not a guideline for the implementation of a TTCN-3 compiler.

NOTE 2: The operational semantics handles parameters of and local constants in test components, test cases, functions and module control like variables. The wrong usage of local constants or **in**, **out** and **inout** parameters has to be checked statically.
7.1 Order of replacement steps

The textual replacements of short forms, global constants and module parameters have to be done in the following order:

1) replacement of lists of module parameter, constant, variable and timer declarations with individual declarations;

2) replacement of global constants and module parameters by concrete values;

3) replacement of all select-case statements by equivalent nested if-else statements;

4) embedding stand-alone receiving operations into alt statements;

5) embedding stand-alone altstep calls into alt statements;

6) expansion of interleave statements;

7) replacement of all trigger operations by equivalent receive operations and repeat statements;

8) adding return at the end of functions without return statement, adding self.stop operations at the end of testcase definitions without a stop statement;

9) adding stop at the end a module control part without stop statement;

10) expansion of break statements;

11) expansion of continue statements;

12) adding default parameters to disconnect and unmap operations without parameters; and

13) adding default values of parameters.

NOTE: Without keeping this order of replacement steps, the result of the replacements would not represent the defined behaviour.

7.2 Replacement of global constants and module parameters

Constants declared in the module definitions part are global for module control and all test components that are created during the execution of a TTCN-3 module. Module parameters are meant to be global constants at run-time.

All references to global constants and module parameters shall be replaced by the actual values before the operational semantics starts the interpretation of the module. If the value of a constant or module parameter is given in form of an expression, the expression has to be evaluated. Then, the result of the evaluation shall replace all references of the constant or module parameter.

7.3 Embedding single receiving operations into alt statements

TTCN-3 receiving operations are: receive, trigger, getcall, getreply, catch, check, timeout, and done.

NOTE: The operations receive, trigger, getcall, getreply, catch and check operate on ports and they allow branching due to the reception of messages, procedure calls, replies and exceptions. The operations timeout and done are not real receiving operations, but they can be used in the same manner as receiving operations, i.e. as alternatives in alt statements. Therefore, the operational semantics handles timeout and done like receiving operations.

A receiving operation can be used as stand-alone statement in a function, an altstep or a test case. The timeout operation can also be used as stand-alone statement in module control. In such a case the receiving operation as considered to be shorthand for an alt statement with only one alternative defined by the receiving operation. For the operational semantics an alt statement in which the receiving statement is embedded shall replace all stand-alone occurrences of receiving operations.
EXAMPLE:

```
// The stand-alone occurrence of
: MyCL.trigger(MyType:?);
:
// shall be replaced by
: alt {
    [] MyCL.trigger (MyType:?) { }
}
:
// or
: MyPTC.done;
:
// shall be replaced by
: alt {
    [] MyPTC.done { }
}
:
```

### 7.4 Embedding stand-alone altstep calls into alt statements

TTCN-3 allows calling altsteps like functions in functions, altsteps, test cases and module control. The meaning of a stand-alone call of an altstep is given by an `alt` statement with one branch only that calls the altstep. The `alt` statement is responsible for the snapshot that is evaluated within the altstep and for the invocation of the default mechanism if none of the alternatives in the altstep can be chosen.

**NOTE:** An altsteps used in module control can only include alternatives with `timeout` operations and an `else` branch.

EXAMPLE:

```
// The stand-alone occurrence of
: myAltstep(MyPar1Val);
:
// shall be replaced by
: alt {
    [] myAltstep(MyPar1Val) { }
}
:
```

### 7.5 Replacement of interleave statements

The meaning of an `interleave` statement is defined by its replacement by a series of nested `alt` statements that has the same meaning. The algorithm for the construction of the replacement for an `interleave` statement is described in this clause. The replacement shall be made on a syntactical level.

Within an `interleave` statement it is not allowed:

1) to use the control transfer statements `for, while, do-while, goto, activate, deactivate, stop, repeat` and `return`;
2) to call altsteps;
3) to call user-defined functions which include communication operations;
4) to guard branches of the `interleave` statement with Boolean expressions; and
5) to specify `else` branches.
Due to these restrictions, all not mentioned stand-alone statements (e.g. assignment, log, send or reply), blocking call operations and the compound statements interleave, if-else and alt can be used within an interleave statement.

NOTE 1: Blocking call operations and if-else statements can be treated like stand-alone statements if they have no embedded alt statements. In case of embedded alt statements, the alternatives contribute to the interleave statement and need a special handling. For simplicity, the algorithm below does not distinguish between these two cases.

NOTE 2: Non-blocking call operations are also allowed in interleave statements, they are considered to be stand-alone statements.

The algorithm described in this clause only works for interleave statements without embedded interleave statements. In case of an interleave statement that has embedded interleave statements, the embedded interleave statements have to be replaced before the algorithm can be applied.

NOTE 3: Due to restrictions 1 to 5, it is always possible to find finite replacements for nested embeddings of interleave statements.

The replacement algorithm works on a graph representation of an interleave statement and transforms it into a semantically equivalent tree structure describing a series of nested alt statements. For this, a graph representation of stand-alone statements, the compound statements if-else, blocking call, alt and interleave is needed.

A stand-alone statement is described by a node with the statement as inscription. A sequence of stand-alone statements is described by a set of nodes connected by a flow lines. This is shown in figure 1.

<table>
<thead>
<tr>
<th>P1.send(MyVar);</th>
<th>P1.send(MyVar);</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) TTCN-3 stand-alone statement</td>
<td>(b) graph representation of (a)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P1.send(MyVar);</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := 7 + 5;</td>
</tr>
<tr>
<td>(c) Sequence of TTCN-3 stand-alone statements</td>
</tr>
</tbody>
</table>

Figure 1: Graph representation of TTCN-3 stand-alone statements

The graph representation of an if-else statement is shown in figure 2. An if-else statement is represented by an IF node with two flow lines connected to the first statement in the two alternatives. An if-else statement without ELSE branch is represented in the same manner, if there are statements following the if-else statement. In this case the flow line representing the else branch is connected to the first statement following the if-else statement. An if-else statement without ELSE branch and without following statements is represented by an IF node with one flow line only.

NOTE 4: The inscriptions on the flow lines in figure 1 are introduced for readability purposes only. The algorithm only uses the relation expressed by the flow line and not the inscription.
if \( (x < 7) \) {
  P1.\_send(MyVar);
} else {
  x := 7 + 5;
} 
\( x := x \times 2 \)

(a) TTCN-3 if-else statement

IF 
\( (x < 7) \)
ELSE 
\( x := 7 + 5; \)

(b) Graph representation of (a)

if \( (x < 7) \) {
  P1.\_send(MyVar);
} \( x := x \times 2 \)

(c) TTCN-3 if-else statement without else branch

(d) Graph representation of (c)

Figure 2: Graph representation of TTCN-3 if-else statements

The graph representation of a blocking call statement is shown in figure 3. A blocking call statement is represented by a BLOCKING-CALL node with flow lines connected to the getreply and catch statements of the different alternatives.

```plaintext
P1.\_call(MyProc:\{-,true\}, 20E-3) {
  [] P1.\_getReply(MyProc:\{?,-\} {
    setverdict(pass);
  }
  [] P1.\_catch(MyProc, MyException) {}
  [] P1.\_catch(timeout) {
    setverdict(fail);
  }
} 
\( x := 7 + 5; \)
```

(a) TTCN-3 blocking call statement

(b) Graph representation of (a)

Figure 3: Graph representation of a TTCN-3 blocking call statement
The graph representation of an `alt` statement is shown in figure 4. An `alt` statement is represented by an `alt-node` with several flow lines connected to the different alternatives.

```
alt {
  [x<5] P1.receive(MyMessageOne) {
    setverdict(pass);
  }
  [] P1.receive(MyMessageTwo) {}
  [] T1.timeout {
    setverdict(fail);
  }
}
x := 7 + 5;
```

(a) TTCN-3 alt statement

```
[\text{x<5}]
\text{P1.receive(MyMessageOne)} & \text{P1.receive(MyMessageTwo)} & \text{T1.timeout}
\text{setverdict(pass);} & \text{setverdict(fail);} \\
\text{x := 7 + 5;}
```

(b) Graph representation of (a)

Figure 4: Graph representation of a TTCN-3 alt statement

In general, the graph representations of `if-else`, `blocking call` and `alt` statements are directed graphs without loops where the flow lines of the different alternatives join when leaving the statement. By means of duplication, it is possible to transform such directed graphs into a semantically equivalent tree representation. This is shown in figure 5 for the alt statement in figure 4. The algorithm described below will construct such tree representations.
alt {
  [x<5] P1.receive(MyMessageOne) {
    setverdict(pass);
    x := 7 + 5;
  }
  [] P1.receive(MyMessageTwo) {
    x := 7 + 5;
  }
  [] T1.timeout {
    setverdict(fail);
    x := 7 + 5;
  }
}

(a) TTCN-3 alt statement that is semantically equivalent to figure 4(a)

(b) Graph representation of (a) (semantically equivalent to figure 4(b))

Figure 5: Graph representation of a TTCN-3 alt statement
An `interleave` statement can be described by a graph that consists of a set of directed sub-graphs, each of which is constructed by means of stand-alone statements and the compound statements `if-else`, `blocking call` and `alt`. The directed sub-graphs describe the interleaved flows of control. An example is shown in figure 6. The node inscriptions in figure 6(b) refer to the labels of the TTCN-3 statements in figure 6(a).

```
interleave {
    [] P1.receive(M1) { // L1
        alt {
            [] P1.receive(M3) { // L2
                setverdict(pass); // L3
            }
            [] T1.timeout { } // L4
        }
    }
    [] P2.receive(M2) { // L5
        if (x < 5 ) {
            alt {
                [] P2.receive(M4) { // L6
                    setverdict(pass); // L7
                }
                [] Compl.done { } // L8
            }
            x := 7 + 5; // L9
        } else {
            P3.call(MyProcTempl, 20E-3) { // BC (= BLOCKING CALL)
                [] P3.getreply(ReplyTempl) { // L10
                    alt {
                        [] P2.receive(M5) { } // L11
                        [] P2.receive(M6) { } // L12
                    }
                }
                [] P3.catch(timeout) { // L13
                    setverdict(fail); // L14
                }
            }
        }
    }
}
```

(a) TTCN-3 interleave statement

(b) Graph representation of (a)

Figure 6: Graph representation of a TTCN-3 interleave statement
Formally, an **interleave** statement can be described by a graph $GI = (St, F)$ where:

- $St$ is the set of allowed TTCN-3 statements; and
- $F \subseteq (St \times St)$ describes the flow relation.

The term **allowed TTCN-3 statements** refers to the static restrictions 1-5 above.

For the construction algorithm the following functions need to be defined:

- **The $REACHABLE$ function** returns all statements that are reachable from a statement $s$ in a graph $GI = (St, F)$:
  $$REACHABLE\ (s, GI) = \{ s \} \cup \{ stmt | stmt \in St \land \exists (s = x_1, x_2, \ldots, x_n = stmt) \text{ where } x_i \in St, \ i \in [1, n] \land (x_i, x_{i+1}) \in F \}$$

- **The $SUCCESSORS$ function** returns all successors of a statement $s$ in a graph $GI = (St, F)$:
  $$SUCCESSORS\ (s, GI) = \{ stmt | stmt \in St \land (s, stmt) \in F \}$$

- **The $ENABLED$ function** returns all statements of a graph $GI = (St, F)$ which have no predecessors:
  $$ENABLED\ (GI) = \{ stmt | stmt \in St \land (F \cap (S \times \{ s \}) = \emptyset) \}$$

- **The $KIND$ function** returns the kind or type of a TTCN-3 statement in a graph representing an **interleave** statement.

- **The $DISCARD$ function** deletes a statement $s$ or a set of statements $S$ from a graph $GI = (St, F)$ and returns the resulting graph $GI' = (St', F')$:
  
  For single nodes:
  $$DISCARD\ (s, GI) = GI' \text{ where: } GI' = (St', F'), \text{ with } St' = St \setminus \{ s \} \text{ and } F' = F \cap (St \setminus \{ s \} \times St \setminus \{ s \})$$

  For sets of nodes:
  $$DISCARD\ (S, GI) = GI' \text{ where: } GI' = (St', F'), \text{ with } St' = St \setminus S \text{ and } F' = F \cap (St \setminus S \times St \setminus S)$$

- **The $RECEIVING$ function** takes a set of statements of a graph $GI$ and returns all receiving statements:
  $$RECEIVING\ (S) = \{ stmt | stmt \in St \land KIND(stmt) \in \{ receive, trigger, getcall, getreply, catch, check, done, timeout \} \}$$

- **The $RANDOM$ function** selects randomly an element $s$ from a given set $S$ and returns $s$.
  $$RANDOM\ (S) = s \text{ where } s \in S$$

The construction algorithm (see figure 7) of the tree is a recursive procedure where in each recursive call the successor nodes for a given node is constructed. The procedure is provided in a C-like pseudo-code notation that uses the functions defined above and some additional mathematical notation.
CONSTRUCT-SUCCESSORS (statementType *predecessor, graphType GI) {
    // - statementType refers to the type of a node of the tree that is constructed
    // - *predecessor refers to the last node that has been created
    // - graphType denotes type of the graph of TTCN-3 statements
    // - GI is called by value and refers to the subgraph consisting of all remaining TTCN-3
    // statements that have to be taken into consideration

    var graphType myGraph;
    var statementType i, myStmt;
    var statementType *newStmt, *firstInBranch; // pointers for new statement nodes in the
    // tree that is constructed recursively

    // Retrieving sets of TTCN-3 statements that have no predecessors in 'GI'
    var statementSet enabStmts := ENABLED(GI);   // all statements without predecessor
    var statementSet enabRecStmts := RECEIVING(enabStmts); // receiving statements in 'enabStmts'
    var statementSet enabNonRecStmts := enabStmts\enabRecStmts;
    // non receiving statements in 'enabStmts'

    if (GI.St == {}) { // It is assumed that GI.St refers to the set of statements in GI
        return;   // No statements are left, termination criterion of Recursion
    }
    elseif (enabNonRecStmts != {}) { // Handling of non receiving statements in 'enabStmts'
        myStmt := RANDOM(enabNonRecStmts);
        // There can only be one statement in 'enabNonRec', because the Algorithm
        // continues the construction until there is a branch that contributes to
        // the interlave statement.
        newStmt := create(myStmt, predecessor);
        // Creation of a new tree node representing 'myStmt' in the tree
        // and update of pointers in 'newStmt' and 'predecessor'.
        if (KIND(myStmt) == IF ||
            KIND(myStmt) == BLOCKING_CALL) {
            for each i in SUCCESSORS(myStmt, GI) {
                firstInBranch := create(i, newStmt);
                // Creation of a second node for the first statement of in a branch due to
                // an if-else statement.
                // Note, this create statement will be used to create tree nodes
                // representing the receiving statements in blocking call operations.
                myGraph := DISCARD({i, myStmt} \ REACHABLE(myStmt, GI)\REACHABLE(i, GI))
                // Removal of i, myStmt and all statements that are reachable from
                // myStmt but not reachable from i. The latter considers the branching of
                // a flow of control in a subgraph of GI.
                CONSTRUCT-SUCCESSORS(firstInBranch, myGraph);  // NEXT RECURSION STEP
            }
        }
        else { // mystmt is a stand-alone statement
            CONSTRUCT-SUCCESSORS(newSonNode, DISCARD(myStmt, GI)); // NEXT RECURSION STEP
        }
    }
    else { // Handling of receiving events that interleave
        if (KIND(predecessor) != ALT) { // an alt node is missing and has to be created, if the
            predecessor := create(ALT, predecessor);
        }
        for each i in enabRecStmts) {
            newStmt := create(i, predecessor); // New tree node
            CONSTRUCT-SUCCESSORS(newStmt, DISCARD(i, GI)); // NEXT RECURSION STEP (G)
        }
    }
}

Figure 7: Replacement algorithm for TTCN-3 interleave statements
Initially, the CONSTRUCT-SUCCESSORS function (see figure 7) will be called with a root node of an empty tree and the graph of TTCN-3 statements describing the **interleave** statement that shall be replaced. After termination, the root node can be used to access the constructed tree.

The application of the CONSTRUCT-SUCCESSORS function to the **interleave** statement shown in figure 6 leads to the tree shown in figure 8. The labels refer to the statements in figure 6(a). Multiple labels are the result of the duplication of code. The TTCN-3 code that corresponds to the tree in figure 8 is shown in figure 9.

**NOTE 5:** The example for the application of the algorithm in figure 7 (see figures 6, 8 and 9) is very comprehensive. This example is provided in order to show most of the special situations, i.e. branching and joining of flow lines, an embedded **alt** statement, a **blocking call** statement and an **if-else** statement.
Figure 8: Result of applying the algorithm in figure 7 to the interleave statement in figure 6
alt { }
   [] P1.receive(M1) { }
      alt { }
         [] P1.receive(M3) { }
            setverdict(pass); 
            alt { }
               [] P2.receive(M2) { }
                  if (x < 5) { }
                     alt { }
                        [] P2.receive(M4) { }
                           setverdict(pass); 
                           x := 7 + 5; 
                           L9 }
                        [] Comp1.done { }
                           x := 7 + 5; 
                           L9 }
               [] Comp1.done { }
                  L8 }
               [] Comp1.done { }
                  L8 }
         [] Comp1.done { }
            L8 }
   [] T1.timeout { }
      alt { }
         [] P2.receive(M2) { }
            if (x < 5) { }
               alt { }
                  [] P2.receive(M4) { }
                     setverdict(pass); 
                     x := 7 + 5; 
                     L9 }
                  [] Comp1.done { }
                     x := 7 + 5; 
                     L9 }
               else { }
                  P3.call(MyProcTempl, 20E-3) { // BC (= BLOCKING CALL) 
                     [] P3.getreply(ReplyTempl) { }
                        alt { }
                           [] P2.receive(M5) { }
                              // L11 }
                           [] P2.receive(M6) { }
                              // L12 }
                        [] P3.catch(timeout) { }
                           // L13 }
                              setverdict(fail); 
                           // L14 }
               [] Comp1.done { }
                  L8 }
               [] Comp1.done { }
                  L8 }
         [] Comp1.done { }
            L8 }
      [] Comp1.done { }
         x := 7 + 5; 
         alt { }
            [] P1.receive(M3) { }
               setverdict(pass); 
               L3 }
            [] T1.timeout { }
               L4 }
         [] Comp1.done { }
            L8 }
         [] Comp1.done { }
            L8 }
      [] Comp1.done { }
         x := 7 + 5; 
         alt { }
            [] P1.receive(M3) { }
               setverdict(pass); 
               L3 }
            [] T1.timeout { }
               L4 }
      [] P1.receive(M3) { }
        setverdict(pass); 
        alt { }
           [] P2.receive(M4) { }
              setverdict(pass); 
              x := 7 + 5; 
              L9 }
           [] Comp1.done { }
              x := 7 + 5; 
              L9 }
         [] Comp1.done { }
            L8 }
         [] Comp1.done { }
            L8 }
      [] Comp1.done { }
         x := 7 + 5; 
         alt { }
            [] P1.receive(M3) { }
               setverdict(pass); 
               L3 }
            [] T1.timeout { }
               L4 }
      [] P1.receive(M3) { }
        setverdict(pass); 
        alt { }
           [] P2.receive(M4) { }
              setverdict(pass); 
              x := 7 + 5; 
              L9 }
           [] Comp1.done { }
              x := 7 + 5; 
              L9 }
      [] Comp1.done { }
         x := 7 + 5; 
         alt { }
            [] P1.receive(M3) { }
               setverdict(pass); 
               L3 }
            [] T1.timeout { }
               L4 }
      [] P1.receive(M3) { }
        setverdict(pass); 
        alt { }
           [] P2.receive(M4) { }
              setverdict(pass); 
              x := 7 + 5; 
              L9 }
           [] Comp1.done { }
              x := 7 + 5; 
              L9 }
      [] Comp1.done { }
         x := 7 + 5; 
         alt { }
            [] P1.receive(M3) { }
               setverdict(pass); 
               L3 }
            [] T1.timeout { }
               L4 }
      [] P1.receive(M3) { }
        setverdict(pass); 
        alt { }
           [] P2.receive(M4) { }
              setverdict(pass); 
              x := 7 + 5; 
              L9 }
           [] Comp1.done { }
              x := 7 + 5; 
              L9 }
[] T1.timeout { // L4
  alt { // ALT
    [] P2.receive(M4) { // L6
      setverdict(pass); // L7
      x := 7 + 5; // L9
    }
    [] Compl.done { // L8
      x := 7 + 5; // L9
    }
  }
}
else { // BC (= BLOCKING CALL)
P3.call(MyProcTempl, 20E-3) { // L10
  [] P3.getreply(ReplyTempl) { // ALT
    [] P1.receive(M3) { // L2
      setverdict(pass); // L3
      alt { // ALT
        [] P2.receive(M5) { // L11
          [] P2.receive(M6) { // L12
          }
        }
        [] T1.timeout { // L4
          alt { // ALT
            [] P2.receive(M5) { // L11
              [] P1.receive(M3) { // L2
                setverdict(pass); // L3
              }
            }
            [] T1.timeout { // L4
              [] P2.receive(M6) { // L12
                [] P1.receive(M3) { // L2
                  setverdict(pass); // L3
                }
              }
            }
          }
        }
      }
    }
  }
}
[] P2.receive(M2) { // L5
  if (x < 5) { // IF
    alt {
      [] P2.receive(M4) { // L6
        setverdict(pass); // L7
        x := 7 + 5; // L9
        alt { // ALT
          [] P1.receive(M1) { // L10
            alt { // ALT
              [] P1.receive(M3) { // L2
                setverdict(pass); // L3
              }
            }
            [] T1.timeout { // L4
            }
          }
        }
      }
      [] Compl.done { // L8
        x := 7 + 5; // L9
        alt { // ALT
          [] P1.receive(M1) { // L10
            alt { // ALT
              [] P1.receive(M3) { // L2
                setverdict(pass); // L3
              }
            }
            [] T1.timeout { // L4
            }
          }
        }
      }
    }
  }
}
Figure 9: Semantically equivalent TTCN-3 code for the interleave statement in figure 6
7.6 Replacement of trigger operations

The trigger operation filters messages with a certain matching criterion from a stream of messages on a given port. The semantics of the trigger operation can be described by its replacement with two receive operations and a goto statement. The operational semantics assumes that this replacement is done on the syntactical level.

EXAMPLE 1:

// The following trigger operation ...

alt {
    [] MyCL.trigger (MyType:?) {  }
}

// shall be replaced by ...

alt {
    [] MyCL.receive (MyType:?) {  }
    [] MyCL.receive {
        repeat
    }
}

If the trigger statement is used in a more complex alt statement, the replacement is done in the same manner.

EXAMPLE 2:

// The following alt statement includes a trigger statement ...

alt {
    [] PCO2.receive {
        stop;
    }
    [] MyCL.trigger (MyType:?) {  }
    [] PCO3.catch {
        setverdict(fail);
        stop;
    }
}

// which will be replaced by

alt {
    [] PCO2.receive {
        stop;
    }
    [] MyCL.receive (MyType:?) {  }
    [] MyCL.receive {
        repeat;
    }
    [] PCO3.catch {
        setverdict(fail);
        stop;
    }
}

7.7 Replacement of select-case statements

The select-case statement is an alternative to using a set of nested if-else statements when comparing a value (defined by a select-expression following the select keyword) to one or several other values (defined by template instances in the case branches). Therefore, the semantics of a select-case statement can be described by its replacement with a set of nested if-else statements. To avoid a multiple evaluation of the select-expression, the set of nested if-else statements has to be placed into a statement block and value of the expression has to be stored in a variable at the beginning of the statement block. The operational semantics assumes that this replacement is done on the syntactical level.
Schematically the **select-case** statement looks as follows:

```plaintext
select (<expression>) {
    case (<templateInst_{1a}>, …, <templateInst_{1n}>)
        <statementblock_1>
    case (<templateInst_{2a}>, …, <templateInst_{2n}>)
        <statementblock_2>
    …
    case (<templateInst_{xa}, …, <templateInst_{xn}>)
        <statementblock_x>
    case else
        <statementblock_{x+1}>
}
```

The syntactical replacement of the schematic **select-case** statement by nested **if-else** statements looks as follows:

```plaintext
{
    var <expression>Type myTempVar := <expression>; // temporary variable for storing the
    // value of the expression

    if (match(myTempVar, <templateInst_{1a}>) or … or match(myTempVar, <templateInst_{1n}>))
        <statementblock_1>
    else {
        if (match(myTempVar, <templateInst_{2a}>) or … or match(myTempVar, <templateInst_{2n}>))
            <statementblock_2>
        else {
            …
            if (match(myTempVar, <templateInst_{xa}>) or … or match(myTempVar, <templateInst_{xn}>)
                <statementblock_x>
            else
                <statementblock_{x+1}>
        …
    }
}
```

**EXAMPLE:**

```plaintext
// The following select-case statement:
select (MyModulePar) { // where MyModulePar is of charstring type
    case ("firstValue") {
        log ("The first branch is selected");
    }
    case (MyCharstringVar, MyCharstringConst) {
        log ("The second branch is selected");
    }
    case else {
        log ("The else branch is selected");
    }
}

// is semantically equivalent to:
{
    var charstring myTempVar := MyModulePar;
    if (match(myTempVar, "firstValue") {
        log ("The first branch is selected");
    } else {
        if (match(myTempVar, MyCharstringVar) or match(myTempVar, MyCharstringConst)) {
            log ("The second branch is selected");
        } else {
            log ("The else branch is selected");
        }
    }
}
7.8 Replacement of simple break statements

“Simple” break statements are break statements used for leaving loops, interleave statements and alt statements. Such simple break statements are considered to be a short-hand form for using a pair of goto-label statements. For each break statement a label statement is added after the loop, alt statement or expanded interleave statement. The label statement shall have an unused label. The break statement is replaced by a goto statement to this specific label.

Note, that interleave statements are explained already. Therefore the limitation that goto statements cannot be used within interleave statements does not hold.

NOTE: The semantics for the break statement used to leave an altstep is defined in clause 9.5a.

EXAMPLE:

```c
// The following loop with a break statement:
while (cond1) { // cond1 is a Boolean condition guarding the loop
    ... if(cond2) {
        break;
    };
    ...
} // is semantically equivalent to:
while (cond1) { // cond1 is a Boolean condition guarding the loop
    ... if(cond2) {
        goto break_12345; // break_12345 is a unique label
    };
    ...
label break_12345;
```

7.9 Replacement of continue statements

The continue statement is a short-hand form for using a pair of goto-label statements. For each continue statement a label statement is added at the end of the loop body. The label statement shall have an unused label. The continue statement is replaced by a goto statement to this specific label.

EXAMPLE:

```c
// The following loop with a continue statement:
while (cond1) { // cond1 is a Boolean condition guarding the loop
    ... if(cond2) {
        continue;
    };
    ...
} // is semantically equivalent to:
while (cond1) { // cond1 is a Boolean condition guarding the loop
    ... if(cond2) {
        goto continue_12345; // continue_12345 is a unique label
    };
    ...
label continue_12345;
```
7.10 Adding default parameters to disconnect and unmap operations without parameters

The usage of a `disconnect` or `unmap` operation without any parameters is a shorthand form for using the operation with the parameter `self:all port`. It disconnects or unmaps all ports of the component that calls the operation. For the operational semantics the parameter `self:all port` shall be added to all occurrences of `disconnect` and `unmap` operations without parameters.

**EXAMPLE:**

```c
// each occurrence of disconnect;
// shall be expanded in the following manner:
disconnect(self:all port);
// and
// each occurrence of unmap;
// shall be expanded in the following manner:
unmap(self:all port);
```

7.11 Adding default values of parameters

Formal parameters may have default values. If no actual parameter is provided in a specific invocation, then the default value is added to the actual parameter list. If list notation has been used for the actual parameter list, then the default value is inserted according to the order in the formal parameter list. If assignment notation has been used for the actual parameter list, then the default values are appended to the actual parameters, the order among the default values corresponds to their order in the formal parameter list.

**EXAMPLE 1:**

```c
function f_comp (\_integer p_int1, \_integer p_int2 := 3) return integer {
    var integer v := p_int1 + p_int2;
    return v;
}
```

// Each occurrence of f_comp(1)
// shall be expanded to
f_comp(1, 3);

// Each occurrence of
f_comp(p_int1 := 1)
// shall be expanded to
f_comp(p_int1 := 1, p_int2 := 3);

**EXAMPLE 2:**

```c
function f(T x := E) runs on C {
    S1;
    ...
    Sn
}
```

// is semantically equivalent to

```c
function f(T x) runs on C {
    if (not isbound(x)) { x := E }
    S1;
    ...
    Sn
}
```
8 Flow graph semantics of TTCN-3

8.0 General

The operational semantics of TTCN-3 is based on the interpretation of flow graphs. In this clause flow graphs are introduced (see clause 8.1), the construction of flow graphs representing TTCN-3 module control, test cases, altsteps, functions and component type definitions is explained (see clause 8.2), module and component states for the description of the execution states of a TTCN-3 module are defined (see clause 8.3), the handling of messages, remote procedure calls, replies to remote procedure calls and exceptions is described (see clause 8.4) and the evaluation procedure of module control and test cases is explained (see clause 8.6).

8.1 Flow graphs

8.1.0 General

A flow graph is a directed graph that consists of labelled nodes and labelled edges. Traversing a flow graph describes the possible flow of control during the execution of a represented behaviour description.

8.1.1 Flow graph frame

A flow graph shall be put into a frame defining the border of the flow graph. The name of flow graph follows the keywords `flow graph` (these are not TTCN-3 core language keywords) and shall be put into the upper left corner of the flow graph. As convention it is assumed that the flow graph name refers to the TTCN behaviour description represented by the flow graph. A simple flow graph is shown in figure 10.

```
flow graph
MySimpleFlowGraph
\downarrow
inscription
\downarrow
```

Figure 10: A simple flow graph

8.1.2 Flow graph nodes

8.1.2.0 General

Flow graphs consist of `start nodes`, `end nodes`, `basic nodes` and `reference nodes`.

8.1.2.1 Start nodes

Start nodes describe the starting point of a flow graph. A flow graph shall only have one start node. A start node is shown in figure 11(a).

```
(a) Flow graph start node
```

```
(b) Flow graph end node
```

Figure 11: Start and end nodes
8.1.2.2 End nodes

End nodes describe end points of a flow graph. A flow graph may have several end nodes or in case of loops no end node. Basic nodes (see clause 8.1.2.3) and reference nodes (see clause 8.1.2.4) that have no successor nodes shall be connected to an end node to indicate that they describe the last action of a path through a flow graph. An end node is shown in figure 11(b).

8.1.2.3 Basic nodes

A basic node describes an execution unit, i.e. it is executed in one step. A basic node has a type and, depending on the type, may have an associated list of attributes. Two basic nodes are shown in figure 12.

In the inscription of a basic node the attributes of a node follow the node type and are put into round parentheses. Type and attributes are used to determine the action to be performed during execution of the represented language construct. The attributes describe information to be retrieved from the corresponding TTCN-3 construct.

Attributes have values and the operational semantics will retrieve these values by referring to the attribute name. If required, it is allowed to assign explicit values in basic nodes by using assignment ":=". An example is shown in figure 12(b).

8.1.2.4 Reference nodes

8.1.2.4.0 General

Reference nodes refer to flow graph segments (see clause 8.1.4) that are sub-flow graphs. The meaning of a reference node is defined by its replacement by the referenced flow graph segment in the flow graph. The node inscription of the reference node provides the reference to a flow graph segment. A reference node is shown in figure 13(a).

8.1.2.4.1 OR combination of reference nodes

In some cases several flow graph segments may replace a reference node. For these cases an OR operator may be used to refer to several flow graph segments (see figure 13(b)). In the actual flow graph representing the module control, a test case or a function, one alternative is determined by the represented construct.
8.1.2.4.2 Multiple occurrences of reference nodes

In some cases the same kind of reference node may occur zero, one or more times in a flow graph. In regular expressions the possible repetition of parts of a regular expression is described by using the operator symbols "+" (one or more repetitions) and "*" (zero or more repetitions). As shown in figure 14, these operators have been adopted to flow graphs by introducing double-framed reference nodes with associated operator symbols. A single flow (see clause 8.1.3) line shall replace a reference node, in case of zero occurrences (using a double-framed reference node with "*-operator).

![Figure 14: Repetition of reference nodes](image)

An upper bound of possible repetitions of a reference node can be given in form of an integer number in round parenthesis following the "*" or "+" symbol in the double framed reference node. The segment reference shown in figure 15 may occur from zero up to 5 times.

![Figure 15: Restricted repetition of a reference node](image)

8.1.3 Flow lines

Flow lines are represented by means of arrows. A flow line has an inscription of true or false which indicates a condition under which the flow line is chosen during the flow graph interpretation. As a short hand notation it is allowed to omit the true inscription. Examples of flow lines are shown in figure 15a.

![Figure 15a: Explicit and implicit inscriptions of flow lines](image)

To support the joining of several flow lines into one flow line on a graphical level, a special join node is introduced. The join node and an example for its usage are shown in figure 15b.
join node:  

usage of join node: 

Figure 15b: Joining of flow lines

Drawing long flow lines in big diagrams as it is, for example, necessary to model the TTCN-3 constructs `goto` and `label`, is awkward. For this purpose, labels for outgoing and incoming flow lines can be used. Examples are shown in figure 15c.

Incoming flow line with label: 

Outgoing flow line with label: 

Figure 15c: Incoming and outgoing flow lines with labels

An outgoing flow line with a label is connected with an incoming flow line with a label, if the labels are identical. The flow line labels for the incoming flow lines shall be unique. If there are several outgoing flow lines with the same label, this is considered to be a join of lines to the incoming flow line with an identical label.

8.1.4 Flow graph segments

Flow graph segments are sub-flow graphs. They are referenced in reference nodes and define the meaning of that reference node. Flow graph segments may include further reference nodes.

As shown in figure 16, flow graph segments have precise interfaces that consist of incoming and outgoing flow lines. There is only one unlabeled incoming and one or none unlabeled outgoing flow lines. In addition there might exist several labelled incoming and outgoing flow lines. For example, the labelled incoming and outgoing flow lines are needed to describe the meaning of TTCN-3 statements `goto` and `alt`.

Flow graph segments are put into a frame and the name of the flow graph segment shall follow the keyword `segment` followed by the segment name in the upper left corner of the frame. The flow lines describing the flow graph segment interface shall cross the flow graph segment frame.
### 8.1.5 Comments

To improve readability and coherence a special comment symbol can be used to associate comments to flow graph nodes and flow lines. The comment symbol and its usage are shown in figure 17.

![Comment symbol and usage](image)

**Figure 17: Flow graph representation of comments**

### 8.1.6 Handling of flow graph descriptions

The evaluation procedure of the operational semantics traverses flow graphs that only consist of basic nodes, i.e. all reference nodes have to be expanded by the corresponding flow graph segment definitions. The `NEXT` function is required to support this traversal. `NEXT` is defined in the following manner:

\[
\text{actualNodeRef}.\text{NEXT}(\text{bool}) := \text{successorNodeRef}
\]

where:

- `actualNodeRef` is the reference of a basic flow graph node;
- `successorNodeRef` is the reference of a successor node of the node referenced by `actualNodeRef`;
- `bool` is a Boolean specifying whether the `true` or the `false` successor is returned (see clause 8.1.3).
8.2 Flow graph representation of TTCN-3 behaviour

8.2.0 General

The operational semantics assumes that TTCN-3 behaviour descriptions are provided in form of a set of flow graphs, i.e. for each TTCN-3 behaviour description a separate flow graph has to be constructed.

The operational semantics interprets the following kinds of TTCN-3 definitions as behaviour descriptions:

- a) module control;
- b) test case definitions;
- c) function definitions;
- d) altstep definitions;
- e) component type definitions.

The module control specifies the test campaign, i.e. the execution order (possibly repetitious) of the actual test cases. Test case definitions define the behaviour of the MTC. Functions structure behaviour. They are executed by the module control or by the test components. Altsteps are used for the definition of default behaviour or in a function-like manner to structure behaviour. Component type definitions are assumed to be behaviour descriptions because they specify the creation, declaration and initialization of ports, constants, variables and timers during the creation of an instance of a component type.

8.2.1 Flow graph construction procedure

The flow graphs presented in the figures 18 to 22 and the flow graph segments presented in clause 8 are only templates. They include placeholders for information that has to be provided in order to produce a concrete flow graph or flow graph segment. The placeholders are marked with "<" and ">" parenthesis.

The construction of a flow graph representation of a TTCN-3 module is done in three steps:

1) For each TTCN-3 statement in module control, test cases, altsteps, functions and component type definitions a concrete flow graph segment is constructed.
2) For the module control and for each test case, altstep, function and component type definition a concrete flow graph (with reference nodes) is constructed.
3) In a stepwise procedure all reference nodes in the concrete flow graphs are replaced by corresponding flow graph segment definitions until all flow graphs only include one start node, end nodes and basic flow graph nodes.

NOTE 1: Basic flow graph nodes describe basic indivisible execution units. The operational semantics for TTCN-3 behaviour is based on the interpretation of basic flow graph nodes. Clause 8.6 presents execution methods for basic flow graph nodes only.

The replacement of a reference node by the corresponding flow graph segment definition may lead to unconnected parts in a flow graph, i.e. parts which cannot be reached from the start node by traversing through the flow graph along the flow lines. The operational semantics will ignore unconnected parts of a flow graph.

NOTE 2: An unconnected part of a flow graph is a result of the mechanical replacement procedure. For the construction of an optimal flow graph representation the different combinations of TTCN-3 statements also has to be taken into consideration. However, the goal of the present document is to provide a correct and complete semantics, not an optimal flow graph representation.
8.2.2 Flow graph representation of module control

Schematically, the syntactical structure of a TTCN-3 module is:

```plaintext
module <identifier> <module-definitions-part> control <statement-block>
```

For the flow graph behaviour representation the following information is relevant only:

```plaintext
module <identifier> <statement-block>
```

This is comparable to a function definition and therefore the flow graph representation of module control is similar to the flow graph representation of a function (see clause 8.2.4). The semantics will access the flow graph representing the module control by using the module name.

**NOTE:** The meaning of the module definitions part is outside the scope of this operational semantics. Module parameters are defined as global constants at run-time. References to module parameters have to be replaced by their concrete values on a syntactical level (see clause 8.3).

The scheme of the flow graph representation of the module control is shown in figure 18. The flow graph name `control` identifies the flow graph representing the module control. The nodes of the flow graph have associated comments describing the meaning of the different nodes. The reference node `<stop-entity-op>` covers the case where no explicit `stop` operation is specified, i.e. the operational semantics assumes that a `stop` operation is implicitly added.

![Flow graph representation of module control](image)

Figure 18: Flow graph representation of module control

8.2.3 Flow graph representation of test cases

Schematically, the syntactical structure of a TTCN-3 test case definition is:

```plaintext
testcase <identifier> (<parameter>) <testcase-interface> <statement-block>
```

The `<testcase-interface>` above refers to the (mandatory) `runs on` and the (optional) `system` clauses in the test case definition. The flow graph description of a test case describes the behaviour of the MTC. Variables, timers and constants defined and declared in the component type definition are made visible to the MTC behaviour by the `runs on` clause in the `<testcase-interface>`. The `system` clause is not relevant for the MTC and is therefore not represented in the flow graph representation of a test case.
The scheme of the flow graph representation of a test case is shown in figure 19. The flow graph name \(<\text{identifier}>\) refers to the name of the represented test case. The nodes of the flow graph have associated comments describing the meaning of the different nodes. The reference node \(<\text{stop-entity-op}>\) covers the case where no explicit \textit{stop} operation for the MTC is specified, i.e. the operational semantics assumes that a \textit{stop} operation is implicitly added.

![Flow graph representation of test cases](image)

**Figure 19: Flow graph representation of test cases**

### 8.2.4 Flow graph representation of functions

Schematically, the syntactical structure of a TTCN-3 function is:

```
function <identifier> (<parameter>) [<function-interface>] <statement-block>
```

The optional \(<\text{function-interface}>\) above refers to the (optional) \textit{runs on} and the (optional) \textit{return} clauses in the function definition.

The scheme of the flow graph representation of a function is shown in figure 20. The flow graph name \(<\text{identifier}>\) refers to the name of the represented function. Variables, timers, constants and ports defined and declared in the component type definition are made visible to within the function by the \textit{runs on} clause in the \(<\text{function-interface}>\). A missing \textit{runs on} clause means that definitions within the component type definition are not known within the scope of the function. The operational semantics distinguishes these two cases by the reference nodes \(<\text{init-scope-with-runs-on}>\) and \(<\text{init-scope-without-runs-on}>\). The reference node \(<\text{return-without-value}>\) covers the case where no explicit \textit{return} statement is specified, i.e. the operational semantics assumes that a \textit{return} statement is implicitly added.
**8.2.5 Flow graph representation of altsteps**

Schematically, the syntactical structure of a TTCN-3 altstep is:

```plaintext
altstep <identifier> (<parameter>) [ [altstep-interface] ]
  <constant-variable-timer-declarations>
  ( <receiving-branch> | <else-branch> ) *
```

**NOTE:** Only the alternatives up to the first else branch and the first else branch are taken into consideration. Branches following the first else branch are unreachable.

The optional `<altstep-interface>` above refers to the `runs on` clause in the altstep definition.

The scheme of the flow graph representation of an altstep is shown in figure 21. The flow graph name `<identifier>` refers to the name of the represented altstep. Variables, timers, constants and ports defined and declared in the component type definition are made visible to within the function by the `runs on` clause in the `<function-interface>`. A missing `runs on` clause means that definitions made within the component type definition are not known within the scope of the function. The operational semantics distinguishes these two cases by the reference nodes `<init-scope-with-runs-on>` and `<init-scope-without-runs-on>`. The reference node `<return-without-value>` covers the case where no else-branch is specified and none of the alternatives can be selected.
8.2.6 Flow graph representation of component type definitions

Schematically, the syntactical structure of a TTCN-3 component type definition is:

\[\text{type component}\ <\text{identifier}> <\text{port-constant-variable-timer-declarations}>\]

The semantics will access flow graphs representing types by using the component type names.

The scheme of the flow graph representation of a component type definition is shown in figure 22. The flow graph name \(<\text{identifier}>\) refers to the name of the represented component type.
8.2.7 Retrieval of start nodes of flow graphs

For the retrieval of the start node reference of a flow graph the following function is required:

The GET-FLOW-GRAPH function: GET-FLOW-GRAPH (flow-graph-identifier)

The function returns a reference to the start node of a flow graph with the name flow-graph-identifier. The flow-graph-identifier refers to the module name for the control, to test case names, to function names, to altstep names and to component type names.

8.3 State definitions for TTCN-3 modules

8.3.0 General

During the interpretation of flow graphs representing TTCN-3 behaviour, module states are manipulated. A module state is a structured state that consists of several sub states describing the states of module control and the different test configurations. A test configuration state describes the states of test components and ports. Module states, configuration states, component states and port states are introduced in this clause. In addition, functions to retrieve information from and to manipulate states are defined.
8.3.1 Module state

8.3.1.0 General

As shown in figure 23 a module state is structured into a module CONTROL state and a TEST-CONFIGURATION state. The module CONTROL state describes the state of the module control. Module control is handled like a test component, i.e. CONTROL is an entity state as defined in clause 8.3.2. The TEST-CONFIGURATION state represents the test configurations that is instantiated when a test case is executed by module control.

![Figure 23: Structure of a module state](image)

8.3.1.1 Accessing the module state

The CONTROL state and the TEST-CONFIGURATION state of the module state can be addressed by using their names, i.e. CONTROL and TEST-CONFIGURATION.

8.3.1a Configuration state

8.3.1a.0 General

As shown in figure 23a the configuration state is structured into ALL-ENTITY-STATES, ALL-PORT-STATES, TC-VERDICT, DONE and KILLED. ALL-ENTITY-STATES describes the states of all instantiated test components during the execution of a test case. The first element of ALL-ENTITY-STATES is the reference to the MTC of the configuration. ALL-PORT-STATES describes the states of the different ports. TC-VERDICT stores the actual global test verdict of a test case. DONE and KILLED are lists of component-verdict-states (CVSs). A component verdict state denotes a stopped or killed component together with its local verdict at the point in time when the component was stopped or killed.

NOTE 1: The number of updates of TC-VERDICT is identical to the number of test components that have terminated.

NOTE 2: An alive-type test component is put into the DONE list each time when it is stopped and removed from the DONE list each time when it is started. It is put into the KILL and the DONE list when it is killed.

NOTE 3: Port states may be considered to be part of the entity states. By connect and map ports are made visible for other components and therefore, this operational semantics handles ports on the top level of the configuration state.

![Figure 23a: Structure of a configuration state](image)

8.3.1a.1 Accessing the configuration state

The TC-VERDICT and the lists ALL-ENTITY-STATES, ALL-PORT-STATES, DONE and KILLED can be accessed like variables by their name.

For the handling of lists, e.g. ALL-ENTITY-STATES, ALL-PORT-STATES, DONE and KILLED in module states, the list operations add, append, delete, member, first, last, length, next, random and change can be used. They have the following meaning:

- myList.add(item) adds item as first element into the list myList and myList.add(sublist) adds the list sublist to list myList, i.e. add can be used to add single elements or lists to lists;

- myList.append(item) appends item as last element into the list myList and myList.append(sublist) appends the list sublist to list myList, i.e. append can be used to append single elements or lists to lists;
• `myList.delete(item)` deletes `item` from the list `myList`;
• `myList.member(item)` returns `true` if `item` is an element of the list `myList`, otherwise `false`;
• `myList.first()` returns the first element of `myList`;
• `myList.last()` returns the last element of `myList`;
• `myList.length()` returns the length of `myList`;
• `myList.next(item)` returns the element that follows `item` in `myList`, or `NULL` if `item` is the last element in `myList`;
• `myList.random(<condition>)` returns randomly an element of `myList`, which fulfils the Boolean condition `<condition>` or `NULL`, if no element of `myList` fulfils `<condition>`;
• `myList.change(<operation>)` allows to apply `<operation>` on all elements of `myList`.

NOTE 1: The operations `random` and `change` are not common list operations. They are introduced to explain the meaning of the keywords `all` and `any` in TTCN-3 operations.

NOTE 2: Arguments of the operations `delete`, `member` and `next` may include "-" symbols denoting a field not relevant for the unique identification of an item. For example, for a list `aList` of 2-tuples containing the tuple `(A, B)`, `aList.member(A, -)` returns `true` if the field A uniquely identifies `(A, B)`, otherwise `false`.

Additionally, a general `copy` operation is available. The `copy` operation copies and returns an item instead of returning a reference to an item:
• `copy(item)` returns a copy of `item`.

8.3.2 Entity states

8.3.2.0 General

Entity states are used to describe the actual states of module control and test components. In the module state, `CONTROL` is an entity state and in the configuration state, the test component states are handled in the list `ALL-ENTITY-STATES`. The structure of an entity state is shown in figure 24.

```
STATUS
CONTROL-STACK
DEFAULT-LIST
DEFAULT-POINTER
VALUE-STACK
E-VERDICT
TIMER-GUARD
DATA-STATE
TIMER-STATE
PORT-REF
SNAP-ALIVE
SNAP-DONE
SNAP-KILLED
KEEP-ALIVE
```

Figure 24: Structure of an entity state

The `STATUS` describes whether the module control or a test component is `ACTIVE`, `BREAK`, `SNAPSHOT`, `REPEAT` or `BLOCKED`. Module control is blocked during the execution of a test case. Test components are blocked during the creation of other test components, i.e. when they call a `create` operation, and when they wait for being started. The status `SNAPSHOT` indicates that the component is active, but in the evaluation phase of a snapshot. The status `REPEAT` denotes that the component is active and in an `alt` statement that should be re-evaluated due to a `repeat` statement. The `BREAK` status is set when a `break` statement is executed for leaving altstep. In this case, the `alt` statement in which the altstep was directly or indirectly (i.e. by means of the default mechanism) called is immediately left.
The **CONTROL-STACK** is a stack of flow graph node references. The top element in **CONTROL-STACK** is the flow graph node that has to be interpreted next. The stack is required to model function calls in an adequate manner.

The **DEFAULT-LIST** is a list of activated defaults, i.e. it is a list of pointers that refer to the start nodes of activated defaults. The list is in the reverse order of activation, i.e. the default that has been activated first is the last element in the list.

During the execution of the default mechanism, the **DEFAULT-POINTER** refers to the next default that has to be evaluated if the actual default terminates unsuccessfully.

The **VALUE-STACK** is a stack of values of all possible types that allows an intermediate storage of final or intermediate results of operations, functions and statements. For example, the result of the evaluation of an expression or the result of the mtc operation will be pushed onto the **VALUE-STACK**. In addition to the values of all data types known in a module, the special value **MARK** has been defined. **MARK** is element of the stack alphabet. When leaving a scope unit, the **MARK** is used to clean **VALUE-STACK**.

The **E-VERDICT** stores the actual local verdict of a test component. The **E-VERDICT** is ignored if an entity state represents the module control.

The **TIMER-GUARD** represents the special timer, which is necessary to guard the execution time of test cases and the duration of call operations. The **TIMER-GUARD** is modelled as a timer binding (see clause 8.3.2.4 and figure 28).

The **DATA-STATE** is considered to be a list of lists of variable bindings. The list of lists structure reflects nested scope units due to nested function and altstep calls. Each list in the list of lists of variable bindings describes the variables declared in a certain scope unit and their values. Entering or leaving a scope unit corresponds to adding or deleting a list of variable bindings from the **DATA-STATE**. A description of the **DATA-STATE** part of an entity state can be found in clause 8.3.2.2.

The **TIMER-STATE** is considered to be a list of lists of timer bindings. The list of lists structure reflects nested scope units due to nested function and altstep calls. Each list in the list of lists of timer bindings describes the known timers and their status in a certain scope unit. Entering or leaving a scope unit corresponds to adding or deleting a list of timer bindings from the **TIMER-STATE**. A description of the **TIMER-STATE** part of an entity state can be found in clause 8.3.2.4.

The **PORT-REF** is considered to be a list of lists of port bindings. The list of lists structure reflects nested scope units due to nested function and altstep calls. Nested scope units for ports are the result of port parameters in functions and altsteps. Each list in the list of lists of port bindings identifies the known ports in a certain scope unit. Entering or leaving a scope unit corresponds to adding or deleting a list of port bindings from the **PORT-REF**. A description of the **PORT-REF** part of an entity state can be found in clause 8.3.2.6.

**NOTE:** The TTCN-3 semantics administrates ports globally in the module state. Due to port parameterization, a test component may access a port by using different names in different scopes. The **PORT-REF** part of an entity state is used to identify port states uniquely in the module state.

The **SNAP-ALIVE** supports the snapshot semantics of test components. When a snapshot is taken, a copy of the **ALL-ENTITY-STATES** list of the module state will be assigned to **SNAP-ALIVE**, i.e. **SNAP-ALIVE** includes all entities (test components and test control) which are alive in the test system.

The **SNAP-DONE** supports the snapshot semantics of test components. When a snapshot is taken, a copy of the **DONE** list of the module state will be assigned to **SNAP-DONE**, i.e. **SNAP-DONE** is a list of component identifiers of stopped components.

The **SNAP-KILLED** supports the snapshot semantics of test components. When a snapshot is taken, a copy of the **KILLED** list of the module state will be assigned to **SNAP-KILL**, i.e. **SNAP-DONE** is a list of component identifiers of terminated components.

The **KEEP-ALIVE** field indicates whether the entity can be restarted after its termination or not. It is set to true if the entity can be restarted. Otherwise it is set to false.
8.3.2.1 Accessing entity states

The STATUS, DEFAULT-POINTER, E-VERDICT and TIMER-GUARD parts of an entity state are handled like variables that are globally visible, i.e. the values of STATUS, DEFAULT-POINTER and E-VERDICT can be retrieved or changed by using the "dot" notation, e.g. myEntity.STATUS, myEntity.DEFAULT-POINTER and myEntity.E-VERDICT, where myEntity refers to an entity state.

NOTE: In the following, it is assumed that the "dot" notation (with references and unique identifiers) is applicable. For example, in myEntity.STATUS, myEntityState may be pointer to an entity state or be the value of the <identifier> field.

The CONTROL-STACK, DEFAULT-LIST and VALUE-STACK of an entity state myEntity can be addressed by using the "dot" notation myEntity.CONTROL-STACK, myEntity.DEFAULT-LIST and myEntity.VALUE-STACK.

CONTROL-STACK and VALUE-STACK can be accessed and manipulated by using the stack operations push, pop, top, clear and clear-until. The stack operations have the following meaning:

- myStack.push(item) pushes item onto myStack;
- myStack.pop() pops the top item from myStack;
- myStack.top() returns the top element of myStack or NULL if myStack is empty;
- myStack.clear() clears myStack, i.e. pops all items from myStack;
- myStack.clear-until(item) pops items from myStack until item is found or myStack is empty.

DEFAULT-LIST can be accessed and manipulated by using the list operations add, append, delete, member, first, length, next, random and change. The meaning of these list operations is defined in clause 8.3.1a.1.

For the creation of a new entity state the function NEW-ENTITY is assumed to be available:

- NEW-ENTITY (flow-graph-node-reference, keep-alive);

creates a new entity state and returns its reference. The components of the new entity state have the following values:

- STATUS is set to ACTIVE;
- flow-graph-node-reference is the only (top) element in CONTROL-STACK;
- DEFAULT-LIST is an empty list;
- DEFAULT-POINTER has the value NULL;
- VALUE-STACK is an empty stack;
- E-VERDICT is set to none;
- TIMER-GUARD is a new timer binding (see clause 8.3.2.4) with name GUARD, status IDLE and no default duration;
- DATA-STATE is an empty list;
- TIMER-STATE is an empty list;
- PORT-REF is an empty list;
- SNAP-ALIVE is an empty list;
- SNAP-DONE is an empty list;
- SNAP-KILLED is an empty list;
- KEEP-ALIVE is set to the value of the keep-alive parameter.
During the traversal of a flow graph the \textit{CONTROL-STACK} changes its value often in the same manner: the top element is popped from and the successor node of the popped node is pushed onto \textit{CONTROL-STACK}. This series of stack operations is encapsulated in the \textit{NEXT-CONTROL} function:

\begin{verbatim}
myEntity.NEXT-CONTROL(myBool) {
    successorNode := myEntity.CONTROL-STACK.NEXT(myBool).top();
    myEntity.CONTROL-STACK.pop();
    myEntity.CONTROL-STACK.push(successorNode);
}
\end{verbatim}

\subsection{Data state and variable binding}

As shown in figure 25, the data state \textit{DATA-STATE} of an entity state is a list of lists of variable bindings. Each list of variable bindings defines the variable bindings in a certain scope unit. Adding a new list of variable bindings corresponds to entering a new scope unit, e.g. a function is called. Deleting a list of variable bindings corresponds to leaving a scope unit, e.g. a function executes a \textbf{return} statement.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{structure_data_state.png}
\caption{Structure of the \textit{DATA-STATE} part of an entity state}
\end{figure}

The structure of a variable binding is shown in figure 26. A variable has a name, a \textit{<location>} and a \textit{VALUE}. \textit{VAR NAME} identifies a variable in a scope unit. The \textit{<location>} is a unique identifier of the storage location of the value of the variable. The \textit{VALUE} part of a variable binding describes the actual value of a variable.

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textit{VAR NAME} & \textit{<location>} & \textit{VALUE} \\
\hline
\end{tabular}
\caption{Structure of a variable binding}
\end{figure}

NOTE: Unique location identifiers should be provided automatically when a variable is declared.

The distinction between variable name and location has been made to model function calls and the execution of test cases with value and reference parameterization in an appropriate manner:

\begin{enumerate}
\item A parameter passed in by value is handled like the declaration of a new variable, i.e. a new variable binding is appended to the list of variable bindings of the scope of the called function or executed test case. The new variable binding uses the formal parameter name as \textit{VAR NAME}, receives a new location and gets the value that is passed into the function or test case.
\item A parameter passed in by reference also leads to a new variable binding in the scope of the called function or executed test case. The new variable binding also uses the formal parameter name as \textit{VAR NAME}, but receives no new location and no new value. The new variable binding gets a copy of \textit{<location>} and \textit{VALUE} of the variable that is passed in by reference.
\end{enumerate}

When updating a variable value, e.g. in case of an assignment to a variable, the variable name is used to identify a location and all variable bindings with the same location are updated at the same time. Thus, when leaving the scope unit, the list of variables belonging to this scope unit can be deleted without further update. Due to the update procedure, variables passed in by reference automatically have the correct value.
8.3.2.3 Accessing data states

The value of a variable can be retrieved by using the "dot" notation `myEntity.myVar.VALUE`, where `myEntity` refers to an entity state and `myVar` is the name of a variable.

For the handling of variables and variable scope the following functions are considered to be defined:

a) The `VAR-SET` function: `myEntity.VAR-SET (myVar, myValue)`
   sets the `VALUE` part of variable `myVar` in the actual scope of an entity `myEntity` to `myVal`. In addition, the `VALUE` part of all variables with the same location as variable `myVar` will also be set to `myVal`.

b) The `INIT-VAR` function: `myEntity.INIT-VAR (myVar, myVal)`
   creates a new variable binding for a variable `myVar` with the initial value `myVal` in the actual scope unit of an entity `myEntity`. Using the keyword `NONE` as `myVal` means that a variable with undefined initial value is created. A new and unique `<location>` value is automatically created.

c) The `GET-VAR-LOC` function: `myEntity.GET-VAR-LOC (myVar)`
   retrieves the location of variable `myVar` owned by `myEntity`.

d) The `INIT-VAR-LOC` function: `myEntity.INIT-VAR-LOC (myVar, myLoc)`
   creates a new variable binding for a variable `myVar` with the location `myLoc` in the actual scope unit of `myEntity`. The variable will be initialized with the value of another variable with the location `myLoc`.

   NOTE: Variables with the same location are a result of parameterization by reference. Due to the handling of reference parameters as described in clause 8.3.2.2 all variables with the same location will have identical values during their lifetime.

e) The `INIT-VAR-SCOPE` function: `myEntity.INIT-VAR-SCOPE ()`
   initializes a new variable scope in the data state of entity `myEntity`, i.e. an empty list is added as first list in the list of lists of variable bindings.

f) The `DEL-VAR-SCOPE` function: `myEntity.DEL-VAR-SCOPE ()`
   deletes a variable scope of the data state of `myEntity`, i.e. the first list in the list of lists of variable bindings is deleted.

8.3.2.4 Timer state and timer binding

As shown in figures 27 and 25 the timer state `TIMER-STATE` and the data state `DATA-STATE` of an entity state are comparable. Both are a list of lists of bindings and each list of bindings defines the valid bindings in a certain scope. Adding a new list corresponds to entering a new scope unit and deleting a list of bindings corresponds to leaving a scope unit.

![Figure 27: Structure of the TIMER-STATE part of an entity state](image)
The structure of a timer binding is shown in figure 28. The meaning of \textit{TIMER-NAME} and \textit{<location>} is similar to the meaning of \textit{VAR-NAME} and \textit{<location>} for a variable binding (figure 26).

\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
\textbf{TIMER-NAME} & \textbf{<location>} & \textbf{STATUS} & \textbf{DEF-DURATION} & \textbf{ACT-DURATION} & \textbf{TIME-LEFT} & \textbf{SNAP-VALUE} & \textbf{SNAP-STATUS} \\
\hline
\end{tabular}

\textbf{Figure 28: Structure of a timer binding}

\textit{STATUS} denotes whether a timer is active, inactive or has timed out. The corresponding \textit{STATUS} values are \textit{IDLE}, \textit{RUNNING} and \textit{TIMEOUT}. \textit{DEF-DURATION} describes the default duration of a timer. \textit{ACT-DURATION} stores the actual duration with which a running timer has been started. \textit{TIME-LEFT} describes the actual duration that a running timer has to run before it times out.

\textbf{NOTE:} \textit{DEF-DURATION} is undefined if a timer is declared without default duration. \textit{ACT-DURATION} and \textit{TIME-LEFT} are set to 0.0 if a timer is stopped or times out. If a timer is started without duration, the value of \textit{DEF-DURATION} is copied into \textit{ACT-DURATION}. A dynamic error occurs if a timer is started without a defined duration.

\textit{SNAP-VALUE} and \textit{SNAP-STATUS} are needed to support the snapshot semantics of TTCN-3. When taking a snapshot, \textit{SNAP-VALUE} gets the actual value of \textit{ACT-DURATION} – \textit{TIME-LEFT}. And \textit{SNAP-STATUS} gets the same value as \textit{STATUS}. The evaluation of a snapshot will only be based on the values in \textit{SNAP-VALUE} and \textit{SNAP-STATUS}.

Timer can be only passed by reference into functions, i.e. the mechanism is similar to the mechanism for variables described in clause 8.3.2.2. This means a new timer binding (with the formal parameter name) is created which gets copies of \textit{<location>}, \textit{STATUS}, \textit{DEF-DURATION}, \textit{ACT-DURATION}, \textit{TIME-LEFT}, \textit{SNAP-VALUE} and \textit{SNAP-STATUS} from the timer that is passed in by reference. When updating a timer all timer bindings with the same \textit{<location>} value are updated at the same time.

\subsection{Accessing timer states}

The values of \textit{STATUS}, \textit{DEF-DURATION}, \textit{ACT-DURATION}, \textit{TIME-LEFT}, \textit{SNAP-VALUE} and \textit{SNAP-STATUS} of a timer \textit{myTimer} can be retrieved by using the dot notation:

- \texttt{myEntity.myTimer.STATUS;}
- \texttt{myEntity.myTimer.DEF-DURATION;}
- \texttt{myEntity.myTimer.ACT-DURATION;}
- \texttt{myEntity.myTimer.TIME-LEFT;}
- \texttt{myEntity.myTimer.SNAP-VALUE;}
- \texttt{myEntity.myTimer.SNAP-STATUS;}

The \textit{myEntity} in the dot notation refers to an entity state representing the state of a test component or module control that owns the timer \textit{myTimer}.

For changing the values of \textit{STATUS}, \textit{DEF-DURATION}, \textit{ACT-DURATION}, \textit{TIME-LEFT}, \textit{SNAP-VALUE} and \textit{SNAP-STATUS} of a timer \textit{timer-name}, the generic \texttt{TIMER-SET} operation has to be used, for example:

- \texttt{myEntity.TIMER-SET(myTimer, STATUS, myVal)}

sets the \textit{STATUS} value of timer \textit{myTimer} in the actual scope of \textit{myEntity} to the value \textit{myVal}. In addition, the \textit{STATUS} of all timers with the same location as timer \textit{myTimer} will also be set to \textit{myVal}. The \texttt{TIMER-SET} function can also be used to change the values of \textit{DEF-DURATION}, \textit{ACT-DURATION}, \textit{TIME-LEFT}, \textit{SNAP-VALUE} and \textit{SNAP-STATUS}.

For the handling of timers, timer scope and snapshot the following functions have to be defined:

\begin{enumerate}
\item[a)] The \texttt{INIT-TIMER} function: \texttt{myEntity.INIT-TIMER (myTimer, myDuration)}
\end{enumerate}

creates a new timer binding for a timer \textit{myTimer} with the default duration \textit{myDuration} in the actual scope of an entity \textit{myEntity}. Using the keyword \texttt{NONE} as \textit{myDuration} means that a timer without default duration is created.
b) The **GET-TIMER-LOC** function: \( \text{myEntity.GET-TIMER-LOC}(\text{myTimer}) \)

retrieves the location of timer \( \text{myTimer} \) owned by \( \text{myEntity} \).

c) The **INIT-TIMER-LOC** function: \( \text{myEntity.INIT-TIMER-LOC}(\text{myTimer, myLocation}) \)

creates a new timer binding for a timer \( \text{myTimer} \) with the location \( \text{myLocation} \) in the actual scope unit of \( \text{myEntity} \). The timer will be initialized with the values of \( \text{STATUS}, \text{DEF-DURATION}, \text{ACT-DURATION} \) and \( \text{TIME-LEFT} \) of another timer with the location \( <\text{location}> \).

**NOTE:** Timers with the same location are a result of parameterization by reference. Due to the handling of timer reference parameters as described in clause 8.3.2.3 all timers with the same location will have identical values for \( \text{STATUS, DEF-DURATION, ACT-DURATION} \) and \( \text{TIME-LEFT} \) during their lifetime.

d) The **INIT-TIMER-SCOPE** function: \( \text{myEntity.INIT-TIMER-SCOPE}() \)

initializes a new timer scope in the timer state of entity \( \text{myEntity} \), i.e. an empty list is added as first list in the list of lists of timer bindings.

e) The **DEL-TIMER-SCOPE** function: \( \text{myEntity.DEL-TIMER-SCOPE}() \)

deletes a timer scope of the timer state of entity \( \text{myEntity} \), i.e. the first list in the list of lists of timer bindings is deleted.

f) The **SNAP-TIMER** function: \( \text{myEntity.SNAP-TIMER}() \)

makes an update of \( \text{SNAP-VALUE} \) and \( \text{SNAP-STATUS} \), in all timers owned by \( \text{myEntity} \), i.e.:

\[
\text{myEntity.SNAP-TIMERS}() \{ \\
\text{for all myTimer in TIMER-STATE} \{ \\
\text{myEntity.myTimer.SNAP-VALUE := myEntity.myTimer.ACT-DURATION - myEntity.myTimer.TIME-LEFT; } \\
\text{myEntity.myTimer.SNAP-STATUS := myEntity.myTimer.STATUS;} \\
\}\}
\]

### 8.3.2.6 Port references and port binding

As shown in figures 28a, 27 and 25 the port references \( \text{PORT-REF} \), the timer state \( \text{TIMER-STATE} \) and the data state \( \text{DATA-STATE} \) of an entity state are comparable. All three are a list of lists of bindings and each list of bindings defines the valid bindings in a certain scope. Adding a new list corresponds to entering a new scope unit and deleting a list of bindings corresponds to leaving a scope unit.

![Figure 28a: Structure of the PORT-REF part of an entity state](image)

The structure of a port binding is shown in figure 28b. A port has two names. \( \text{PORT-NAME} \) identifies a port in a scope unit. \( \text{COMP-PORT-NAME} \) is the port name given in the component type to a port.

![Figure 28b: Structure of a port binding](image)
NOTE: PORT-NAME and COMP-PORT-NAME are equal directly after the creation of a component.

Ports can be only passed by reference into functions and alstps, i.e. the mechanism is similar to the mechanism for variables described in clause 8.3.2.2. This means a new port binding (with the formal parameter name) is created which gets a copy of COMP-PORT-NAME from the port that is passed in by reference. When accessing a port which is passed in by reference, the corresponding port binding is used to retrieve the port name declared in the component type definition.

8.3.2.7 Accessing port references

The value of COMP-PORT-NAME can be retrieved by using the dot notation:

- myEntity.myport.COMP-PORT-NAME

The myEntity in the dot notation refers to an entity state representing the state of a test component that owns the port myPort.

For the handling of port parameters and port scopes the following functions have to be defined:

a) The INIT-PORT function: myEntity.INIT-PORT (myPort, myCompPortName)

creates a new port binding for a port myPort with myCompPortName as value for COMP-PORT-NAME in the actual scope of an entity myEntity.

b) The INIT-PORT-SCOPE function: myEntity.INIT-PORT-SCOPE ()

initializes a new port scope in the port references of entity myEntity, i.e. an empty list is added as first list in the list of lists of port bindings.

c) The DEL-PORT-SCOPE function: myEntity.DEL-PORT-SCOPE ()

deletes a port scope of the port references of entity myEntity, i.e. the first list in the list of lists of port bindings is deleted.

8.3.3 Port states

8.3.3.0 General

Port states are used to describe the actual states of ports. Within a module state, the port states are handled in the ALL-PORT-STATES list (see figure 23). The structure of a port state is shown in figure 29. The COMP-PORT-NAME refers to the port name that is used to declare the port in the component type definition of the test component OWNER that owns the port. STATUS provides the actual status of the port. A port may either be STARTED, HALTED or STOPPED.

NOTE: A port in a test system is uniquely identified by the owning test component and by the port name used in the component type definition to declare the port.

The CONNECTIONS-LIST of a port state keeps track of the connections between the different ports in the test system. The mechanism is explained in clause 8.3.3.1.

The VALUE-QUEUE in a port state stores the messages, calls, replies and exceptions that are received at this port but have not yet been consumed.

The SNAP-VALUE supports the TTCN-3 snapshot mechanism. When a snapshot is taken, the first element in VALUE-QUEUE is copied into SNAP-VALUE. SNAP-VALUE will get the value NULL if VALUE-QUEUE is empty or STATUS is STOPPED.

<table>
<thead>
<tr>
<th>COMP-PORT-NAME</th>
<th>OWNER</th>
<th>STATUS</th>
<th>CONNECTIONS-LIST</th>
<th>VALUE-QUEUE</th>
<th>SNAP-VALUE</th>
</tr>
</thead>
</table>

Figure 29: Structure of a port state
8.3.3.1 Handling of connections among ports

A connection between two test components is made by connecting two of their ports by means of a `connect` operation. Thus, a component can afterwards use its local port name to address the remote queue. As shown in figure 30, connection is represented in the states of both connected queues by a pair of `REMOTE-ENTITY` and `REMOTE-PORT-NAME`. The `REMOTE-ENTITY` is the unique identifier of the test component that owns the remote port. The `REMOTE-PORT-NAME` refers to the port name that is used to declare the port in the component type definition of the test component `REMOTE-ENTITY`. TTCN-3 supports one-to-many connections of ports and therefore all connections of a port are organized in a list.

NOTE 1: Connections made by `map` operations are also handled in the list of connections. The `map` operation: `map(PTC1:MyPort, system, PCO1)` leads to a new connection `(system, PCO1)` in the port state of MyPort owned by PTC1. The remote side to which PCO1 is connected to, resides inside the SUT. Its behaviour is outside the scope of this semantics.

NOTE 2: The operational semantics handles the keyword `system` as a symbolic address. A connection `(system, myPort)` in the list of connections of a port it indicates that the port is mapped onto the port myPort in the test system interface.

![Figure 30: Structure of a connection](image)

8.3.3.2 Handling of port states

The queue of values in a port state can be accessed and manipulated by using the known queue operations `enqueue`, `dequeue`, `first` and `clear`. Using a `GET-PORT` or a `GET-REMOTE-PORT` function references the queue that shall be accessed.

NOTE 1: The queue operations `enqueue`, `dequeue`, `first` and `clear` have the following meaning:

- `myQueue.enqueue(item)` puts item as last item into `myQueue`;
- `myQueue.dequeue()` deletes the first item from `myQueue`;
- `myQueue.first()` returns the first item in `myQueue` or `NULL` if `myQueue` is empty;
- `myQueue.clear()` removes all elements from `myQueue`.

The handling of port states is supported by the following functions:

a) The `NEW-PORT` function: `NEW-PORT (myEntity, myPort)`

   creates a new port and returns its reference. The `OWNER` entry of the new port is set to `myEntity` and `COMP-PORT-NAME` has the value `myPort`. The status of the new port is `STARTED`. The `CONNECTIONS-LIST` and the `VALUE-QUEUE` are empty. The `SNAP-VALUE` has the value `NULL` (i.e. the input queue of the new port is empty).

b) The `GET-PORT` function: `GET-PORT (myEntity, myPort)`

   returns a reference to the port identified by `OWNER myEntity` and `COMP-PORT-NAME myPort`.

c) The `GET-REMOTE-PORT` function: `GET-REMOTE-PORT (myEntity, myPort, myRemoteEntity)`

   returns the reference to the port that is owned by test component `myRemoteEntity` and connected to a port identified by `OWNER myEntity` and `COMP-PORT-NAME myPort`. The symbolic address `SYSTEM` is returned, if the remote port is mapped onto a port in the test system interface.

NOTE 2: `GET-REMOTE-PORT` returns `NULL` if there is no remote port or if the remote port cannot be identified uniquely. The special value `NONE` can be used as value for the `myRemoteEntity` parameter if the remote entity is not known or not required, i.e. there exists only a one-to-one connection for this port.
d) The STATUS of a port is handled like a variable. It can be addressed by qualifying STATUS with a GET-PORT call:

\[
\text{GET-PORT}(\text{myEntity, myPort}) \text{ STATUS}
\]

e) The ADD-CON function: \( \text{ADD-CON} (\text{myEntity, myPort, myRemoteEntity, myRemotePort}) \)

adds a connection (myRemoteEntity, myRemotePort) to the list of connections of the port identified by OWNER myEntity and COMP-PORT-NAME myPort.

f) The DEL-CON function: \( \text{DEL-CON} (\text{myEntity, myPort, myRemoteEntity, myRemotePort}) \)

removes a connection (myRemoteEntity, myRemotePort) from the list of connections of the port identified by OWNER myEntity and COMP-PORT-NAME myPort.

g) The SNAP-PORTS function: \( \text{SNAP-PORTS} (\text{myEntity}) \)

updates SNAP-VALUE for all ports owned by myEntity, i.e.

\[
\text{SNAP-PORTS} (\text{myEntity}) \{
\text{for all ports p /* in the module state */} \{
\text{if} (\text{p.OWNER == myEntity}) \{
\text{if} (\text{p.STATUS == STOPPED}) \{
\text{p.SNAP-VALUE := NULL;}
\}
\text{else} \{
\text{if} (\text{p.STATUS == HALTED && p.first() == HALT-MARKER}) \{
\text{// Port is halted and halt marker is reached}
\text{p.SNAP-VALUE := NULL;}
\text{p.dequeue();} \text{// Removal of halt marker}
\text{p.STATUS := STOPPED;}
\}
\text{else} \{
\text{p.SNAP-VALUE := p.first();}
\}
\}
\}
\}
}\]

NOTE 3: The SNAP-PORTS function handles the HALT-MARKER that may be put by a halt port operation into the port queue. If such a marker is found, the marker is removed, the SNAP-VALUE of the port is set to NULL and the status of the port is changed to STOPPED.

8.3.3a Component verdict states

Component verdict states are the elements of the DONE and KILLED lists in a configuration state. As shown in figure 30a, a component verdict state consists of a component identifier (COMP-ID) and a verdict (FIN-VERDICT). In DONE and KILLED, a component verdict state denotes a stopped or killed component together with its local verdict when it was stopped or killed.

\[
\begin{array}{|c|c|}
\hline
\text{COMP-ID} & \text{FIN-VERDICT} \\
\hline
\end{array}
\]

Figure 30a: Component Verdict State

8.3.4 General functions for the handling of module states

The operational semantics assumes the existence of the following functions for the handling of module states.

NOTE 1: During the interpretation of a TTCN-3 module, there only exists one module state. It is assumed that the components of the module state are stored in global variables and not in a complex data object. Thus, the following functions are assumed to work on global variables and do not address a specific module state object.
a) The **DEL-ENTITY** function:  
`DEL-ENTITY(myEntity)`  
deletes an entity with the unique identifier `myEntity`. The deletion comprises:
- the deletion of the entity state of `myEntity`;
- deletion of all ports owned by `myEntity`;
- deletion of all connections in which `myEntity` is involved.

b) The **UPDATE-REMOTE-REFERENCES** function:

`UPDATE-REMOTE-REFERENCES (source, target)`

the **UPDATE-REMOTE-REFERENCES** updates variables and timers with the same location in both entities. The values that will be used for the update are the values of variables and timers owned by `source`.

**NOTE 2**: The **UPDATE-REMOTE-REFERENCES** is used during the termination of test cases. It allows updating of variables of module control, which are passed as reference parameters to test cases.

## 8.4 Messages, procedure calls, replies and exceptions

### 8.4.0 General

The exchange of information among test components and between test components and the SUT is related to **messages, procedure calls, replies to procedure calls and exceptions**. For communication purposes these items have to be constructed, encoded and decoded. The concrete encoding, i.e. mapping of TTCN-3 data types to bits and bytes, and decoding, i.e. mapping of bits and bytes to TTCN-3 data types, is outside the scope of the operational semantics. In the present document **messages, procedure calls, replies to procedure calls and exceptions** are handled on a conceptual level.

### 8.4.1 Messages

Messages are related to message-based communication. Values of all (pre- and user-defined) data types can be exchanged among the entities that communicate. As shown in figure 31, the operational semantics handles a message as structured object that consist of a `sender` a `type` and a `value` part. The `sender` part identifies the sender entity of a message, the `type` part specifies the type of a message and the `value` part defines the message value.

![Figure 31: Structure of a message](image)

**NOTE**: The operational semantics only presents a model for the concepts of TTCN-3. Whether and how the `sender` information is or has to be sent and/or received depends on the implementation of the test system, e.g. in some cases the sender information may be part of the value part of a message and therefore is no separate part of the message structure.

### 8.4.2 Procedure calls and replies

Procedure calls and replies to procedures are related to procedure-based communication. They are defined like values of a record with components representing the parameters. The operational semantics also handles procedure calls and replies to procedure calls like values in structured types. The structure of a procedure call and the structure of a reply are presented in figures 32 and 33.

The `sender` and the `procedure-reference` parts have the same meaning in both figures. The `sender` part refers to the sender entity of a call or the reply to a procedure call. The `procedure-reference` refers to the procedure to which call and reply belong. The `parameter-part` of the procedure call in figure 32 refers to the `in` parameters and `inout` parameters and the `parameter-part` of the reply in figure 33 refers to the `inout` parameters and `out` parameters of the procedure to which call and reply belong. In addition, the reply has a `value` part for the return values in the reply to a procedure.
NOTE 1: As stated in the previous note, the operational semantics only presents a model for the concepts of TTCN-3. Whether and how the information described in figures 32 and 33 is or has to be sent and/or received depends on the implementation of the test system.

NOTE 2: For a procedure call, **out** parameters are of no relevance and are omitted in figure 32. For a reply to a procedure call, **in** parameters are of no relevance and are omitted in figure 33.

NOTE 3: The types of parameters and the type of the return value can always be derived unanimously from the related signature definition.

<table>
<thead>
<tr>
<th>sender</th>
<th>procedure-reference</th>
<th>parameter-part</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>in-or-inout-parameter</strong>₁</td>
</tr>
</tbody>
</table>

Figure 32: Structure of a procedure call

<table>
<thead>
<tr>
<th>sender</th>
<th>procedure-reference</th>
<th>parameter-part</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>inout-or-out-parameter</strong>₁</td>
<td><strong>…</strong></td>
</tr>
</tbody>
</table>

Figure 33: Structure of a reply to a procedure call

### 8.4.3 Exceptions

Exceptions are also related to procedure-based communication. The structure of an exception is shown in figure 34. It consists of four parts. The **sender** part identifies the sender of the exception; the **procedure-reference** part refers to the procedure to which the exception belongs, the **type** part identifies the type of the exception and the **value** part provides the value of the exception. The procedure signature referred to in the procedure reference part defines the list of allowed types of exceptions. A received exception shall comply with one of the listed types. In general it can be of any pre- or user-defined TTCN-3 data type.

<table>
<thead>
<tr>
<th>sender</th>
<th>procedure-reference</th>
<th>type</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 34: Structure of an exception

### 8.4.4 Construction of messages, procedure calls, replies and exceptions

The operations for sending a message, a procedure call, a reply to a procedure call or an exception are **send**, **call**, **reply** and **raise**. All these sending operations are built up in the same manner:

```
<port-name>.<sending-operation>({<send-specification>}) [to <receiver>]   
```

The <port-name> and <sending-operation> define port and operation used for sending an item. In case of one-to-many connections a <receiver> entity needs to be specified. The item to be sent is constructed by using the <send-specification>. The send specification may use concrete values, template references, variable values, constants, expressions, functions, etc. to construct and encode the item to be sent.

The operational semantics assumes that there exists a generic **CONSTRUCT-ITEM** function:

```
CONSTRUCT-ITEM (myEntity, <sending-operation>, <send-specification>)  
```

returns a message, a procedure call, a reply to a procedure call or an exception depending on the <sending-operation> and the <send-specification> (both, <sending-operation> and the <send-specification> refer to the corresponding parts in the TTCN-3 sending operation). The entity reference myEntity is the sender of the item to be sent. This **sender** information is also assumed to be part of the item to be sent (figures 31 to 34).

### 8.4.5 Matching of messages, procedure calls, replies and exceptions

The operations for receiving a message, a procedure call, a reply to a procedure call or an exception are **receive**, **getcall**, **getreply** and **catch**. All these receiving operations are built up in the same manner:

```
<port-name>.<receiving-operation>({<matching-part>}) [from <sender>] [assignment-part]  
```
The `<port-name>` and `<receiving-operation>` define port and operation used for the reception of an item. In case of one-to-many connections a from-clause can be used to select a specific sender entity `<sender>`. The item to be received has to fulfill the conditions specified in the `<matching-part>`, i.e. it has to match. The `<matching-part>` may use concrete values, template references, variable values, constants, expressions, functions, etc. to specify the matching conditions.

The operational semantics assumes that there exists a generic `MATCH-ITEM` function:

\[
\text{MATCH-ITEM} (\text{myItem}, \text{<matching-part>}, \text{<sender>})
\]

returns `true` if `myItem` fulfills the conditions of `<matching-part>` and if `myItem` has been sent by `<sender>`, otherwise it returns `false`.

### 8.4.6 Retrieval of information from received items

Information from received messages, procedure calls, replies to procedure calls and exceptions can be retrieved in the `<assignment-part>` (see clause 8.4.5) of the receiving functions `receive`, `getcall`, `getreply` and `catch`. The `<assignment-part>` describes how the parameters of procedure calls and replies, return values encoded in replies, messages, exceptions and the identifier of the `<sender>` entity are assigned to variables.

The operational semantics assumes that there exists a generic `RETRIEVE-INFO` function:

\[
\text{RETRIEVE-INFO} (\text{myItem}, \text{<assignment-part>})
\]

all values to be retrieved according to the `<assignment-part>` are retrieved and assigned to the variables listed in the assignment part. Assignments are done by means of the `VAR-SET` operation, i.e. variables with the same location are updated at the same time.

### 8.5 Call records for functions, altsteps and test cases

#### 8.5.0 General

Functions, altsteps and test cases are called (or executed) by their name and a list of actual parameters. The actual parameters provide references for reference parameter and concrete values for the value parameter as defined by the formal parameters in the corresponding function, altstep or test case definition. The operational semantics handles calls of functions, altsteps and test cases by using call records as shown in figure 35. The value of `BEHAVIOUR-ID` is the name of a function or test case, value parameters provide concrete values `<parId1>` … `<parIdn>` for the formal parameters `<parId1>` … `<parIdz>`. Variable and timer reference parameters provide references to locations of existing variables and timers. Port reference parameters provide the port names declared in the component type definition of the test component that calls the function or altstep. Before a function or test case can be executed an appropriate call record has to be constructed.

**NOTE:** Port reference parameters can only appear in functions and altsteps which are executed on a test component.

<table>
<thead>
<tr>
<th>behaviour-id</th>
<th>value-parameters</th>
<th>variable and timer reference-parameters</th>
<th>port reference-parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>&lt;parId1&gt;</code> … <code>&lt;parIdm&gt;</code></td>
<td><code>&lt;parIdn&gt;</code> … <code>&lt;parIdr&gt;</code></td>
<td><code>&lt;parIds&gt;</code> … <code>&lt;parIdz&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;value1&gt;</code> … <code>&lt;valuem&gt;</code></td>
<td><code>&lt;locn&gt;</code> … <code>&lt;locz&gt;</code></td>
<td><code>&lt;namez&gt;</code> … <code>&lt;names&gt;</code></td>
</tr>
</tbody>
</table>

**Figure 35: Structure of a call record**

#### 8.5.1 Handling of call records

The function, altstep or test case name and the actual parameter values can be retrieved by using the dot notation, e.g. `myCallRecord.parIdn` or `myCallRecord.behaviour-id` where `myCallRecord` is a pointer to a call record.
For the construction of a call the function \texttt{NEW-CALL-RECORD} is assumed to be available:

\texttt{NEW-CALL-RECORD}(myBehaviour)

creates a new call record for function, allstep or test case \textit{myBehaviour} and returns a pointer to the new record. The parameter fields of the new call record have undefined values.

\textit{myEntity.INIT-CALL-RECORD}(myCallRecord)

creates variables, timers and port references for the handling of value and reference parameters in the actual scope of the test component or module control \textit{myEntity}. The variables for the handling of value parameters are initialized with the corresponding values provided in the call record. The variables and timers for the handling of reference parameters get the provided location. In addition, they get a value of an existing variable or timer in another scope unit of the component in which the call record was created. Port references get the provided name as value for the \texttt{COMP-PORT-NAME} field.

8.6 The evaluation procedure for a TTCN-3 module

8.6.1 Evaluation phases

8.6.1.0 General

The evaluation procedure for a TTCN-3 module is structured into:

1) \textit{initialization phase};
2) \textit{update phase};
3) \textit{selection phase}; and
4) \textit{execution phase}.

The phases (2), (3) and (4) are repeated until module control terminates. The evaluation procedure is described by means of a mixture of informal text, pseudo-code and the functions introduced in the previous clauses.

8.6.1.1 Phase I: Initialization

The initialization phase includes the following actions:

\begin{itemize}
  \item[a)] Declaration and initialization of variables:
    \begin{itemize}
      \item \texttt{INIT-FLOW-GRAPHS}(); // Initialization of flow graph handling. \texttt{INIT-FLOW-GRAPHS} is explained in clause 8.6.2.
      \item \textit{Entity} := \texttt{NULL}; // \textit{Entity} will be used to refer to an entity state. An entity state either represents module control or a test component.
      \item \textit{MTC} := \texttt{NULL}; // \textit{MTC} will be used to refer to the entity state of the main test component of a test case during test case execution.
    \end{itemize}
\end{itemize}

\textbf{NOTE 1:} The global variable \texttt{CONTROL} form the control state of a module state during the interpretation of a TTCN-3 module (see clause 8.3.1).

\texttt{CONTROL} := \texttt{NULL}; // \texttt{CONTROL} will be used to refer to the control state of a module state.

\textbf{NOTE 2:} The following global variables \texttt{ALL-ENTITY-STATES}, \texttt{ALL-PORT-STATES}, \texttt{TC-VERDICT}, \texttt{DONE} and \texttt{KILLED} form the test configuration state of a module state during the interpretation of a TTCN-3 module (see clause 8.3.1).

\texttt{ALL-ENTITY-STATES} := \texttt{NULL};

\texttt{ALL-PORT-STATES} := \texttt{NULL};
b) Creation and initialization of module control:

- \[ \text{CONTROL} := \text{NEW-ENTITY} \left( \text{GET-FLOW-GRAPH} \left( <\text{moduleId}> \right), \text{false} \); \]
  
  // A new entity state is created and initialized with the start node of
  // the flow graph representing the behaviour of the control of the
  // module with the name <moduleId>. The Boolean parameter
  // indicates that module control cannot be restarted after it is
  // stopped.

- \[ \text{CONTROL}.\text{INIT-VAR-SCOPE}(); \] // New variable scope

- \[ \text{CONTROL}.\text{INIT-TIMER-SCOPE}(); \] // New timer scope

- \[ \text{CONTROL}.\text{VALUE-STACK}.\text{push} \left( \text{MARK} \right); \] // A mark is pushed onto the value stack

8.6.1.2 Phase II: Update

The update phase is related to all actions that are outside the scope of the operational semantics but influence the interpretation of a TTCN-3 module. The update phase comprises the following actions:

a) Time progress: All running timers are updated, i.e. the \text{TIME-LEFT} values of running timers are (possibly) decreased, and if due to the update a timer expires, the corresponding timer bindings are updated, i.e. \text{TIME-LEFT} is set to 0.0 and \text{STATUS} is set to \text{TIMEOUT}.

\text{NOTE 1:} The update of timers includes the update of all running \text{TIMER-GUARD} timers in module states. \text{TIMER-GUARD} timers are used to guard the execution of test cases and call operations.

b) Behaviour of the SUT: Messages, remote procedure calls, replies to remote procedure calls and exceptions (possibly) received from the SUT are put into the port queues at which the corresponding receptions shall take place.

\text{NOTE 2:} This operational semantics makes no assumptions about time progress and the behaviour of the SUT.

8.6.1.3 Phase III: Selection

The selection phase consists of the following two actions:

a) Selection: Select a non-blocked entity, i.e. an entity that has not the \text{STATUS} value \text{BLOCKED}. The entity may be \text{CONTROL}, i.e. module control, or a test component, i.e. an element of \text{ALL-ENTITY-STATES}.

b) Storage: Store the identifier of the selected entity in the global variable \text{Entity}.

8.6.1.4 Phase IV: Execution

The execution phase consists of the following two actions:

a) Execution step of the selected entity: Execute the top flow graph node in the \text{CONTROL-STACK} of \text{Entity}.

b) Check termination criterion: Stop execution if module control has terminated, i.e. \text{CONTROL} is \text{NULL}. Otherwise continue with Phase II.

\text{NOTE:} The execution step of the selected entity can be seen as a procedure call. The check of the termination criterion is done when the execution step terminates, i.e. returns the control.
8.6.2 Global functions

The evaluation procedure uses the global function \textit{INIT-FLOW-GRAPHS}:

\begin{itemize}
\item[a)] \textit{INIT-FLOW-GRAPHS} is assumed to be the function that initializes the flow graph handling. The handling may include the creation of the flow graphs and the handling of the pointers to the flow graphs and flow graph nodes.
\end{itemize}

The pseudo-code used the following clauses to describe execution of flow graph nodes use the functions \textit{CONTINUE-COMPONENT}, \textit{RETURN}, \textit{***DYNAMIC-ERROR***}:

\begin{itemize}
\item[b)] \textit{CONTINUE-COMPONENT} the actual test component continues its execution with the node lying on top of the control stack, i.e. the control is not given back to the module evaluation procedure described in this clause.
\item[c)] \textit{RETURN} returns the control back to the module evaluation procedure described in this clause. The \textit{RETURN} is the last action of the "execution step of the selected entity" of the execution phase.
\item[d)] \textit{***DYNAMIC-ERROR***} refers to the occurrence of a dynamic error. The error handling procedure itself is outside the scope of the operational semantics. If a dynamic error occurs all following behaviour of the test case is meant to be undefined. In this case resources allocated to the test case shall be cleared and the \textit{error} verdict is assigned to the test case. Control is given to the statement in the control part following the execute statement in which the error occurred. This is modelled by the flow graph segment \textit{<dynamic-error>} (clause 9.17.3).
\item[e)] \textit{APPLY-OPERATOR} used as generic function for describing the evaluation of operators (e.g. \textit{+, *, / or -}) in expressions (see clause 9.18.4).
\end{itemize}

9 Flow graph segments for TTCN-3 constructs

9.0 General

The operational semantics represents TTCN-3 behaviour in form of flow graphs. The construction algorithm for the flow graphs representing behaviour is described in clause 8.2. It is based on templates for flow graphs and flow graph segments that have to be used for the construction of concrete flow graphs for module control, test cases, altsteps, functions and component type definitions defined in a TTCN-3 module. The definitions of the templates for the flow graph segments can be found in this clause. They are presented in an alphabetical order and not in a logical order.

The flow graph segment definitions are provided in the form of figures. The flow graph nodes are presented on the left side of the figures and comments associated to nodes and flow lines are shown on the right side. Descriptive comments are presented for reference nodes and comments in form of pseudo-code are associated to basic nodes. The pseudo-code describes how a basic node is interpreted, i.e. changes the module state. It makes use of the functions defined in clause 8 and the global variables declared and initialized in the evaluation procedure for TTCN-3 modules (see clause 8.6). An overall view of all functions and keywords used by the pseudo-code can be found in clause 8.

9.1 Action statement

The syntactical structure of an \texttt{action} statement is:

\begin{verbatim}
  action (<informal description>)
\end{verbatim}

The flow graph segment \texttt{<action-stmt>} in figure 36 defines the execution of the \texttt{action} statement.
9.2 Activate statement

The syntactical structure of the activate statement is:

\[
\text{activate}(<\text{altstep-name}>([<\text{act-par-desc}_1>, \ldots, <\text{act-par-desc}_n>]))
\]

The <altstep-name> denotes to the name of an altstep that is activated as default behaviour, and <act-par-desc_1>, ..., <act-par-desc_n> describe the actual parameter values of the altstep at the time of its activation.

It is assumed that for each <act-par-desc_i> the corresponding formal parameter identifier <f-par-Id_i> is known, i.e. the syntactical structure above can be extended to:

\[
\text{activate}(<\text{altstep-name}>((<f-par-Id_1>,<\text{act-par-desc}_1>), \ldots, (<f-par-Id_n>,<\text{act-par-desc}_n>)))
\]

The flow graph segment <activate-stmt> in figure 37 defines the execution of the activate statement. The execution is structured into three steps. In the first step, a call record for the altstep <function-name> is created. In the second step the values of the actual parameter are calculated and assigned to the corresponding field in the call record. In the third step, the call record is put as first element in the DEFAULT-LIST of the entity that activates the default.

NOTE: For altsteps that are activated as default behaviour, only value parameters are allowed. In figure 37, the handling of the value parameters is described by the flow graph segment <value-par-calculation>, which is defined in clause 9.24.1.
9.2a Alive component operation

9.2a.0 General

The syntactical structure of the alive component operation is:

```
<component-expression>.alive
```

The alive component operation checks whether a component has been created and is ready to execute or is already executing a behaviour function. The component to be checked is identified by a component reference, which may be provided in form of a variable or value returning function, i.e. is an expression. For simplicity, the keywords "all component" and "any component" are considered to be special expressions.

The alive component operation distinguishes between its usage in a Boolean guard of an alt statement or blocking call operation and all other cases. If used in a Boolean guard, the result of alive component operation is based on the actual snapshot. In all other cases the alive component operation evaluates directly the module state information.

The result of the alive component operation is pushed onto the value stack of the entity, which called the operation.

The flow graph segment <alive-component-op> in figure 37a defines the execution of the running component operation.
```c
if (Entity.STATUS == ACTIVE) {
    Entity.NEXT-CONTROL(true);
} else { // Entity is in a snapshot
    Entity.NEXT-CONTROL(false);
} RETURN;
```

Figure 37a: Flow graph segment <alive-component-op>
9.2a.1 Flow graph segment <alive-comp-act>

The flow graph segment <alive-comp-act> in figure 37b describes the execution of the alive component operation outside a snapshot, i.e. the entity is in the status **ACTIVE**.

```plaintext
if (Entity.VALUE-STACK.top() == 'all component') {
    if (Entity != MTC) { ***DYNAMIC-ERROR*** // 'all component' is not allowed }
    else { if (KILLED.length() == 0) { // no entity has terminated 
            Entity.VALUE-STACK.push(true); 
        } else { // at least one component has terminated 
            Entity.VALUE-STACK.push(false); 
        } }
} else { if (Entity.VALUE-STACK.top() == 'any component') {
    if (Entity != MTC) { ***DYNAMIC-ERROR*** // 'any component' is not allowed }
    else if (ALL-ENTITY-STATES.length() > 1) { 
        // at least one PTC is alive 
        Entity.VALUE-STACK.push(true); 
    } else { 
        Entity.VALUE-STACK.push(false); 
    } }
} else { if (ALL-ENTITY-STATES.member(Entity.VALUE-STACK.top())) { 
    // Specified component is alive 
    Entity.VALUE-STACK.push(true); 
} else { 
    Entity.VALUE-STACK.push(false); 
} }
}
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 37b: Flow graph segment <alive-comp-act>
9.2a.2 Flow graph segment <alive-comp-snap>

The flow graph segment <alive-comp-snap> in figure 37c describes the execution of the alive component operation during the evaluation of a snapshot, i.e. the entity is in the status SNAPSHOT.

```c
if (Entity.VALUE-STACK.top() == 'all component') {
  if (Entity != MTC) // 'all component' is not allowed
    ***DYNAMIC-ERROR***
  else {
    if (Entity.SNAP-KILLED.length() == 0) {
      Entity.VALUE-STACK.push(true);
    } else {
      Entity.VALUE-STACK.push(false);
    }
  }
} else {
  if (Entity.VALUE-STACK.top() == 'any component') {
    if (Entity != MTC) // 'any component' is not allowed
      ***DYNAMIC-ERROR***
    else {
      if (Entity.SNAP-ALIVE.length() > 1) {
        // at least one PTC was alive when the
        // snapshot has been taken
        Entity.VALUE-STACK.push(true);
      } else {
        Entity.VALUE-STACK.push(false);
      }
    }
  } else {
    if (Entity.SNAP-ALIVE.member(Entity.VALUE-STACK.top())) {
      // Component was alive when the snapshot was taken
      Entity.VALUE-STACK.push(true);
    } else {
      // Component was not alive when the snapshot was taken
      Entity.VALUE-STACK.push(false);
    }
  }
}
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 37c: Flow graph segment <alive-comp-snap>

9.3 Alt statement

9.3.0 General

The alt statement is the most complicated and important statement of TTCN-3. It implements the snapshot semantics and specifies the branching due to the reception of messages, replies, calls and exceptions, due to the occurrence of timeouts and due to the termination of components. In addition, the evocation of the TTCN-3 default mechanism is also related to the alt statement.

The flow graph representation of the alt statement in figure 38. The different alternatives due to the reception of messages, replies, calls and exceptions, due to the occurrence of timeouts and due to the termination of components are hidden in the flow graph segment <receiving-branch>.
segment <alt-stmt>

- <take-snapshot>
  // A snapshot is taken

+ <receiving-branch> OR <altstep-call-branch> OR <else-branch>
  // The different alternatives // are evaluated

- <default-evocation>
  // The default mechanism may // be evoked.

if (Entity.STATUS == ACTIVE) {
    Entity.NEXT-CONTROL(true);
} else {
    if (Entity.STATUS == BREAK) {
        // altstep is left via a break statement.
        Entity.STATUS(Active);
        Entity.NEXT-CONTROL(true);
    } else {
        // A new snapshot needs to be taken, the
        // status of the entity is SNAPSHOT (none
        // of the alternatives could be selected
        // and executed) or REPEAT (due to a
        // repeat statement)
        Entity.NEXT-CONTROL(false);
    }
} RETURN;

Figure 38: Flow graph segment <alt-stmt>
9.3.1 Flow graph segment <take-snapshot>

The flow graph segment <take-snapshot> in figure 39 describes the procedure of taking a snapshot. The snapshot records values of ports, timers and stopped components.

```plaintext
// Take Snapshot
SNAP-PORTS(Entity);  // Ports
Entity.SNAP-TIMER();  // Timer

Entity.SNAP-ALIVE := copy(ALL-ENTITY-STATES);  // ALIVE
Entity.SNAP-DONE := copy(DONE);  // DONE
Entity.SNAP-KILLED := copy(KILLED);  // KILLED

Entity.STATUS := SNAPSHOT;  // new component status
Entity.DEFAULT-POINTER := Entity.DEFAULT-LIST.first();

Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 39: Flow graph segment <take-snapshot>
9.3.2 Flow graph segment <receiving-branch>

The execution of the flow graph segment <receiving-branch> is shown in figure 40.

---

**Figure 40: Flow graph segment <receiving-branch>**
9.3.3 Flow graph segment `<altstep-call-branch>`

The invocation of an altstep within an `alt` statement is described by the flow graph segment `<altstep-call-branch>` in figure 41.

```plaintext
<altstep-call-branch>

// The branch is only evaluated,
// if the entity is in status SNAPSHOT
if (Entity.STATUS == SNAPSHOT) {
    Entity.NEXT-CONTROL(true);
} else {
    Entity.NEXT-CONTROL(false);
}
RETURN;

// Boolean expression that
// guards a branch
<expression>

Entity.NEXT-CONTROL(Entity.VALUE-STACK.top());
Entity.VALUE-STACK.pop();
RETURN;

// The altstep is called, the status of the
// entity may be changed inside the altstep
// by the different alternatives in the
// altstep.
<altstep-call>

// STATUS of Entity is ACTIVE if
// one of the alternatives in the
// altstep has been executed
if {Entity.STATUS == ACTIVE} {
    Entity.NEXT-CONTROL(true);
} else {
    Entity.NEXT-CONTROL(false);
}
RETURN;

// Execution of optional statement
// block
<statement-block>
```

Figure 41: Flow graph segment `<altstep-call-branch>`
9.3.4 Flow graph segment \(<\text{else-branch}>\)

The execution of an \texttt{else} branch within an \texttt{alt} statement is described by the flow graph segment \(<\text{else-branch}>\) in figure 42.

```plaintext
// An else-branch is always selected, i.e.,
// status of Entity will be set of ACTIVE
Entity.STATUS := ACTIVE;

// The statement block in an else branch
// is always executed.
<statement-block>

// The branch is only evaluated,
// if the entity is in status SNAPSHOT
if (Entity.STATUS == SNAPSHOT) {
    Entity.NEXT-CONTROL(true);
} else {
    Entity.NEXT-CONTROL(false);
} RETURN;
```

Figure 42: Flow graph segment \(<\text{else-branch}>\)
9.3.5 Flow graph segment <default-evocation>

The evocation of defaults behaviour at the end of `alt` statements is described by the flow graph segment `<default-evocation>` in figure 43.

```plaintext
// A default is only evoked, if the entity is in status SNAPSHOT
if (Entity.STATUS == SNAPSHOT) {
    Entity.NEXT-CONTROL(true);
} else {
    Entity.NEXT-CONTROL(false);
}
RETURN;
```

```plaintext
// A call record in DEFAULT-LIST, identified by DEFAULT-POINTER is pushed onto the VALUE-STACK of Entity. Afterwards DEFAULT-POINTER is updated, i.e., will point to the next record in DEFAULT-LIST. If DEFAULT-POINTER is NULL, the Entity status will not change and, thus, a new SNAPSHOT will be initiated in <alt-stmt>
if (Entity.DEFAULT-POINTER == NULL) {
    Entity.NEXT-CONTROL(false);
} else {
    Entity.VALUE-STACK.push(Entity.DEFAULT-POINTER);
    Entity.DEFAULT-POINTER = Entity.DEFAULT-LIST.next(Entity.DEFAULT-POINTER);
    Entity.NEXT-CONTROL(true);
}
RETURN;
```

```plaintext
// The actual default altstep is invoked or called like a user defined function.
```

```plaintext
// Jump back to the beginning of the segment to check if the next default behaviour has to be invoked.
```

Figure 43: Flow graph segment <default-evocation>
9.4 Altstep call

As shown in figure 44, the call of an altstep is handled like a function call.

```
segment <altstep-call>

  <function-call>

  // Reference to the flow graph segment
  // describing the function call
```

Figure 44: Flow graph segment <altstep-call>

9.5 Assignment statement

The syntactical structure of an assignment statement is:

```
<varId> := <expression>
```

The value of the expression `<expression>` is assigned to variable `<varId>`. The execution of an assignment statement is defined by the flow graph segment `<assignment-stmt>` in figure 45.

```
segment <assignment-stmt>

  <expression>

  // The expression is evaluated and the
  // result is pushed onto the value stack

  Entity.VAR-SET(varId, Entity.VALUE-STACK.top());
  Entity.VALUE-STACK.pop();
  Entity.NEXT-CONTROL(true);
  RETURN;
```

Figure 45: Flow graph segment <assignment-stmt>

9.5a Break statements in altsteps

The syntactical structure of the break statement in an altstep is:

```
break
```

NOTE: The semantics of a break statement used for leaving a loop, an interleave or an alt statement is defined in clause 7.8 as a shorthand form for using a pair of goto-label statements.
Basically, the **break** statement used for leaving an altstep is a **return** statement without return value, which also changes the entity status to **BREAK**. The status **BREAK** prevents the re-evaluation of the **alt** statement in which the altstep has been called statement has been called and also prevents the execution of the optional statement block following the altstep call in the **alt** statement. The break statement also works for altsteps called indirectly by the default mechanism. In this case, the alt statement that invokes the default mechanism is left. The flow graph segment `<break-altstep-stmt>` shown in figure 45a defines the execution of the **break** statement for leaving an altstep.

**Figure 45a: Flow graph segment `<break-altstep-stmt>`**

### 9.6 Call operation

#### 9.6.0 General

The syntactical structure of the **call** operation is:

```plaintext
<portId>.call (<callSpec> [ <blocking-info> ]) [ to <receiver-spec> ] [ <call-reception-part> ]
```

The optional `<blocking-info>` consists of either the keyword **nowait** or a duration for a timeout exception. The optional `<receiver-spec>` in the **to** clause refers to the receivers of the call. In case of a one-to-one communication, the `<receiver-spec>` addresses a single entity (including the SUT or an entity within the SUT). In case of multicast or broadcast communication, the `<receiver-spec>` specifies a set or all test components connected via the specified port with the calling component. The optional `<call-reception-part>` denotes the alternative receptions in case of a blocking **call** operation.

The operational semantics distinguishes between **blocking** and **non-blocking** **call** operations. A **call** is non-blocking if the keyword **nowait** is used in the **call** operation, or if the called procedure is non-blocking, i.e. defined by using the keyword **noblock**. A blocking **call** has a `<call-reception-part>`.

The flow graph segment `<call-op>` in figure 46 defines the execution of a **call** operation. It reflects the distinction between blocking and non-blocking calls.

**Figure 46: Flow graph segment `<call-op>`**

For blocking and non-blocking call operations a receiver entity may be specified in form of an expression. The possibilities are shown in figures 47 and 48.

**Figure 47: Flow graph segment <blocking-call-op>**

<table>
<thead>
<tr>
<th><strong>segment</strong> &lt;blocking-call-op&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;b-call-without-duration&gt; OR &lt;b-call-with-duration&gt;</td>
</tr>
<tr>
<td>// A blocking call may or may not be supervised by TIMER-GUARD</td>
</tr>
</tbody>
</table>

**Figure 48: Flow graph segment <non-blocking-call-op>**

<table>
<thead>
<tr>
<th><strong>segment</strong> &lt;non-blocking-call-op&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;nb-call-with-one-receiver&gt; OR &lt;nb-call-with-multiple-receivers&gt; OR &lt;nb-call-without-receiver&gt;</td>
</tr>
<tr>
<td>// A non-blocking call may address one, multiple (multicast and broadcast) or no receiver entities.</td>
</tr>
</tbody>
</table>
9.6.1  Flow graph segment <nb-call-with-one-receiver>

The flow graph segment `<nb-call-with-one-receiver>` in figure 49 defines the execution of a non-blocking `call` operation where one receiver is specified in form of an expression.

```plaintext
segment <nb-call-with-one-receiver>

<expression>

// The expression shall evaluate
// to a component reference or
// address value

nb-call-with-one-receiver (portId, callSpec)

let {
    var receiver := Entity.VALUE-STACK.top();
    var remotePort := GET-REMOTE-PORT(Entity, Entity.portId.COMP-PORT-NAME, receiver);
    if (remotePort == NULL) {
        ***DYNAMIC-ERROR***; // Remote port cannot be found
    } else {
        // sending of call
        remotePort.enqueue(CONSTRUCT-ITEM(Entity, call, callSpec));
    }
} // end of scope of receiver and remotePort
Entity.VALUE-STACK.pop();  // clean value stack
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 49: Flow graph segment `<nb-call-with-one-receiver>`

9.6.1a  Flow graph segment `<nb-call-with-multiple-receivers>`

The flow graph segment `<nb-call-with-multiple-receivers>` in figure 49a defines the execution of a non-blocking `call` operation where multiple receivers are addressed. In case of broadcast communication the keyword `all component` is used as receiver specification. In case of multicast communication a list of expressions is provided which shall evaluate to component references or address values.

The component references or address values of the addressed entities (or the keyword `all component`) are pushed onto the value stack of the calling entity. The number of references or address values stored in the value stack is considered to be known, i.e. it is the parameter `number` of the basic flow graph node `<nb-call-with-multiple-receivers>` in figure 49a. The number parameter is 1 in case of broadcast communication, i.e. the keyword `all component` is top element in the value stack.
let | // loop counter variable
    var i; // variable for connections in port states
    var receiver; // variable for receiver component references or
        // receiver address values
    var localPort, remotePort; // variables for port references
    localPort := Entity.portId.COMP-PORT-NAME; // local port
    if (Entity.VALUE-STACK.top() == all component) {} // all component
        connection := localPort.CONNECTIONS-LIST.next(connection);
        while (connection != NULL) {
            remotePort := connection.REMOTE-PORT-NAME;
            if (remotePort == SYSTEM) {} // Port is mapped onto a port of the test system
                                                // reception of the reply by the SUT is outside
                                                // the scope of the operational semantics
                
                } else { // sending of call
                    remotePort.enqueue(CONSTRUCT-ITEM(Entity, call, callSpec));
                
                } else {} // end of local scope
            
        }
    else {
        for (i == 1; i <= number; i := i+1) {
            receiver := Entity.VALUE-STACK.top();
            Entity.VALUE-STACK.pop(); // clean value stack
            remotePort := GET-REMOTE-PORT(Entity, localPort, receiver);
            if (remotePort == NULL) {} // Remote port cannot be found
                
                } if (remotePort == SYSTEM) {
                    // Port is mapped onto a port of the test system
                    // reception of the reply by the SUT is outside
                    // the scope of the operational semantics
                
                } else { // sending of call
                    remotePort.enqueue(CONSTRUCT-ITEM(Entity, call, callSpec));
                
                } 
        } // end of local scope
    Entity.NEXT-CONTROL(true);
    RETURN;

Figure 49a: Flow graph segment <nb-call-with-multiple-receivers>
9.6.2 Flow graph segment <nb-call-without-receiver>

The flow graph segment <nb-call-without-receiver> in figure 50 defines the execution of a non-blocking call operation without a to-clause.

```plaintext
segment <nb-call-without-receiver-op>

let {
  var remotePort := GET-REMOTE-PORT(Entity, Entity.portId.COMP-PORT-NAME, NONE);
  if (remotePort == NULL) {
    ***DYNAMIC-ERROR***; // Remote port cannot be found
  } else { // sending of call
    remotePort.enqueue(CONSTRUCT-ITEM(Entity, call, callSpec));
  }
} // end of scope of remotePort

Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 50: Flow graph segment <nb-call-without-receiver>

9.6.3 Flow graph segment <b-call-without-duration>

Blocking calls are modelled by a non-blocking call followed by the body of the call, which handles the replies and exceptions. The flow graph segment <b-call-without-duration> shown in figure 51 describes the execution of a blocking call without a given duration as time guard.

```plaintext
segment <b-call-without-duration>

<nb-call-with-one-receiver> OR <nb-call-with-multiple-receivers> OR <nb-call-without-receiver> // Call of remote procedure

<call-reception-part> // Handling of replies and exceptions of the called procedure.
```

Figure 51: Flow graph segment <b-call-without-duration>
9.6.4 Flow graph segment `<b-call-with-duration>`

The flow graph segment `<b-call-with-duration>` (see figure 52) describes the execution of a blocking call with a duration as time guard.

![Flow graph segment diagram](image-url)

**Figure 52: Flow graph segment `<b-call-with-duration>`**
9.6.5 Flow graph segment <call-reception-part>

The flow graph segment <call-reception-part> (see figure 53) describes the handling of replies, exceptions and the timeout exception of a blocking call operation.

```
if (Entity.STATUS == ACTIVE) {
    Entity.NEXT-CONTROL(true);
    // To assure a defined state of Entity
    Entity.TIMER-GUARD.STATUS := IDLE;
}
else { // A new snapshot needs to be taken, the
    // status of the entity is SNAPSHOT (none
    // of the alternatives could be selected
    // and executed)
    Entity.NEXT-CONTROL(false);
}
RETURN;
```

Figure 53: Flow graph segment <call-reception-part>
9.6.6 Flow graph segment <catch-timeout-exception>

The flow graph segment <catch-timeout-exception> (see figure 54) is for the handling of a timeout exception of a blocking call operation that is guarded by a duration.

```plaintext
if (Entity.TIMER-GUARD.STATUS == TIMEOUT) {
    Entity.NEXT-CONTROL(true);
    // To assure a defined state of Entity
    Entity.STATUS := ACTIVE;
} else { // continue evaluation
    Entity.NEXT-CONTROL(false);
}
RETURN;
```

Figure 54: Flow graph segment <catch-timeout-exception>

9.7 Catch operation

The syntactical structure of the catch operation is:

```plaintext
<portId>.catch (<matchingSpec>) [from <component_expression>] -> [<assignmentPart>]
```

Apart from the catch keyword this syntactical structure is identical to the syntactical structure of the receive operation. Therefore, the operational semantics handles the catch operation in the same manner as the receive operation. This is also shown in the flow graph segment <catch-op> (figure 55), which defines the execution of a catch operation. The figure refers to flow graph segments related to the receive operation (see clause 9.37).

Figure 55: Flow graph segment <catch-op>
9.8 Check operation

9.8.0 General

The syntactical structure of the check operation is:

```plaintext
<portId>.check( receive|getcall|catch|getreply (<matchingSpec>)
    [from <component-expression>]) [⇒ <assignmentPart>]
```

The optional `<component-expression>` in the from clause refers to the sender entity. It may be provided in form of a variable value or the return value of a function, i.e. it is assumed to be an expression. The optional `<assignmentPart>` denotes the assignment of received information if the received information matches to the matching specification `<matchingSpec>` and to the (optional) from clause.

The operational semantics handles the operations `receive`, `getcall`, `catch` and `getreply` in the same manner, i.e. they are described by referencing the same flow graph segments `<receive-with-sender>` and `<receive-without-sender>`. The check operation also handles the different operations in the same manner. Thus the flow graph segment `<check-op>` in figure 56, which defines the execution of the check operation, also references only two flow graph segments. The only difference to the flow graph segments `<receive-with-sender>` and `<receive-without-sender>` is that the received items are not deleted after the match.

---

**Figure 56: Flow graph segment `<check-op>`**
9.8.1 Flow graph segment <check-with-sender>

The flow graph segment <check-with-sender> in figure 57 defines the execution of a check operation where the sender entity is specified in form of an expression.

```plaintext
let () // local scope for portRef and sender
  var portRef := NULL;
  var sender := Entity.VALUE-STACK.top(); // Sender
  Entity.VALUE-STACK.pop(); // Clean value stack
  if (portID == "any port") {
    portRef := ALL-PORT-STATES.random(MATCH-ITEM(SNAP-VALUE, matchingSpec, sender) && OWNER == Entity);
    if (portRef == NULL) { // no 'matching' port found
      Entity.NEXT-CONTROL(false);
      RETURN;
    }
  } else {
    portRef := GET-PORT(Entity, Entity.portId.COMP-PORT-NAME) // Specified port
  }
  // MATCHING
  if (PortRef.first() == NULL) { // Port queue is empty, no match
    Entity.NEXT-CONTROL(false);
    RETURN;
  } else {
    if (MATCH-ITEM(portRef.SNAP-VALUE, matchingSpec, sender)) {
      // The message in the queue matches
      Entity.VALUE-STACK.push(portRef); // Saving port reference
      Entity.STATUS := ACTIVE; // successful match, Entity status is changed
      // from SNAPSHOT to ACTIVE
      Entity.NEXT-CONTROL(true);
    } else {
      // The top item in the queue does not match
      Entity.NEXT-CONTROL(false);
    }
    RETURN;
  }
} // End of scope of portRef and sender
```

Figure 57: Flow graph segment <check-with-sender>
9.8.2 Flow graph segment `<check-without-sender>`

The flow graph segment `<check-without-sender>` in figure 58 defines the execution of a `check` operation without a `from` clause.

```plaintext
let { // local scope
  var portRef := NULL;
  if (portID == "any port") {
    portRef := ALL-PORT-STATES.MATCH-ITEM(SNAP-VALUE, matchingSpec, NONE)
    && OWNER == Entity;
    if (portRef == NULL) { // no 'matching' port found
      Entity.NEXT-CONTROL(false);
      RETURN;
    }
  }
  else {
    portRef := GET-PORT(Entity, Entity.portId.COMP-PORT-NAME); // Specified port
  }
  // MATCHING
  if (PortRef.first() == NULL) { // Port queue is empty, no match
    Entity.NEXT-CONTROL(false);
    RETURN;
  } else {
    if (MATCH-ITEM(portRef.SNAP-VALUE, matchingSpec, NONE)) { // The message in the queue matches
      Entity.VALUE-STACK.push(portRef); // Saving port reference
      Entity.STATUS := ACTIVE; // successful match, Entity status is changed
      Entity.NEXT-CONTROL(true);
    } else { // The first item in the queue does not match
      Entity.NEXT-CONTROL(false);
    }
    RETURN;
  }
} // End of scope
```

Figure 58: Flow graph segment `<check-without-sender>`
9.8a Checkstate port operation

9.8a.0 General

The syntactical structure of the checkstate port operation is:

\[ <\text{portId}>,\text{checkstate}(\text{<charstring-expression>}) \]

The checkstate port operation allows to examine the state of a port. If a port is in the state specified by the charstring parameter, the checkstate operation returns the Boolean value \text{true}. If the port is not in the specified state, the checkstate operation returns the Boolean value \text{false}. Calling the checkstate operation with an invalid argument leads to an error. For simplicity, the keywords "all port" and "any port" are considered to be special values of \text{<portId>}.

The result of the checkstate port operation is pushed onto the value stack of the entity, which called the operation.

The flow graph segment <checkstate-port-op> in figure 58a defines the execution of the running component operation.

Figure 58a: Flow graph segment <checkstate-port-op>
9.8a.1 Flow graph segment <check-port-status>

The flow graph segment `<check-port-status>` in figure 58b describes the execution of the `checkstate` component operation by checking for the `STATUS` field in port states (see clause 8.3.3).

```plaintext
let // local scope
  var portId; // for storing the portId
  var checkstate-par; // checkstate parameter to be checked for
  var checkState; // port state to be checked for
  var result; // Boolean for intermediate results
  var port;

  portId := Entity.VALUE-STACK.top();
  Entity.VALUE-STACK.pop();

  checkstate-par := Entity.VALUE-STACK.top();
  Entity.VALUE-STACK.pop();

  if (checkstate-par == "Started") checkState := STARTED;
  if (checkstate-par == "Halted") checkState := HALTED;
  if (checkstate-par == "Stopped") checkState := STOPPED;

  if (Entity.PORT-REF == NULL) { // Entity has no ports
    result := false;
  } else if (portId == 'all port') {
    port := ALL-PORT-STATES.first();
    result := true;
    while (port != NULL and result == true) {
      if (port.OWNER == Entity) {
        if (port.STATUS == checkState) result := false;
      }
      port := ALL-PORT-STATES.next();
    }
  } else if (portId == 'any port') {
    port := ALL-PORT-STATES.first();
    result := false;
    while (port != NULL and result == false) {
      if (port.OWNER == Entity) {
        if (port.STATUS == checkState) result := true;
      }
      port := ALL-PORT-STATES.next();
    }
  } else {
    port := Entity.portId.COMP-PORT-NAME;
    if (port == NULL) { // port cannot be retrieved
      ***DYNAMIC-ERROR***
    } else {
      if (port.STATUS == checkState) result := true;
      if (port.STATUS != checkState) result := false
    }
  }

  Entity.VALUE-STACK.push(result);
}
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 58b: Flow graph segment `<check-port-status>`

9.8a.2 Flow graph segment `<check-port-connection>`

The flow graph segment `<check-port-connection>` in figure 58c describes the execution of the `checkstate` component operation by investigating the `CONNECTIONS-LIST` in port states (see clause 8.3.3).
let \{ // local scope
  var portId; // for storing the portId
  var checkstate-par; // checkstate parameter to be checked for
  var result; // Boolean for intermediate results
  var isNotLinked := false; // Boolean for intermediate results
  var isMapped := false; // Boolean for intermediate results
  var isConnected := false; // Boolean for intermediate results
  var singleport := false; // Boolean for intermediate results
  var port;

  portId := Entity.VALUE-STACK.top();
  Entity.VALUE-STACK.pop();
  checkstate-par := Entity.VALUE-STACK.top();
  Entity.VALUE-STACK.pop();

  if (portId == 'any port' or portId == 'all port') {
    singleport := false;
    port := ALL-PORT-STATES.first();
  } else {
    singleport := true;
    port := Entity.portId.COMP-PORT-NAME;
  }

  while (port != NULL) {
    if (port.OWNER == Entity) {
      if (port.CONNECTIONS-LIST == NULL) {
        isNotLinked := true; // unlinked port
      } else {
        isNotMapped := true; // mapped port
      }
    } else { // more than one connection
      isConnected := true; // connected port
    }

    if (singleport == false) port := ALL-PORT-STATES.next();
    if (singleport == true) port := NULL;

    if (portId == 'any port') {
      if (checkstate-par == "Connected") result := isConnected;
      if (checkstate-par == "Mapped") result := isMapped;
      if (checkstate-par == "Linked") result := (isMapped or isConnected);
    } else { // portId is a single port or 'all port'
      if (checkstate-par == "Connected") {
        result := (isConnected and not(isMapped or isNotLinked));
      } else if (checkstate-par == "Mapped") {
        result := (isMapped and not(isConnected or isNotLinked));
      } else { // checkstate-par == "Linked"
        result := (isMapped or isConnected) and not(isNotLinked);
      }
    }
  }

  Entity.VALUE-STACK.push(result);
}

Entity.NEXT-CONTROL(true);
RETURN;

Figure 58c: Flow graph segment <check-port-connection>
9.9 Clear port operation

The syntactical structure of the clear port operation is:

<portId>.clear

The flow graph segment <clear-port-op> in figure 59 defines the execution of the clear port operation.

```javascript
let { // Begin of local scope
  var portRef := NULL
  var portState := NULL;
  if (portId == "all port") {
    portState := ALL-PORT-STATES.first();
    while (portState != NULL) {
      if (portState.OWNER == Entity) {
        portState.VALUE-QUEUE.clear();
      }
      portState := ALL-PORT-STATES.next(portState);
    }
  } else {
    portRef := Entity.portId.COMP-PORT-NAME;
    GET-PORT(Entity, portRef).clear();
  } // End of scope
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 59: Flow graph segment <clear-port-op>

9.10 Connect operation

The syntactical structure of the connect operation is:

connect(<component-expression_1>:<portId1>, <component-expression_2>:<portId2>)

The identifiers <portId1> and <portId2> are considered to be port identifiers of the corresponding test components. The components to which the ports belong are referenced by means of the component references <component-expression_1> and <component-expression_2>. The references may be stored in variables or is returned by a function, i.e. they are expressions, which evaluate to component references. The value stack is used for storing the component references.

The execution of the connect operation is defined by the flow graph segment <connect-op> shown in figure 60. In the flow graph description the first expression to be evaluated refers to <component-expression_1> and the second expression to <component-expression_2>, i.e. the <component-expression_2> is on top of the value stack when the connect-op node is executed.
Figure 60: Flow graph segment <connect-op>

9.11 Constant definition

The syntactical structure of a constant definition is:

```plaintext
const <constType> <constId> := <constType-expression>
```

The value of a constant is considered to be an expression that evaluates to a value of the type of the constant.

**NOTE:** Global constants are replaced by their values in a pre-processing step before this semantics is applied (see clause 9.2). Local constants are treated like variable declarations with initialization. The correct usage of constants, i.e. constants should never occur on the left side of an assignment, should be checked during the static semantics analysis of a TTCN-3 module.

The flow graph segment <constant-definition> in figure 61 defines the execution of a constant declaration where the value of the constant is provided in form of an expression.
9.12 Create operation

The syntactical structure of the `create` operation is:

```
<componentTypeId>.create [alive]
```

A present `alive` clause indicates that the created component can be restarted after it has been stopped. Presence and absence of the alive clause is handled as a Boolean flag in the operational semantics (see `alive` parameter of the basic flow graph node `create-op` in figure 62).

The flow graph segment `<create-op>` in figure 62 defines the execution of the `create` operation.
Figure 62: Flow graph segment <create-op>

9.13 Deactivate statement

9.13.0 General

The syntactical structure of a **deactivate** statement is:

```plaintext
deactivate [([<default-expression>])]  
```

The **deactivate** statement specifies the deactivation of one or all active defaults of the entity that executes the **deactivate** statement. If one default shall be deactivated, the optional `<default-expression>` shall evaluate to a default reference which identifies the default to be deactivated. The call of a **deactivate** statement without `<default-expression>` deactivates all active defaults.

The execution of a **deactivate** statement is defined by the flow graph segment `<deactivate-stmt>` in figure 63a.
9.13.1 Flow graph segment <deactivate-one-default>

The flow graph segment <deactivate-one-default> in figure 63b specifies the deactivation of one active default. The value of the expression <default-expression> shall evaluate to a default reference. The expression may be provided in form of a variable value or value returning function. The deactivate statement removes the specified default from the DEFAULT-LIST of the entity that executes the deactivate statement.

```
Entity.DEFAULT-LIST.delete(Entity.VALUE-STACK.top());
Entity.VALUE-STACK.pop(); // clean value stack
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 63b: Flow graph segment <deactivate-one-default>

9.13.2 Flow graph segment <deactivate-all-defaults>

The flow graph segment <deactivate-all-defaults> in figure 63c specifies the deactivation of all active defaults. The deactivate statement clears the DEFAULT-LIST of the entity that executes the deactivate statement.

```
Entity.DEFAULT-LIST := NULL;
RETURN;
```

Figure 63c: Flow graph segment <deactivate-all-defaults>
9.14 Disconnect operation

9.14.0 General

The syntactical structure of the disconnect operation is:

\[
\text{\texttt{disconnect}}\left(\text{\texttt{<component-expression}_1:portId1}} \left[\text{, \texttt{<component-expression>_2}:portId2}\right]\right)
\]

The identifiers \texttt{<portId1>} and \texttt{<portId2>} are considered to be port identifiers of the corresponding test components. The components to which the ports belong are referenced by means of the component references \texttt{<component-expression>_1>} and \texttt{<component-expression>_2>}. The references may be stored in variables or are returned by functions, i.e. they are expressions, which evaluate to component references. The value stack is used for storing the component references.

The disconnect operation can be used with one parameter pair and with two parameters pairs. The usage of the disconnect operation with one parameter pair may disconnect connections for one component or, if executed by the MTC for all components. The usage of the disconnect operation with two parameter pairs allows to disconnect specific connections.

Both usages are distinguished in the flow graph segment \texttt{<disconnect-op>} shown in figure 64, which defines the execution of the disconnect operation.

![Figure 64: Flow graph segment <disconnect-op>](image)

9.14.1 Flow graph segment <disconnect-one-par-pair>

The flow graph segment \texttt{<disconnect-one-par-pair>} shown in figure 64 defines the execution of the disconnect operation with one parameter pair. In the flow graph segment three cases are distinguished:

1) the \texttt{mTC} disconnects all connections of all components;
2) all connections of one component are disconnected; and
3) all connections of one port of one component are disconnected. In the flow graph segment the expression to be evaluated refers to \texttt{<component-expression>_1>} (see syntactical structure of the disconnect operation in clause 9.14).
segment <disconnect-one-par-pair>

<expression>

if (Entity.VALUE-STACK.top() == "all component") {
    if (Entity != MTC) OR (Entity == MTC && portId != "all port") {
        ***DYNAMIC-ERROR***
    } else {
        Entity.VALUE-STACK.pop();
        Entity.NEXT-CONTROL(true);
    }
} else {
    Entity.VALUE-STACK.push(portId);
    Entity.NEXT-CONTROL(false);
}
RETURN;

else {
    Entity.VALUE-STACK.push(portId);
    Entity.NEXT-CONTROL(false);
}
RETURN;

if (Entity.VALUE-STACK.top() == "all port") {
    Entity.VALUE-STACK.pop();
    Entity.NEXT-CONTROL(true);
} else {
    Entity.NEXT-CONTROL(false);
}
RETURN;

Figure 64a: Flow graph segment <disconnect-one-par-pair>
9.14.2 Flow graph segment <disconnect-all>

The flow segment <disconnect-all> defines the disconnection of all components at all connected ports.

```plaintext
segment <disconnect-all>

let { // local scope
    var port := ALL-PORT-STATES.first();
    var connection;
    while (port != NULL) {
        connection := port.CONNECTIONS.first();
        while (connection != NULL) {
            if (connection.REMOTE-ENTITY == system) {
                connection := NULL; // mapped port
            } else {
                port.CONNECTIONS.delete(connection);
                connection := port.CONNECTIONS.first();
            }
        }
        port := ALL-PORT-STATES.next(port);
    }
} // End of local scope

Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 64b: Flow graph segment <disconnect-all>
9.14.3 Flow graph segment <disconnect-comp>

The flow segment `<disconnect-comp>` defines the disconnection of all ports of a specified component.

```plaintext
let { // local scope
  var comp := Entity.VALUE-STACK.top();
  var connection;
  var port := ALL-PORT-STATES.first();

  while (port != NULL) {
    connection := port.CONNECTIONS.first();
    while (connection != NULL) {
      if (connection.REMOTE-ENTITY == system) {
        connection := port.CONNECTIONS.next(connection);
      } else if (connection.REMOTE-ENTITY == comp
                 or (port.OWNER == comp)) {
        port.CONNECTIONS.delete(connection);
        connection := port.CONNECTIONS.first();
      } else {
        connection := port.CONNECTIONS.next(connection);
      }
    }
    port := ALL-PORT-STATES.next(port);
  }

  Entity.VALUE-STACK.pop(); // clear value stack
} // End of local scope

Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 64c: Flow graph segment `<disconnect-comp>`
9.14.4 Flow graph segment <disconnect-port>

The flow segment <disconnect-port> defines the disconnection of a specified port.

```plaintext
let { // local scope
  var portId, rPortId;
  var comp, rComp;
  var port;
  portId := Entity.VALUE-STACK.top();
  Entity.VALUE-STACK.pop();
  comp := Entity.VALUE-STACK.top();
  Entity.VALUE-STACK.pop();
  port := GET-PORT(comp, portId);
  var connection := port.CONNECTIONS.first();
  while (connection != NULL) {
    if (connection.REMOTE-ENTITY == SYSTEM) {
      ***DYNAMIC-ERROR*** // port is not a connected port
    } else {
      rComp := connection.REMOTE-ENTITY;
      rPortId := connection.REMOTE-PORT-NAME;
      DEL-CON(comp, portId, rComp, rPortId);
      DEL-CON(rComp, rPortId, comp, portId);
      connection := port.CONNECTIONS.first();
    }
  }
} // End of local scope
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 64d: Flow graph segment <disconnect-port>

9.14.5 Flow graph segment <disconnect-two-par-pairs>

The flow graph segment <disconnect-two-par-pairs> shown in figure 64e defines the execution of the disconnect operation with two parameter pairs which disconnects specific connections. In the flow graph segment the first expression to be evaluated refers to <component-expression₁> (see syntactical structure of the disconnect operation in clause 9.14) and the second expression to <component-expression₂>, i.e. the <component-expression₂> is on top of the value stack when the disconnect-two node is executed.
9.15 Do-while statement

The syntactical structure of the do-while statement is:

```plaintext
do <statement-block>
while (<boolean-expression>)
```

The execution of a do-while statement is defined by the flow graph segment <do-while-stmt> shown in figure 65.
9.16 Done component operation

9.16.0 General

The syntactical structure of the done component operation is:

\[
\text{<component-expression>}.\text{done} [\rightarrow \text{<assignmentPart>}]\
\]

The done component operation checks whether a component is running or has stopped. Depending on whether a checked component is running or has stopped the done operation decides how the flow of control continues. Using a component reference identifies the component to be checked. The reference may be stored in a variable or be returned by a function, i.e. it is an expression. For simplicity, the keywords "all component" and "any component" are considered to be special expressions.

The optional <assignmentPart> allows the retrieval of the local verdict of the addressed component at the time of its stopping. The assignment part identifies a variable of type verdicttype to which the retrieved verdict is assigned.

The flow graph segment <done-op> in figure 66 defines the execution of the done component operation.
<expression>

let { // local scope
  var aliveNr := Entity.SNAP-ALIVE.length();
  var doneNr := Entity.SNAP-DONE.length();
  var killedNr := Entity.SNAP-KILLED.length();
  var nonWaitingNr := aliveNr - doneNr - killedNr;
  // number of alive entities which are executing a behaviour
  // or neither have stopped and nor have terminated.
  var doneEntity := Entity.VALUE-STACK.top();
  var doneVerdict := none;

  if (doneEntity == 'all component') {
    if (Entity != MTC) {
      ***DYNAMIC-ERROR*** // 'all component' is not allowed
    } else if (nonWaitingNr == 1) { // MTC is only active Entity
      Entity.NEXT-CONTROL(true);
      Entity.STATUS := ACTIVE; // DONE is successful
      Entity.VALUE-STACK.push(error);
    } else {
      Entity.NEXT-CONTROL(false);
    }
  } else if (doneEntity == 'any component') {
    if (Entity != MTC) {
      ***DYNAMIC-ERROR*** // 'any component' not allowed
    } else if (doneNr > 0) {
      Entity.NEXT-CONTROL(true);
      Entity.STATUS := ACTIVE; // DONE is successful
      Entity.VALUE-STACK.push(error);
    } else {
      Entity.NEXT-CONTROL(false);
    }
  } else if (Entity.SNAP-DONE.member((doneEntity,-))) {
    Entity.NEXT-CONTROL(true);
    Entity.STATUS := ACTIVE; // DONE is successful
doneVerdict := Entity.SNAP-DONE.random((doneEntity,-)).FIN-VERDICT;
    Entity.VALUE-STACK.push(doneVerdict);
  } else {
    Entity.NEXT-CONTROL(false);
  }
} // end of local scope
RETURN;

// optional verdict
// assignment

* (1)
<done-assignment>

clean-value-stack

Entity.VALUE-STACK.pop(); // removal of doneVerdict
Entity.VALUE-STACK.pop(); // removal of expression
Entity.NEXT-CONTROL(true);
RETURN;

Figure 66: Flow graph segment <done-op>
9.16.1 Flow graph segment <done-assignment>

The flow graph segment <receive-assignment> in figure 66a defines the retrieval of information from received messages and their assignment to variables.

```plaintext
done-assignment
// assuming that assignmentPart denotes a variable
Entity.VALUE-STACK.top()
```

Figure 66a: Flow graph segment <receive-assignment>

9.17 Execute statement

9.17.0 General

The syntactical structure of the execute statement is:

```
execute(<testCaseId>([<act-par 1>, … , <act-par n>]) [, <float-expression>])
```

The execute statement describes the execution of a test case <testCaseId> with the (optional) actual parameters <act-par 1>, … , <act-par n>. Optionally the execute statement may be guarded by a duration provided in form of an expression that evaluates to a float. If within the specified duration the test case does not return a verdict, a timeout exception occurs, the test case is stopped and an error verdict is returned.

NOTE: The operational semantics models the stopping of the test case by a stop of the MTC. In reality, other mechanisms may be more appropriate.

If no timeout exception occurs, the MTC is created, the control instance (representing the control part of the TTCN-3 module) is blocked until the test case terminates, and for the further test case execution the flow of control is given to the MTC. The flow of control is given back to the control instance when the MTC terminates.

The flow graph segment <execute-stmt> in figure 67 defines the execution of an execute statement.

```plaintext
<execute-without-timeout>
OR
<execute-timeout>
```

Figure 67: Flow graph segment <execute-stmt>
9.17.1 Flow graph segment <execute-without-timeout>

The execution of a test case starts with the creation of the mtc. Then the mtc is started with the behaviour defined in the test case definition. Afterwards, the module control waits until the test case terminates. The creation and the start of the MTC can be described by using `create` and `start` statements:

```plaintext
var mtcType MyMTC := mtcType.create;
MyMTC.start(TestCaseName(P1…Pn));
```

The flow graph segment `<execute-without-timeout>` in figure 68 defines the execution of an `execute` statement without the occurrence of a timeout exception by using the flow graph segments of the operations `create` and the `start`.

![Figure 68: Flow graph segment <execute-without-timeout>](image)

\[ <execute-without-timeout> \]

++\[ <create-op> \]

\[ // Creation of the MTC \]

\[ \text{MTC} := \text{Entity.VALUE-STACK.top}(); \]
\[ \text{TC-VERDICT} := \text{none}; \]
\[ \text{DONE} := \text{NULL}; \]
\[ \text{Entity.NEXT-CONTROL}(\text{true}); \]
\[ \text{RETURN}; \]

++\[ <start-component-op> \]

\[ // Start of MTC \]

\[ \text{Entity.STATUS} := \text{BLOCKED}; \]
\[ // MTC will set status to ACTIVE \]
\[ // before it terminates \]
\[ \text{Entity.NEXT-CONTROL}(\text{true}); \]
\[ \text{RETURN}; \]
9.17.2 Flow graph segment <execute-timeout>

The flow graph segment <execute-timeout> in figure 69 defines the execution of an `execute` statement that is guarded by a timeout value. The flow graph segment also models the creation and start of the MTC by a `create` and a `start` operation. In addition, `TIMER-GUARD` guards the termination.

```
segment <execute-timeout>

// The Expression shall evaluate to a float value. This value defines the duration of TIMER-GUARD
<expression>

set-timer-guard
Entity.TIMER-GUARD.STATUS := IDLE;
Entity.TIMER-GUARD.ACT-DURATION :=
Entity.VALUE-STACK.top();
Entity.VALUE-STACK.pop();
Entity.NEXT-CONTROL(true);
RETURN;

<create-op>
// Creation of the MTC
MTC := Entity.VALUE-STACK.top();
TC-VERDICT := none;
DONE := NULL;
Entity.NEXT-CONTROL(true);
RETURN;

<start-component-op>
// Start of MTC
Entity.STATUS := SNAPSHOT;
// MTC will set status to ACTIVE before it terminates
Entity.TIMER-GUARD.STATUS := RUNNING;
Entity.NEXT-CONTROL(true);
RETURN;

prepare-wait
Entity.STATUS := SNAPSHOT;
// MTC will set status to ACTIVE before it terminates
Entity.TIMER-GUARD.STATUS := RUNNING;
Entity.NEXT-CONTROL(true);
RETURN;

if (Entity.STATUS == SNAPSHOT and Entity.TIMER-GUARD.STATUS != TIMEOUT) {
  // Control waits
  Entity.NEXT-CONTROL(true);
}
else {
  // Test case terminated or
  // timer guard timed out
  Entity.NEXT-CONTROL(true);
}
RETURN;

stop-or-timeout
true  false
if (Entity.STATUS != SNAPSHOT) {
  // normal termination
  Entity.TIMER-GUARD.STATUS := IDLE;
  Entity.NEXT-CONTROL(true);
}
else { // guarding timer timed out
  Entity.NEXT-CONTROL(false);
}
RETURN;
```

Figure 69: Flow graph segment <execute-timeout>
9.17.3 Flow graph segment <dynamic-error>

In case of a dynamic error the flow graph segment <dynamic-error> is invoked by the test system. In addition, the flow graph segment <dynamic-error> is also used for describing the behaviour of the test case stop operation (clause 9.53a). All resources allocated to the test case are cleared and the error verdict is assigned to the test case. Control is given to the statement in the control part following the execute statement in which the error occurred.

The flow graph segment <dynamic-error> is invoked by the module control in case that a test case does not terminate within the specified time limit (clause 9.17.2).

```plaintext
// Reset of configuration state
ALL-ENTITY-STATES := NULL;
ALL-PORT-STATES := NULL;
MTC := NULL;
TC-VERDICT := error;
DONE := NULL;
KILLED := NULL;

// Update of the entity state of module control
Control.TIMER-GUARD.STATUS := IDLE;
Control.STATUS := ACTIVE;

// Push error verdict (result of test case execution) onto
// the stack of module control
Control.VALUE-STACK.push(error);

Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 69a: Flow graph segment <dynamic-error>

9.18 Expression

9.18.0 General

For the handling of expressions, the following four cases have to be distinguished:

a) the expression is a literal value (or a constant);

b) the expression is a variable;

c) the expression is an operator applied to one or more operands;

d) the expression is a function or operation call.

NOTE: The cases b), c) and d) may require lazy or fuzzy evaluation. This operational semantics does not model lazy and fuzzy evaluation. It assumes that the correct evaluation is done implicitly.

The syntactical structure of an expression is:

```plaintext
<lit-val> | <var-val> | <func-op-call> | <operand-appl>
```
where:

- `<lit-val>` denotes a literal value;
- `<var-val>` denotes a variable value;
- `<func-op-call>` denotes a function or operation call;
- `<operator-appl>` denotes the application of arithmetic operators like +, -, not, etc.

The execution of an expression is defined by the flow graph segment `<expression>` shown in figure 70.

![Figure 70: Flow graph segment <expression>](image)

### 9.18.1 Flow graph segment `<lit-value>`

The flow graph segment `<lit-value>` in figure 71 pushes a literal value onto the value stack of an entity.

![Figure 71: Flow graph segment <lit-value>](image)

### 9.18.2 Flow graph segment `<var-value>`

The flow graph segment `<var-value>` in figure 72 pushes the value of a variable onto the value stack of an entity.

![Figure 72: Flow graph segment <var-value>](image)
9.18.3 Flow graph segment <func-op-call>

The flow graph segment `<func-op-call>` in figure 73 refers to calls of functions and operations, which return a value that is pushed onto the value stack of an entity. All these calls are considered to be expressions.

![Flow graph segment `<func-op-call>`](image1)

Figure 73: Flow graph segment `<func-op-call>`

9.18.4 Flow graph segment <operator-appl>

The flow graph representation in figure 74 directly refers to the assumption that reverse polish notation is used to evaluate operator expressions. The operands of the operator are calculated and pushed onto the evaluation stack. For the application of the operator, the operands are popped from the evaluation stack and the operator is applied. The result of the operator application is finally pushed onto the evaluation stack. Both, the popping of operands and the pushing the result are considered to be part of the operator application (`Entity.APPLY-OPERATOR(operator)` statement in figure 74), i.e. are not modelled by the operational semantics.

![Flow graph segment `<operator-appl>`](image2)

Figure 74: Flow graph segment `<operator-appl>`
9.19 Flow graph segment <finalize-component-init>

The flow graph segment <finalize-component-init> is part of the flow graph representing the behaviour of a component type definition. Its execution is defined in figure 75.

```
// The status of the father entity is restored. The identifier of the 'father'
// entity is deleted from the VALUE-STACK.
Entity.VALUE-STACK.top().STATUS := Entity.VALUE-STACK.top().VALUE-STACK.top();
Entity.VALUE-STACK.top().VALUE-STACK.pop();
Entity.VALUE-STACK.pop();

// A mark is pushed on the value stack, the entity goes into a blocking state,
// i.e., waits for being started) and control is given back to the module
// evaluation procedure
Entity.VALUE-STACK.push(MARK);
Entity.STATUS := BLOCKED;
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 75: Flow graph segment <finalize-component-init>

9.20 Flow graph segment <init-component-scope>

The flow graph segment <init-component-scope> is part of the flow graph representing the behaviour of a component type definition. Its execution is defined in figure 76.

```
// New scopes for variables, timers
// and ports are created
Entity.INIT-VAR-SCOPE();
Entity.INIT-TIMER-SCOPE();
Entity.INIT-PORT-SCOPE();

Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 76: Flow graph segment <init-component-scope>
9.20a Flow graph segment <init-scope-with-runs-on>

The flow graph segment <init-scope-with-runs-on> is part of the flow graph representing the behaviour of function and altstep definitions. It creates new scopes for variables, timers and ports, which include the names and values declared in the component type definition referred to in the runs on-clause. The execution of the flow graph segment is defined in figure 76a.

```
let { // local scope
  var actVarScope := copy(Entity.DATA-STATE.first());
  var actTimerScope := copy(Entity.TIMER-STATE.first());
  var actPORTScope := copy(Entity.PORT-REF.first());
  Entity.INIT-VAR-SCOPE();
  Entity.DATA-STATE.first().add(actVarScope);
  Entity.INIT-TIMER-SCOPE();
  Entity.DATA-TIMER.first().add(actTimerScope);
  Entity.INIT-PORT-SCOPE();
  Entity.PORT-REF.first().add(actPORTScope)
  Entity.VALUE-STACK.push(MARK);
}
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 76a: Flow graph segment <init-scope-with-runs-on>

9.20b Flow graph segment <init-scope-without-runs-on>

The flow graph segment <init-scope-without-runs-on> is part of the flow graph representing the behaviour of function and altstep definitions. It creates new empty scopes for variables, timers and ports. Functions and altsteps without runs on-clause do not know the names and values declared in the component type definition of the invoking component. The execution of the flow graph segment is defined in figure 76b.

```
Entity.INIT-VAR-SCOPE();
Entity.INIT-TIMER-SCOPE();
Entity.INIT-PORT-SCOPE();
Entity.VALUE-STACK.push(MARK);
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 76b: Flow graph segment <init-scope-without-runs-on>
9.21 Flow graph segment <parameter-handling>

The flow graph-segment \textit{<parameter-handling>} is used in the beginning of flow graphs representing test cases, altsteps and functions. It initializes a new scope and creates variables and timers for the handling of parameters. The flow graph-segment \textit{<parameter-handling>} assumes that the call record of the called test case, altstep or function is the top of the value stack.

\textbf{NOTE:} Parameters may be declared to be lazy or fuzzy. This operational semantics does not model lazy and fuzzy evaluation. It assumes that the correct evaluation of such parameters is done internally.

The execution of flow graph-segment \textit{<parameter-handling>} is shown in figure 77.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure77.png}
\caption{Flow graph segment \textit{<parameter-handling>}}
\end{figure}

9.22 Flow graph segment <statement-block>

The syntactical structure of a statement block is:

\begin{verbatim}
 { <statement_1>; ... ; <statement_n> }
\end{verbatim}

A statement block is a scope unit. When entering a scope unit, new scopes for variables, timers and the value stack have to be initialized. When leaving a scope unit, all variables, timers and stack values of this scope have to be destroyed.

\textbf{NOTE 1:} A Statement block can be embedded in another statement blocks or can occur as body of functions, altsteps, test cases and module control, and within compound statements, e.g. \texttt{alt if-else} or \texttt{do-while}.

\textbf{NOTE 2:} Receiving operations and altstep calls cannot appear in statement blocks, they are embedded in \texttt{alt} statements or \texttt{call} operations.

\textbf{NOTE 3:} The operational semantics also handles operations and declarations like statements, i.e. they are allowed in statement blocks.

\textbf{NOTE 4:} Some TTCN-3 functions, like e.g. \texttt{system} or \texttt{self}, are considered to be expressions, which are not useful as stand-alone statements in statement blocks. Their flow graph representations are not listed in figure 78.

The flow graph segment \textit{<statement-block>} in figure 78 defines the execution of a statement block.
9.23 For statement

The syntactical structure of the for-statement is:

```
for (assignment|variable-declaration, boolean_expression, assignment) statement-block
```

The initialization of the index variable and the corresponding manipulation of the index variable are considered to be assignments to the index variable. It is also allowed to declare and initialize the index variable directly in the for statement. The boolean-expression describes the termination criterion of the loop specified by the for-statement and the statement-block describes the loop body.
The execution of the `for` statement is defined by the flow graph segment `<for-stmt>` shown in figure 79. The initial `<assignment>` or alternative variable declaration with assignment `<var-declaration-init>` (see clause 9.57.1) describes the initialization of the index variable. The `<assignment>` in the `true` branch of the decision node describes the manipulation of the index variable. The `for` statement is a scope unit for a newly declared index variable, this is modelled by means of the nodes `enter-var-scope` and `exit-var-scope.`

```
// The index variable is only
// initialised (<assignment>)
// or declared and initialised
// (<var-declaration-init>)
```

9.24 Function call

9.24.0 General

The syntactical structure of a function call is:

```
<function-name>({<act-par-descr1>, …. , <act-par-descrN>})
```

The `<function-name>` denotes the name of a function and `<act-par-descr1>, …. , <act-par-descrN>` describe the description of the actual parameter values of the function call.
NOTE 1: A function call and an altstep call are handled in the same manner. Therefore, the altstep call (see clause 9.4) refers to this clause.

It is assumed that for each `<act-par-desc>` the corresponding formal parameter identifier `<f-par-Id>` is known, i.e. the syntactical structure above can be extended to:

```plaintext
<function-name>({(<f-par-Id>,<act-par-desc>), ..., (<f-par-Id>,<act-par-desc>)}
```

The flow graph segment `<function-call>` in figure 80 defines the execution of a function call. The execution is structured into three steps. In the first step a call record for the function `<function-name>` is created. In the second step the values of the actual parameter are calculated and assigned to the corresponding field in the call record. In the third step, two situations have to be distinguished: the called function is a user-defined function (`<user-def-func-call>`), i.e. there exists a flow graph representation for the function, or the called function is a pre-defined or external function (`<predef-ext-func-call>`). In case of a user-defined function call, the control is given to the called function. In case of a pre-defined or external function, it is assumed that the call record can be used to execute the function in one step. The correct handling of reference parameters and return value (has to be pushed onto the value stack) is in the responsibility of the called function, i.e. is outside the scope of this operational semantics.

NOTE 2: If the function call models an altstep call, only the `<user-def-func-call>` branch will be chosen, because there exists a flow graph representation of the called altstep.

NOTE 3: The `<function-call>` segment is also used to describe the start of the MTC in an `execute` statement. In this case, a call record for the test case is constructed and only the `<user-def-func-call>` branch will be chosen.

---

**Figure 80: Flow graph segment `<function-call>`**

```plaintext
// For each pair `{(<f-par-Id>, <act-parameter-desc>)} the value of <act-parameter-desc> is calculated and assigned to the corresponding field <f-par-Id> in the call record. The call record is assumed to be the top element in the value stack.

// Retrieves the locations for variables and timers used as reference parameters and declared names of port parameters

// The called function may either be an external or predefined function, or a user-defined function.
```
9.24.1 Flow graph segment <value-par-calculation>

The flow graph-segment <value-par-calculation> is used to calculate actual parameter values and to assign them to the corresponding fields in call records for functions, altsteps and test cases.

It is assumed that a call record is the top element of the value stack and that a pair of:

\[(\text{<f-par-Id_i>}, \text{<act-parameter-desc_i>})\]

has to be handled. \(\text{<act-parameter-desc_i>}\) that has to be evaluated and \(\text{<f-par-Id_i>}\) is the identifier of a formal parameter that has a corresponding field in the call record in the value stack.

The execution of flow graph-segment <value-par-calculation> is shown in figure 81.

```plaintext
let () // scope unit for parVal
  var parVal = Entity.VALUE-STACK.top();
  // parVal is a local variable that stores the value of the expression
  // Removal of expression value.
  // Afterwards the call record is again top of the value stack
  Entity.VALUE-STACK.pop();
  Entity.VALUE-STACK.top().f-par-Id := parVal;
  // Value assignment to call record
) // end of scope for parVal
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 81: Flow graph segment <value-par-calculation>

9.24.2 Flow graph segment <ref-par-var-calc>

The flow graph-segment <ref-par-var-calc> is used to retrieve the locations of variables used as actual reference parameters and to assign them to the corresponding fields in call records for functions, altsteps and test cases.

It is assumed that a call record is the top element of the value stack and that a pair of:

\[(\text{<f-par-Id_i>}, \text{<act-par_i>})\]

has to be handled. \(\text{<act-par_i>}\) is the actual parameter for which the location has to be retrieved and \(\text{<f-par-Id_i>}\) is the identifier of a formal parameter that has a corresponding field in the call record in the value stack.

The execution of flow graph-segment <ref-par-var-calc> is shown in figure 82.
9.24.3 Flow graph segment <ref-par-timer-calc>

The flow graph-segment <ref-par-timer-calc> is used to retrieve the locations of timers used as actual reference parameters and to assign them to the corresponding fields in call records for functions, altsteps and test cases.

It is assumed that a call record is the top element of the value stack and that a pair of:

\( (<f-par-Id_i>, <act-par_i>) \)

has to be handled. \( <act-par_i> \) is the actual parameter for which the location has to be retrieved and \( <f-par-Id_i> \) is the identifier of a formal parameter that has a corresponding field in the call record in the value stack.

The execution of flow graph-segment <ref-par-timer-calc> is shown in figure 83.

9.24.3a Flow graph segment <ref-par-port-calc>

The flow graph-segment <ref-par-port-calc> is used to retrieve the names of ports used as in the component type definitions for the declaration of the port and to assign them to the corresponding fields in call records for functions and altsteps.
It is assumed that a call record is the top element of the value stack and that a pair of:

\[(f\text{-par}_i, \text{act-par}_i)\]

has to be handled. \(\text{act-par}_i\) is the actual parameter for which the location has to be retrieved and \(f\text{-par}_i\) is the identifier of a formal parameter that has a corresponding field in the call record in the value stack.

The execution of flow graph-segment \(<\text{ref-par-timer-calc}>\) is shown in figure 83a.

```
// Value assignment to call record
Entity.VALUE-STACK.top().f-par-Id := Entity.act-par.COMP-PORT-NAME;
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 83a: Flow graph segment \(<\text{ref-par-port-calc}>\)

### 9.24.4 Flow graph segment \(<\text{user-def-func-call}>\)

The flow graph-segment \(<\text{user-def-func-call}>\) (figure 84) describes the transfer of control to a called user-defined function.

```
// Storage of return address
Entity.NEXT-CONTROL(true);
// Control is transferred to called function
Entity.CONTROL-STACK.push(GET-FLOW-GRAPH(function-name));
RETURN;
```

Figure 84: Flow graph segment \(<\text{user-def-func-call}>\)
9.24.5 Flow graph segment <predef-ext-func-call>

The flow graph-segment <predef-ext-func-call> (figure 85) describes the call of a pre-defined or external function.

```plaintext
let { // scope for argument variable
  var argument := Entity.VALUE-STACK.top();
  Entity.VALUE-STACK.pop(); // removal of call record
  // Application of function-name
  function-name(argument);
} // end of scope for argument
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 85: Flow graph segment <predef-ext-func-call>

9.25 Getcall operation

The syntactical structure of the getcall operation is:

```
<portId>.getcall (<matchingSpec>) [from <component_expression>] -\> [<assignmentPart>]
```

Apart from the getcall keyword this syntactical structure is identical to the syntactical structure of the receive operation. Therefore, the operational semantics handles the getcall operation in the same manner as the receive operation. This is also shown in the flow graph segment <getcall-op> (see figure 86), which defines the execution of a getcall operation. The figure refers to flow graph segments related to the receive operation (see clause 9.37).

```plaintext
segment <getcall-op>

<receive-with-sender>
OR
<receive-without-sender>

// Distinction due to the optional
// from-clause
```

Figure 86: Flow graph segment <getcall-op>

9.26 Getreply operation

The syntactical structure of the getreply operation is:

```
<portId>.getreply (<matchingSpec>) [from <component_expression>] -\> <assignmentPart>
```

Apart from the getreply keyword this syntactical structure is identical to the syntactical structure of the receive operation. Therefore, the operational semantics handles the getreply operation in the same manner as the receive operation. This is also shown in the flow graph segment <getreply-op> (see figure 87), which defines the execution of a getreply operation. The figure refers to flow graph segments related to the receive operation (see clause 9.37).
9.27 Getverdict operation

The syntactical structure of the `getverdict` operation is:

```
getverdict
```

The flow graph segment `<getverdict-op>` in figure 88 defines the execution of the `getverdict` operation.

```
segment <getverdict-op>
    getverdict-op
    // E-VERDICT is pushed onto VALUE-STACK
    Entity.VALUE-STACK.push(Entity.E-VERDICT);
    Entity.NEXT-CONTROL(true);
    RETURN;

Figure 88: Flow graph segment `<getverdict-op>`
```

9.28 Goto statement

The syntactical structure of the `goto` statement is:

```
goto <labelId>
```

The flow graph segment `<goto-stmt>` in figure 89 defines the execution of the `goto` statement.

```
segment <goto-stmt>
    nop
    // 'nop' means 'no operation'
    Entity.NEXT-CONTROL(true);
    RETURN;

Figure 89: Flow graph segment `<goto-stmt>`
```

NOTE: The `<labelId>` parameter of the `goto` statement indicates the transfer of control to the place at which a label `<labelId>` is defined (see also clause 9.30).
9.28a Halt port operation

The syntactical structure of the `halt` port operation is:

```plaintext
<portId>.halt
```

The flow graph segment `<halt-port-op>` in figure 89a defines the execution of the `halt` port operation.

```plaintext
let { // Begin of local scope
  var portRef := NULL
  var portState := NULL;
  if (portId == "all port") {
    portState := ALL-PORT-STATES.first();
    while (portState != NULL) {
      if (portState.OWNER == Entity) {
        portState.STATUS := HALTED;
        portState.enqueue(HALT-MARKER);
      }
      portState := ALL-PORT-STATES.next(portState);
    }
  } else {
    portRef := Entity.portId.COMP-PORT-NAME;
    GET-PORT(Entity, portRef).STATUS := HALTED;
    GET-PORT(Entity, portRef).enqueue(HALT-MARKER);
  }
} // End of scope

Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 89a: Flow graph segment `<halt-port-op>`

NOTE: The `HALT-MARKER` that is put by a `halt` operation into the port queue is removed by the `SNAP-PORTS` function (see clause 8.3.3.2) when the marker is reached, i.e. all messages preceding the marker have been processed. The `SNAP-PORTS` function is called when taking a snapshot.

9.29 If-else statement

The syntactical structure of the `if-else` statement is:

```plaintext
if (<boolean-expression>) <statement-block1>
[else <statement-block2>]
```

The else part of the `if-else` statement is optional.

The flow graph segment `<if-else-stmt>` in figure 90 defines the execution of the `if-else` statement.
9.29a Kill component operation

9.29a.0 General

The syntactical structure of the kill component statement is:

```component-expression`.kill```

The kill component operation stops the specified component and removes it from the test system. All test components will be stopped and removed from the test system, i.e. the test case terminates, if the MTC is killed (e.g. `mtc.kill`) or kills itself (e.g. `self.kill`). The MTC may kill all parallel test components by using the all keyword, i.e. `all component.kill`.

A component to be killed is identified by a component reference provided as expression, e.g. a value or value returning function. For simplicity, the keyword "all component" is considered to be special values of `<component-expression>`. The operations `mtc` and `self` are evaluated according to clauses 9.33 and 9.43.
The flow graph segment `<kill-component-op>` in figure 90a defines the execution of the `kill` component operation.

```plaintext
if (Entity.VALUE-STACK.top() == 'all component') {
    Entity.VALUE-STACK.pop(); // clean value stack
    if (Entity != MTC) {
        ***DYNAMIC-ERROR*** // 'all' not allowed
    } else {
        Entity.NEXT-CONTROL(true);
    }
} else {
    Entity.NEXT-CONTROL(false);
}
RETURN;
```

```plaintext
if (Entity.VALUE-STACK.top() == MTC) {
    Entity.VALUE-STACK.pop(); // clean value stack
    Entity.NEXT-CONTROL(true);
} else {
    Entity.NEXT-CONTROL(false);
}
RETURN;
```

```plaintext
if (ALL-ENTITY-STATES.member(Entity.VALUE-STACK.top())) {
    Entity.NEXT-CONTROL(true);
} else {
    if (KILLED.member(Entity.VALUE-STACK.top())) {
        // NULL operation, component already terminated
        Entity.VALUE-STACK.pop(); // clean value stack
        Entity.NEXT-CONTROL(false);
    } else {
        // component id has not been allocated
        ***DYNAMIC-ERROR***
    }
}
RETURN;
```

---

**Figure 90a: Flow graph segment `<kill-component-op>`**
9.29a.1 Flow graph segment <kill-mtc>

The <kill-mtc> flow graph segment in figure 90b describes the killing of the MTC. The effect is that the test case terminates, i.e. the final verdict is calculated and pushed onto the value stack of module control, all resources are released, the KILLED and DONE lists of the module state are emptied and all test components including the MTC are removed from the test system.

```plaintext
let { // local scope for variables
    var myEntity := ALL-ENTITY-STATES.first();
    // Update test case verdict and deletion of components
    while (myEntity != NULL) {
        if (myEntity.E-VERDICT == fail or TC-VERDICT == fail) {
            TC-VERDICT := fail;
        } else {
            if (myEntity.E-VERDICT == inconc or TC-VERDICT == inconc) {
                TC-VERDICT := inconc;
            } else {
                if (myEntity.E-VERDICT == pass or TC-VERDICT == pass) {
                    TC-VERDICT := pass;
                }
            }
            myEntity := ALL-ENTITY-STATES.next(myEntity);
        }
        // TC-VERDICT is the result of the execute operation
        CONTROL.VALUE-STACK.push(TC-VERDICT);
        // Update of test case reference parameters
        UPDATE-REMOTE-LOCATIONS(MTC, CONTROL);
        // Deletion of test components, release of resources, clearing lists
        ALL-ENTITY-STATES := NULL; // Deletion of Entity states
        ALL-PORT-STATES := NULL;
        DONE := NULL;
        KILLED := NULL;
        TC-VERDICT := none;
        MTC := NULL; // Deletion of the last reference to the MTC
        CONTROL.STATUS := ACTIVE; // Control continues
    } // End of local scope
    RETURN;
}
```

Figure 90b: Flow graph segment <kill-mtc-op>
9.29a.2 Flow graph segment <kill-component>

The <kill-component> flow graph segment in figure 90c describes the stopping of a parallel test component (i.e. not the MTC or module control) and its removal from the test system. The effect is that the test case verdict TC-VERDICT and the lists of stopped and killed test components (DONE, and KILLED) are updated and that the component is deleted from the module state. The <kill-component> flow graph assumes that the identifier of the component to be stopped is on top of the value stack of the component that executes the segment.

```plaintext
let { // local scope for variable myEntity
    var myEntity := Entity.VALUE-STACK.top();

    // for test continuation, if kill is executed by another component
    if (Entity != myEntity) {
        Entity.VALUE-STACK.pop(); // clean value stack
        Entity.NEXT-CONTROL(true);
    }

    // Update test case verdict
    if (myEntity.E-VERDICT == fail or TC-VERDICT == fail) {
        TC-VERDICT := fail;
    } else {
        if (myEntity.E-VERDICT == inconc or TC-VERDICT == inconc) {
            TC-VERDICT := inconc;
        } else {
            if (myEntity.E-VERDICT == pass or TC-VERDICT == pass) {
                TC-VERDICT := pass;
            }
        }
    }

    // Deletion of test component
    DONE.append((myEntity, E-VERDICT)); // Update of DONE
    KILLED.append((myEntity, E-VERDICT)); // Update of KILLED
    DEL-ENTITY(myEntity); // Deletion of entity
}

} // End of local scope
RETURN;
```

Figure 90c: Flow graph segment <kill-component>
9.29a.3 Flow graph segment <kill-all-comp>

The <kill-all-comp> flow graph segment in figure 90d describes the termination of all parallel test components of a test case.

![Flow graph segment <kill-all-comp>]

```
let ( // local scope for variable myEntity
  var myEntity := ALL-ENTITY-STATES.next(MTC);

  // Update test case verdict
  while (myEntity != NULL) {
    if (myEntity.E-VERDICT == fail or TC-VERDICT == fail) {
      TC-VERDICT := fail;
    } else {
      if (myEntity.E-VERDICT == inconc or TC-VERDICT == inconc) {
        TC-VERDICT := inconc;
      } else {
        if (myEntity.E-VERDICT == pass or TC-VERDICT == pass) {
          TC-VERDICT := pass;
        }
      }
    }
    myEntity := ALL-ENTITY-STATES.next(myEntity);
  }

  // Deletion of test components
  myEntity := ALL-ENTITY-STATES.next(MTC);
  while (myEntity != NULL) {
    DONE.append(myEntity, TC-VERDICT);  // Update of DONE
    KILLED.append(myEntity, TC-VERDICT);  // Update of KILLED
    DEL-ENTITY(myEntity);  // Deletion of entity
    myEntity := ALL-ENTITY-STATES.next(MTC);  // Next component to delete
  }
} // End of local scope

RETURN;
```

Figure 90d: Flow graph segment <stop-all-comp>

9.29b Kill execution statement

9.29b.0 General

The syntactical structure of the `kill` execution statement is:

```
kill
```

The effect of the `kill` execution statement depends on the entity that executes the `kill` execution statement:

- a) If `kill` is performed by the module control, the test campaign ends, i.e. all test components and the module control disappear from the module state.

- b) If the `kill` is executed by the MTC, all parallel test components and the MTC stop execution. The global test case verdict is updated and pushed onto the value stack of the module control. Finally, control is given back to the module control and the MTC terminates.
c) If the kill is executed by a test component, the global test case verdict TC-VERDICT and the global DONE and KILLED lists are updated. Then the component disappears from the module.

The flow graph segment <kill-exec-stmt> in figure 90e describes the execution of the kill statement.

Figure 90e: Flow graph segment <kill-exec-stmt>

9.29b.1 Flow graph segment <kill-control>

The <kill-control> flow graph segment in figure 90f describes the stopping of module control. The effect is that CONTROL is set to NULL, i.e. the termination condition of the module evaluation procedure (see clause 8.6) is fulfilled.

Figure 90f: Flow graph segment <kill-control>
9.29c Killed component operation

The syntactical structure of the killed component operation is:

\[<\text{component-expression}>.killed \rightarrow <\text{assignmentPart}>]\]

The killed component operation checks whether a component is alive or has been removed from the test system. Depending on whether a checked component is alive or has been removed from the test system, the killed operation decides how the flow of control continues. Using a component reference identifies the component to be checked. The reference may be stored in a variable or be returned by a function, i.e. it is an expression. For simplicity, the keywords "all component" and "any component" are considered to be special expressions.

The optional <assignmentPart> allows the retrieval of the local verdict of the addressed component at the time when the component was killed. The assignment part identifies a variable of type verdicttype to which the retrieved verdict is assigned.

The flow graph segment <killed-op> in figure 90g defines the execution of the killed component operation.
// The Expression shall evaluate to a component reference. The result is pushed onto VALUE-STACK

let { // local scope
  var killedEntity := Entity.VALUE-STACK.top();
  var killedVerdict := none;
  if (killedEntity == 'all component') {
    if (Entity != MTC) {
      ***DYNAMIC-ERROR*** // 'all component' is not allowed
    } else if (Entity.SNAP-ALIVE.lenght() == 1) { // MTC is alive
      Entity.NEXT-CONTROL(true);
      Entity.STATUS := ACTIVE; // KILLED is successful
      Entity.VALUE-STACK.push(error);
    } else {
      Entity.NEXT-CONTROL(false);
    }
  } else if (killedEntity == 'any component') {
    if (Entity != MTC) {
      ***DYNAMIC-ERROR*** // 'any component' not allowed
    } else if (Entity.SNAP-KILLED.length() > 0) {
      Entity.NEXT-CONTROL(true);
      Entity.STATUS := ACTIVE; // KILLED is successful
      Entity.VALUE-STACK.push(error);
    } else {
      Entity.NEXT-CONTROL(false);
    }
  } else if(Entity.SNAP-KILLED.member((killedEntity,-))) { // KILLED is successful
    Entity.NEXT-CONTROL(true);
    Entity.STATUS := ACTIVE; // KILLED is successful
    killedVerdict := Entity.SNAP-killed.random((killedEntity,-)).FIN-VERDICT;
    Entity.VALUE-STACK.push(killedVerdict);
  } else {
    Entity.NEXT-CONTROL(false);
  }
} // end of local scope
RETURN;

* (1)
// optional verdict // assignment

Entity.VALUE-STACK.pop(); // removal of killedVerdict
Entity.VALUE-STACK.pop(); // removal of expression
Entity.NEXT-CONTROL(true);
RETURN;

Figure 90g: Flow graph segment <killed-op>
9.30 Label statement

The syntactical structure of the label statement is:

```
label <labelId>
```

The flow graph segment `<label-stmt>` in figure 91 defines the execution of the label statement.

**NOTE:** The `<labelId>` parameter of the label statement indicates the possibility that a label can be the target for a jump by means of a goto statement (see also clause 9.28).

```
// 'nop' means 'no operation'
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 91: Flow graph segment `<label-stmt>`

9.31 Log statement

The syntactical structure of the log statement is:

```
log (<informal-description>)
```

The flow graph segment `<log-stmt>` in figure 92 defines the execution of the log statement.

**NOTE:** The `<informal description>` parameter of the log statement has no meaning for the operational semantics and is therefore not represented in the flow graph segment.

```
// inscription 'nop' means 'no operation'
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 92: Flow graph segment `<log-stmt>`
9.32 Map operation

The syntactical structure of the map operation is:

\[
\text{map}(<\text{component-expression}>,<\text{portId1}>, \text{system},<\text{portId2}>)
\]

The identifiers <portId1> and <portId2> are considered to be port identifiers of the corresponding test component and test system interface. The component to which the <portId1> belongs is referenced by means of the component reference <component-expression>. The reference may be stored in variables or is returned by a function, i.e. it is an expression, which evaluates to a component reference. The value stack is used for storing the component reference.

NOTE: The map operation does not care whether the system:<portId> statement appears as first or as second parameter. For simplicity, it is assumed that it is always the second parameter.

The execution of the map operation is defined by the flow graph segment <map-op> shown in figure 93.

9.33 Mtc operation

The syntactical structure of the mtc operation is:

\[
\text{mtc}
\]

The flow graph segment <mtc-op> in figure 94 defines the execution of the mtc operation.
9.34 Port declaration

The syntactical structure of a port declaration is:

\[
\text{<portType> <portName>}
\]

Port declarations can be found in component type definitions. The effect of a port declaration is the creation of a new port when a new component of the corresponding type is created. Furthermore, a port reference is created in the actual scope of the test component. In the newly created port reference, the values \text{PORT-NAME} and \text{COMP-PORT-NAME} are equal. The flow graph segment \text{<port-declaration>} in figure 95 defines the execution of a port declaration.

```
segment <port-declaration>

// A new port state and a port reference
// are created
ALL-PORT-STATES.append(NEW-PORT(Entity, portName);
Entity.INIT-PORT(portName, portName);
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 95: Flow graph segment \text{<port-declaration>}

9.35 Raise operation

9.35.0 General

The syntactical structure of the \text{raise} operation is:

\[
\text{<portId>}.raise(<\text{exceptSpec}>, [to <\text{receiver-spec}>])
\]

The optional \text{<receiver-spec>} in the \text{to} clause refers to the receivers of the exception. In case of a one-to-one communication, the \text{<receiver-spec>} addresses a single entity (including the SUT or an entity within the SUT). In case of multicast or broadcast communication, the \text{<receiver-spec>} specifies a set or all test components connected via the specified port with the calling component.

The flow graph segment \text{<raise-op>} in figure 96 defines the execution of a \text{raise} operation.

```
segment <raise-op>

// A raise operation may address one,
// multiple (multicast and broadcast)
// or no receiver entities.
```

Figure 96: Flow graph segment \text{<raise-op>}
9.35.1 Flow graph segment <raise-with-one-receiver-op>

The flow graph segment <raise-with-one-receiver-op> in figure 97 defines the execution of a `raise` operation where the receiver is specified in form of an expression.

```plaintext
segment <raise-with-one-receiver-op>

<expression>

raise-with-one-receiver-op (portId, exceptSpec)

let {
  var receiver := Entity.VALUE-STACK.top();
  var portRef := Entity.portId.COMP-PORT-NAME;
  var remotePort := GET-REMOTE-PORT(Entity, portref, receiver);
  if (remotePort == NULL) {
    ***DYNAMIC-ERROR***; // Remote port cannot be found
  }
  if (remotePort == SYSTEM) {
    // Port is mapped onto a port of the test system
    // reception of the reply by the SUT is outside
    // the scope of the operational semantics
  } else { // sending of exception
    remotePort.enqueue(CONSTRUCT-ITEM(Entity, raise, exceptSpec));
  }
} // end of scope of receiver and remotePort

Entity.VALUE-STACK.pop(); // clean value stack
Entity.NEXT-CONTROL(true);
RETURN;
}
```

Figure 97: Flow graph segment <raise-with-one-receiver-op>

9.35.1a Flow graph segment <raise-with-multiple-receivers-op>

The flow graph segment <raise-with-multiple-receivers-op> in figure 97a defines the execution of a `raise` operation where multiple receivers are addressed. In case of broadcast communication the keyword `all component` is used as receiver specification. In case of multicast communication a list of expressions is provided which shall evaluate to component references or address values.

The component references or address values of the addressed entities (or the keyword `all component`) are pushed onto the value stack of the calling entity. The number of references stored in the value stack is considered to be known, i.e. it is the parameter `number` of the basic flow graph node `raise-with-multiple-receivers-op` in figure 97a. The `number` parameter is 1 in case of broadcast communication, i.e. the keyword `all component` is top element in the value stack.
Figure 97a: Flow graph segment `<raise-with-multiple-receivers-op>`
9.35.2 Flow graph segment \texttt{<raise-without-receiver-op>}

The flow graph segment \texttt{<raise-without-receiver-op>} in figure 98 defines the execution of a raise operation without \texttt{to}-clause.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure98.png}
\caption{Flow graph segment \texttt{<raise-without-receiver-op>}}
\end{figure}

9.36 Read timer operation

The syntactical structure of the \texttt{read} timer operation is:

\begin{verbatim}
<timerId>.read
\end{verbatim}

The flow graph segment \texttt{<read-timer-op>} in figure 99 defines the execution of the \texttt{read} timer operation.

The \texttt{read} timer operation distinguishes between its usage in a Boolean guard of an \texttt{alt} statement or blocking \texttt{call} operation and all other cases. If used in a Boolean guard, the result of the \texttt{read} timer operation is based on the actual snapshot, i.e. the \texttt{SNAP-STATUS} and \texttt{SNAP-VALUE} entries of the timer binding, in all other cases, the \texttt{STATUS}, \texttt{ACT-DURATION} and \texttt{TIME-LEFT} entries of the timer binding determine the result of the operation.
9.37 Receive operation

9.37.0 General

The syntactical structure of the receive operation is:

```
<portId>.receive (<matchingSpec>) [from <component-expression>] [→ <assignmentPart>]
```

The optional <component-expression> in the from clause refers to the sender entity. It may be provided in form of a variable value or the return value of a function, i.e. it is assumed to be an expression. The optional <assignmentPart> denotes the assignment of received information if the received message matches to the matching specification <matchingSpec> and to the (optional) from clause.

The flow graph segment <receive-op> in figure 100 defines the execution of a receive operation.
9.37.1 Flow graph segment <receive-with-sender>

The flow graph segment <receive-with-sender> in figure 101 defines the execution of a receive operation where the sender is specified in form of an expression.
let { // local scope for portRef and sender
    var portRef := NULL;
    var sender := Entity.VALUE-STACK.top(); // Sender
    Entity.VALUE-STACK.pop(); // Clean value stack
    if (portID == "any port") {
        portRef := ALL-PORT-STATES.random(MATCH-ITEM(SNAP-VALUE, matchingSpec, sender)
            & & OWNER == "Entity");
        if (portRef == NULL) { // no 'matching' port found
            Entity.NEXT-CONTROL(false);
            RETURN;
        }
    } else {
        portRef := GET-PORT(Entity, Entity.portId.COMP-PORT-NAME); // Specified port
    }
    // MATCHING
    if (PortRef.first() == NULL) { // Port queue is empty, no match
        Entity.NEXT-CONTROL(false);
        RETURN;
    } else {
        if { MATCH-ITEM(portRef.SNAP-VALUE, matchingSpec, sender)} {
            // The message in the queue matches
            Entity.VALUE-STACK.push(portRef); // Saving port reference
            Entity.STATUS := ACTIVE; // successful match, Entity status is changed
            // from SNAPSHOT to ACTIVE
            Entity.NEXT-CONTROL(true);
        } else { // The top item in the queue does not match
            Entity.NEXT-CONTROL(false);
        }
    }
} // End of scope of portRef and sender

receive-with-sender
(portId, matchingSpec)

true

/*(1)*/

receive-assignment

remove-from-port

false

true

// Removal of received item from port
Entity.VALUE-STACK.top().dequeue();
Entity.VALUE-STACK.pop();
Entity.NEXT-CONTROL(true);
RETURN;

Figure 101: Flow graph segment <receive-with-sender>
9.37.2 Flow graph segment <receive-without-sender>

The flow graph segment <receive-without-sender> in figure 102 defines the execution of a `receive` operation without a `from` clause.

```plaintext
let { // local scope
    var portRef := NULL;
    if (portID == "any port") {
        portRef := ALL-PORT-STATES.random(MATCH-ITEM(SNAP-VALUE, matchingSpec, NONE) & OWNER == Entity);
        if (portRef == NULL) { // no 'matching' port found
            Entity.NEXT-CONTROL(false);
            RETURN;
        }
    } else {
        portRef := GET-PORT(Entity, Entity.portId.COMP-PORT-NAME); // Specified port
    }
    // MATCHING
    if (PortRef.first() == NULL) { // Port queue is empty, no match
        Entity.NEXT-CONTROL(false);
        RETURN;
    } else {
        if (MATCH-ITEM(portRef.SNAP-VALUE, matchingSpec, NONE)) {
            // The message in the queue matches
            Entity.VALUE-STACK.push(portRef); // Saving port reference
            Entity.STATUS := ACTIVE; // successful match, Entity status is changed
            Entity.NEXT-CONTROL(true); // from SNAPSHOT to ACTIVE
        } else { // The first item in the queue does not match
            Entity.NEXT-CONTROL(false);
        }
        RETURN;
    }
} // End of scope
```

Figure 102: Flow graph segment <receive-without-sender>
9.37.3 Flow graph segment <receive-assignment>

The flow graph segment <receive-assignment> in figure 103 defines the retrieval of information from received messages and their assignment to variables.

![Flow graph segment <receive-assignment>](image)

Figure 103: Flow graph segment <receive-assignment>

9.38 Repeat statement

The syntactical structure of the repeat statement is:

```
repeat
```

Basically, the repeat statement is a return statement without return value, which also changes the entity status to REPEAT. The status REPEAT will force the re-evaluation of the alt statement in which the repeat statement has been executed. The flow graph segment <repeat-stmt> shown in figure 104 defines the execution of the repeat statement.

![Flow graph segment <repeat-stmt>](image)

Figure 104: Flow graph segment <repeat-stmt>
9.39 Reply operation

9.39.0 General

The syntactical structure of the \textit{reply} operation is:

\begin{verbatim}
<portId>reply (<replySpec>) [to <receiver-spec>]
\end{verbatim}

The optional \textit{<receiver-spec>} in the \textit{to} clause refers to the receivers of the reply. In case of a one-to one communication, the \textit{<receiver-spec>} addresses a single entity (including the SUT or an entity within the SUT). In case of multicast or broadcast communication, the \textit{<receiver-spec>} specifies a set or all test components or entities in the SUT connected via the specified port with the calling component.

The flow graph segment \textit{<reply-op>} in figure 105 defines the execution of a \textit{reply} operation.

Figure 105: Flow graph segment <reply-op>
9.39.1 Flow graph segment <reply-with-one-receiver-op>

The flow graph segment <reply-with-one-receiver-op> in figure 106 defines the execution of a reply operation where the receiver is specified in form of an expression.

```plaintext
let {
  var receiver := Entity.VALUE-STACK.top();
  var portRef := Entity.portId.COMP-PORT-NAME;
  var remotePort := GET-REMOTE-PORT(Entity, portRef, receiver);
  if (remotePort == NULL) {
    ***DYNAMIC-ERROR***; // Remote port cannot be found
  }
  if (remotePort == SYSTEM) {
    // Port is mapped onto a port of the test system
    // reception of the reply by the SUT is outside
    // the scope of the operational semantics
  }
  else { // sending of reply
    remotePort.enqueue(CONSTRUCT-ITEM(Entity, reply, replySpec));
  }
} // end of scope of receiver and remotePort
Entity.VALUE-STACK.pop(); // clean value stack
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 106: Flow graph segment <reply-with-one-receiver-op>

9.39.1a Flow graph segment <reply-with-multiple-receivers-op>

The flow graph segment <reply-with-multiple-receivers-op> in figure 106a defines the execution of a reply operation where multiple receivers are addressed. In case of broadcast communication the keyword all component is used as receiver specification. In case of multicast communication a list of expressions is provided which shall evaluate to component references or address values.

The component references or address values of the addressed entities (or the keyword all component) are pushed onto the value stack of the calling entity. The number of component references or address values stored in the value stack is considered to be known, i.e. it is the parameter number of the basic flow graph node reply-with-multiple-receivers-op in figure 106a. The number parameter is 1 in case of broadcast communication, i.e. the keyword all component is top element in the value stack.
segment <reply-with-multiple-receivers-op>

let { // loop counter variable
    var i; // variable for connections in port states
    var receiver; // variable for receiver component references or
    // address values
    var localPort, remotePort; // variables for port references
    localPort := Entity.portId.COMP-PORT-NAME; // local port

    if (Entity.VALUE-STACK.top() == all component) {
        connection := localPort.CONNECTIONS-LIST.next(connection);
        while (connection != NULL) {
            remotePort := connection.REMOTE-PORT-NAME;
            if (remotePort == SYSTEM) {
                // Port is mapped onto a port of the test system
                // reception of the reply by the SUT is outside
                // the scope of the operational semantics
            } else { // sending of call
                remotePort.enqueue(CONSTRUCT-ITEM(Entity, reply, replySpec));
            }
            connection := localPort.CONNECTIONS-LIST.next(connection);
        }
    } else {
        for (i == 1; i <= number; i := i+1) {
            receiver := Entity.VALUE-STACK.top();
            Entity.VALUE-STACK.pop(); // clean value stack
            remotePort := GET-REMOTE-PORT(Entity, localPort, receiver);
            if (remotePort == NULL) {
                ***DYNAMIC-ERROR***; // Remote port cannot be found
            } else if (remotePort == SYSTEM) {
                // Port is mapped onto a port of the test system
                // reception of the reply by the SUT is outside
                // the scope of the operational semantics
            } else { // sending of call
                remotePort.enqueue(CONSTRUCT-ITEM(Entity, reply, replySpec));
            }
        }
    }
} // end of local scope

Entity.NEXT-CONTROL(true);
RETURN;

Figure 106a: Flow graph segment <reply-with-multiple-receivers-op>
9.39.2 Flow graph segment <reply-without-receiver-op>

The flow graph segment <reply-without-receiver-op> in figure 107 defines the execution of a reply operation without to-clause.

```plaintext
let {
  var portRef := Entity.portId.COMP-PORT-NAME;
  var remotePort := GET-REMOTE-PORT(Entity, portRef, NONE);
  if (remotePort == NULL) {
    ***DYNAMIC-ERROR***; // Remote port cannot be found
  }
  if (remotePort == SYSTEM) {
    // Port is mapped onto a port of the test system
    // reception of the reply by the SUT is outside
    // the scope of the operational semantics
  }
  else { // sending of reply
    remotePort.enqueue(CONSTRUCT-ITEM(Entity, reply, replySpec));
  }
} // end of scope of remotePort

Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 107: Flow graph segment <reply-without-receiver-op>

9.40 Return statement

9.40.0 General

The syntactical structure of the return statement is:

```plaintext
return [<expression>]
```

The optional `<expression>` describes a possible return value of a function. The execution of a return statement means that the control leaves the actual scope unit, i.e. variables and timers only known in this scope have to be deleted and the value stack has to be updated. A `return` statement has the effect of a `stop` component operation, if it is the last statement in a behaviour description.

NOTE: Test cases and module control will always end with a `stop` component operation. This is due to their flow graph representation (see clause 8.2). Only other test components may terminate with a `return` statement.
The flow graph segment `<return-stmt>` in figure 108 defines the execution of a `return` statement.

![Diagram](Image)

Figure 108: Flow graph segment `<return-stmt>`
9.40.1 Flow graph segment <return-with-value>

The flow graph segment `<return-with-value>` in figure 109 defines the execution of a `return` that returns a value specified in form of an expression.

```
let {
    var return-value := Entity.VALUE-STACK.top();
    Entity.DEL-VAR-SCOPE();
    Entity.DEL-TIMER-SCOPE();
    Entity.DEL-PORT-SCOPE();
    Entity.VALUE-STACK.clear-until(MARK);
    Entity.VALUE-STACK.push(return-value);
} // end of scope of return-value
Entity.CONTROL-STACK.pop();  // return address is lying on the control stack
if (Entity.CONTROL-STACK.top() == NULL) {
    // return is stop or kill
    Entity.VALUE-STACK.push(Entity);
    Entity.NEXT-CONTROL(false);
}
RETURN;
if (Entity.VALUE-STACK.top().KEEP-ALIVE == true) {
    Entity.NEXT-CONTROL(true);
} else {
    Entity.NEXT-CONTROL(false);
}
RETURN;
```

Figure 109: Flow graph segment `<return-with-value>`
9.40.2 Flow graph segment <return-without-value>

The flow graph segment <return-without-value> in figure 110 defines the execution of a return statement that returns no value.

```plaintext
let {
    var return-value := Entity.VALUE-STACK.top();
    Entity.DEL-VAR-SCOPE();
    Entity.DEL-TIMER-SCOPE();
    Entity.DEL-PORT-SCOPE();
    Entity.VALUE-STACK.clear-until(MARK);
} // end of scope of return-value

Entity.CONTROL-STACK.pop(); // return address
    // is lying on the control stack
if (Entity.CONTROL-_STACK.top() == NULL) {
  // return is stop or kill
  Entity.VALUE-STACK.push(Entity);
  Entity.NEXT-CONTROL(false);
} RETURN;

if (Entity.VALUE-STACK.top().KEEP-ALIVE == true)) {
  Entity.NEXT-CONTROL(true);
} else {
  Entity.NEXT-CONTROL(false);
} RETURN;
```

Figure 110: Flow graph segment <return-without-value>
9.41 Running component operation

9.41.0 General

The syntactical structure of the running component operation is:

```
<component-expression>.running
```

The running component operation checks whether a component is running or has either stopped or terminated and been removed from the test system. The component to be checked is identified by a component reference, which may be provided in form of a variable or value returning function, i.e. is an expression. For simplicity, the keywords "all component" and "any component" are considered to be special expressions.

The running component operation distinguishes between its usage in a Boolean guard of an alt statement or blocking call operation and all other cases. If used in a Boolean guard, the result of running component operation is based on the actual snapshot. In all other cases evaluates directly the state information.

The result of the running component operation is pushed onto the value stack of the entity, which called the operation.

The flow graph segment <running-component-op> in figure 111 defines the execution of the running component operation.

```
segment <running-component-op>

<expression>

// The expression shall evaluate
// to a component reference. The
// result is pushed onto VALUE-STACK

decision

if (Entity.STATUS == ACTIVE) {
    Entity.NEXT-CONTROL(true);
} else { // Entity is in a snapshot
    Entity.NEXT-CONTROL(false);
}
RETURN;

<running-comp-act>

<running-comp-snap>
```

Figure 111: Flow graph segment <running-component-op>
9.41.1 Flow graph segment \texttt{<running-comp-act>}

The flow graph segment \texttt{<running-comp-act>} in figure 112 describes the execution of the \texttt{running} component operation outside a snapshot, i.e. the component is in the status \texttt{ACTIVE}.

egin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{flow_graph_segment.png}
\caption{Flow graph segment \texttt{<running-comp-act>}}
\end{figure}
9.41.2 Flow graph segment <running-comp-snap>

The flow graph segment <running-comp-snap> in figure 113 describes the execution of the running component operation during the evaluation of a snapshot, i.e. the component is in the status SNAPSHOT.

```
let { // local scope
  var comp;  // for storing a component reference
  var decision; // Boolean

  if (Entity.VALUE-STACK.top() == 'all component') {
    if (Entity != MTC) {
      ***DYNAMIC-ERROR*** // 'all component' is not allowed
    } else {
      if (Entity.SNAP-DONE.length() == 0) {
        Entity.VALUE-STACK.push(true);
      } else { // at least one component has been stopped
        Entity.VALUE-STACK.push(false);
      }
    }
  } else {
    if (Entity.VALUE-STACK.top() == 'any component') {
      if (Entity != MTC) {
        ***DYNAMIC-ERROR*** // 'any component' not allowed
      } else {
        comp := Entity.SNAP-ALIVE.next(MTC);
        while (comp != NULL and decision == false) {
          if (comp.STATUS == ACTIVE) {
            decision := true;
          }
        }
        comp := ALL-ENTITY-STATES.next(comp);
        Entity.VALUE-STACK.push(decision);
      }
    } else {
      if (Entity.SNAP-ALIVE.member(Entity.VALUE-STACK.top())) {
        // Specified component is alive
        Entity.VALUE-STACK.push(true);
      } else {
        Entity.VALUE-STACK.push(false);
      }
    }
  }

  Entity.NEXT-CONTROL(true);
  RETURN;
}
```

Figure 113: Flow graph segment <running-comp-snap>
9.42 Running timer operation

The syntactical structure of the running timer operation is:

```plaintext
<timerId>.running
```

The flow graph segment `<running-timer-op>` in figure 114 defines the execution of the running timer operation.

The running timer operation distinguishes between its usage in a Boolean guard of an alt statement or blocking call operation and all other cases. If used in a Boolean guard, the result of running timer operation is based on the actual snapshot, i.e. the SNAP-STATUS entry of the timer binding, in all other cases, the STATUS entry of the timer binding determines the result of the operation.

The any keyword is handled as a special value of timerId.

```plaintext
segment <running-timer-op>

let { // local scope for variables myStatus and myTimerList
    var myStatus; // for storing status values of timers
    var myTimerList; // for storing a list of timer Bindings

    if (timerId == "any timer") {
        myTimerList := Entity.TIMER-STATE.first();
        timerId := NULL;
        if (Entity.STATUS == SNAPSHOT) {
            while (myTimerList != NULL && timerId == NULL) {
                timerId := myTimerList.random(SNAP-STATUS == RUNNING);
                myTimerList := Entity.TIMER-STATE.next(myTimerList);
            }
        } else {
            while (myTimerList != NULL && timerId == NULL) {
                timerId := myTimerList.random(STATUS == RUNNING);
                myTimerList := Entity.TIMER-STATE.next(myTimerList);
            }
        }
    }

    if (timerId != NULL) {
        myStatus := Entity.timerId.STATUS;
        if (Entity.STATUS == SNAPSHOT) {
            myStatus := Entity.timerId.SNAP-STATUS;
        }
        if (myStatus == RUNNING) {
            Entity.VALUE-STACK.push(true);
        } else {
            Entity.VALUE-STACK.push(false);
        }
    } else {
        Entity.VALUE-STACK.push(false);
    }
} // end local scope

Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 114: Flow graph segment `<running-timer-op>`
9.43 Self operation

The syntactical structure of the self operation is:

```plaintext
self
```

The flow graph segment `<self-op>` in figure 115 defines the execution of the self operation.

```
Entity.VALUE-STACK.push(Entity);
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 115: Flow graph segment `<self-op>`

9.44 Send operation

9.44.0 General

The syntactical structure of the send operation is:

```plaintext
<portId>.send (<send-spec>) [to <receiver-spec>]
```

The optional `<receiver-spec>` in the to clause refers to the receivers of the message. In case of a one-to one communication, the `<receiver-spec>` addresses a single entity (including the SUT or an entity within the SUT). In case of multicast or broadcast communication, the `<receiver-spec>` specifies a set or all test components or entities in the SUT connected via the specified port with the calling component.

The flow graph segment `<send-op>` in figure 116 defines the execution of a send operation.

```
// A send operation may address one,
// multiple (multicast and broadcast)
// or no receiver entities.
```

Figure 116: Flow graph segment `<send-op>`
9.44.1 Flow graph segment <send-with-one-receiver-op>

The flow graph segment <send-with-one-receiver-op> in figure 117 defines the execution of a send operation where the receiver is specified in form of an expression.

```
let {
  var receiver := Entity.VALUE-STACK.top();
  var portRef := Entity.portId.COMP-PORT-NAME;
  var remotePort := GET-REMOTE-PORT(Entity, portRef, receiver);

  if (remotePort == NULL) {
    ***DYNAMIC-ERROR***; // Remote port cannot be found
  }
  if (remotePort == SYSTEM) {
    // Port is mapped onto a port of the test system
    // reception of the reply by the SUT is outside
    // the scope of the operational semantics
  }
  else { // sending of message
    remotePort.enqueue(CONSTRUCT-ITEM(Entity, send, sendSpec));
  }
} // end of scope of receiver and remotePort
Entity.VALUE-STACK.pop(); // clean value stack
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 117: Flow graph segment <send-with-one-receiver-op>

9.44.1a Flow graph segment <send-with-multiple-receivers-op>

The flow graph segment <send-with-multiple-receivers-op> in figure 117a defines the execution of a send operation where multiple receivers are addressed. In case of broadcast communication the keyword all component is used as receiver specification. In case of multicast communication a list of expressions is provided which shall evaluate to component references or address values.

The component references or address values of the addressed entities (or the keyword all component) are pushed onto the value stack of the calling entity. The number of references stored in the value stack is considered to be known, i.e. it is the parameter number of the basic flow graph node send-with-multiple-receivers-op in figure 117a. The number parameter is 1 in case of broadcast communication, i.e. the keyword all component is top element in the value stack.
<send-with-multiple-receivers-op>

let { // loop counter variable
    var i; // loop counter variable
    var connection; // variable for connections in port states
    var receiver; // variable for receiver component references
    var localPort, remotePort; // variables for port references
    localPort := Entity.portId.COMP-PORT-NAME; // local port
    if (Entity.VALUE-STACK.top() == all component) {
        connection := localPort.CONNECTIONS-LIST.next(connection);
        while (connection != NULL) {
            remotePort := connection.REMOTE-PORT-NAME;
            if (remotePort == SYSTEM) {
                // Port is mapped onto a port of the test system
                // reception of the reply by the SUT is outside
                // the scope of the operational semantics
            } else { // sending of call
                remotePort.enqueue(CONSTRUCT-ITEM(Entity, send, sendSpec));
            }
            connection := localPort.CONNECTIONS-LIST.next(connection);
        }
    } else {
        for (i == 1; i <= number; i := i+1) {
            receiver := Entity.VALUE-STACK.top();
            Entity.VALUE-STACK.pop(); // clean value stack
            remotePort := GET-REMOTE-PORT(Entity, localPort, receiver);
            if (remotePort == NULL) {  // Remote port cannot be found
                ***DYNAMIC-ERROR***;
            } else { // sending of call
                remotePort.enqueue(CONSTRUCT-ITEM(Entity, send, sendSpec));
            }
        }
    }
} // end of local scope

Entity.NEXT-CONTROL(true);
RETURN;

// Each expression shall evaluate
// to a component reference or an
// address value.

Figure 117a: Flow graph segment <send-with-multiple-receivers-op>
9.44.2 Flow graph segment <send-without-receiver-op>

The flow graph segment <send-without-receiver-op> in figure 118 defines the execution of a send operation without to-clause.

```plaintext
segment <send-without-receiver-op>

send-without-receiver-op (portId, sendSpec)

let {
    var portRef := Entity.portId.COMP-PORT-NAME;
    var remotePort := GET-REMOTE-PORT(Entity, portRef, NONE);
    if (remotePort == NULL) {
        ***DYNAMIC-ERROR***; // Remote port cannot be found
    }
    if (remotePort == SYSTEM) {
        // Port is mapped onto a port of the test system
        // reception of the reply by the SUT is outside
        // the scope of the operational semantics
    } else {
        // sending of message
        remotePort.enqueue(CONSTRUCT-ITEM(Entity, send, sendSpec));
    }
} // end of scope of remotePort

Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 118: Flow graph segment <send-without-receiver-op>

9.45 Setverdict operation

The syntactical structure of the setverdict operation is:

```plaintext
setverdict(<verdicttype-expression> [ , <verdict-reason>])
```

The <verdicttype-expression> parameter of the setverdict operation is an expression that shall evaluate to a value of type verdicttype, i.e. none, pass, inconc or fail. The expression is evaluated before the setverdict operation is applied.

The second optional parameter allows specifying a reason for setting a verdict. This reason does not contribute to the test behaviour and is therefore not considered in the operational semantics.

The flow graph segment <setverdict-op> in figure 119 defines the execution of the setverdict operation.
9.46 Start component operation

The syntactical structure of the start component operation is:

<component-expression>.start(<function-name>(<act-par-descr1>, ..., <act-par-descrn>))

The start component operation starts a component. Using a component reference identifies the component to be started. The reference may be stored in a variable or be returned by a function, i.e. it is an expression that evaluates to a component reference.

The <function-name> denotes to the name of the function that defines the behaviour of the new component and <act-par-descr1>, ..., <act-par-descrn> provide the description of the actual parameter values of <function-name>. The descriptions of the actual parameters are provided in form of expressions that have to be evaluated before the call can be executed. The handling of formal and actual value parameters is similar to their handling in function calls (see clause 9.24).

The flow graph segment <start-component-op> in figure 120 defines the execution of the start component operation. The start component operation is executed in four steps. In the first step a call record is created. In the second step the actual parameter values are calculated. In the third step the reference of the component to be started is retrieved, and, in the fourth step, control and call record are given to the new component.

NOTE: The flow graph segment in figure 120 includes the handling of reference parameters (<ref-var-par-calc>). Reference parameters are needed to explain reference parameters of test cases. The operational semantics assumes that these parameters are handled by the MTC.
**Figure 120: Flow graph segment <start-component-op>**

```
segment <start-component-op>

// The expression shall evaluate to a component reference. // It refers to the component to be started
<expression>

construct-call-record (function-name)

// This flow graph segment is also used to explain // the execute statement. Test cases are allowed to have // reference parameters. The operational semantics // assumes that these parameters are owned (and updated) // by the MTC.
<ref-var-par-calc>

// For each pair (<f-par-Idi>, <act-parameter-desci>) the // value of <act-parameter-desci is calculated and // assigned to the corresponding field <f-par-Idi> // in the call record. The call record is assumed to be // the top element in the value stack.
/value-par-calc>

let {
  var toBeStarted := Entity.VALUE-STACK.top();  // toBeStarted is a local variable that stores the // identifier of the component to be started
  Entity.VALUE-STACK.pop(); // Removal of component reference. Afterwards the // call record is on top of the value stack
toBeStarted.VALUE-STACK.push(Entity.VALUE-STACK.top()); // Call record is transferred to toBeStarted.
  Entity.VALUE-STACK.pop(); // Removal of the call record from the value stack // of the starting component (= Entity).
toBeStarted.CONTROL-STACK.push(GET-FLOW-GRAPH(function-name)); // Control stack of toBeStarted is set to // the start node of its behaviour.
  toBeStarted.STATUS := ACTIVE; // Control is given to toBeStarted
  if (DONE.member(toBeStarted)) { // Update DONE list
    DONE.delete(toBeStarted);
  }
} // end of scope for variable toBeStarted
  Entity.NEXT-CONTROL(true);
```
9.47 Start port operation

The syntactical structure of the \texttt{start} port operation is:

\texttt{<portId>.start}

The flow graph segment \texttt{<start-port-op>} in figure 121 defines the execution of the \texttt{start} port operation.

\begin{verbatim}
segment <start-port-op>
start-port-op (portId)
let { // Begin of local scope
  var portRef := NULL
  var portState := NULL;
  if (portId == "all port") {
    portState := ALL-PORT-STATES.first();
    while (portState != NULL) {
      if (portState.OWNER == Entity) {
        portState.VALUE-QUEUE.clear();
        portState.STATUS := STARTED
      }
      portState := ALL-PORT-STATES.next(portState);
    }
  } else {
    portRef := Entity.portId.COMP-PORT-NAME;
    GET-PORT(Entity, portRef).clear();
    GET-PORT(Entity, portRef).STATUS := STARTED;
  } // End of scope
Entity.NEXT-CONTROL(true);
RETURN;
\end{verbatim}

Figure 121: Flow graph segment \texttt{<start-port-op>}

9.48 Start timer operation

9.48.0 General

The syntactical structure of the \texttt{start} timer operation is:

\texttt{<timerId>.start [(<float-expression>)]}

The optional \texttt{<float-expression>} parameter of the timer \texttt{start} operation denotes the actual duration of the timer. If it is not provided, the default duration will be used by the \texttt{start} operation. The expression that shall evaluate to a value of type \texttt{float}. If provided, the expression shall be evaluated before the \texttt{start} operation is applied. The result of the evaluation is pushed onto the \texttt{VALUE-STACK} of \texttt{Entity}.

The flow graph segment \texttt{<start-timer-op>} in figure 122 defines the execution of the \texttt{start} timer operation.
9.48.1 Flow graph segment `<start-timer-op-default>`

The flow graph segment `<start-timer-op-default>` in figure 123 defines the execution of the `start` timer operation with the default value.

```plaintext
// The timer reference <timerId> is copied into the node // attribute 'timerId'
if (Entity.timerId.DEF-DURATION == NONE) {
    ***DYNAMIC-ERROR*** // Timer has no default duration
} else {
    Entity.TIMER-SET(timerId, ACT-DURATION, Entity.timerId.DEF-DURATION);
    Entity.TIMER-SET(timerId, TIME-LEFT, Entity.timerId.DEF-DURATION);
    Entity.TIMER-SET(timerId, STATUS, RUNNING);
}
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 123: Flow graph segment `<start-timer-op-default>`
9.48.2 Flow graph segment <start-timer-op-duration>

The flow graph segment <start-timer-op-duration> in figure 124 defines the execution of the start timer operation with a provided duration.

![Flow graph segment](image)

```plaintext
segment <start-timer-op-duration>

<expression>

// The expression shall evaluate to a float. The result is pushed onto VALUE-STACK.

start-timer-op-duration (timerId)

// The timer reference <timerId> is copied into the node attribute 'timerId'
Entity.TIMER-SET(timerId, ACT-DURATION, Entity.VALUE-STACK.top());
Entity.TIMER-SET(timerId, TIME-LEFT, Entity.VALUE-STACK.top());
Entity.TIMER-SET(timerId, STATUS, RUNNING);

Entity.VALUE-STACK.pop(); // clean VALUE-STACK
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 124: Flow graph segment <start-timer-op-duration>

9.49 Stop component operation

9.49.0 General

The syntactical structure of the stop component statement is:

```
<component-expression>.stop
```

The stop component operation stops the specified component. All test components will be stopped, i.e. the test case terminates, if the MTC is stopped (e.g. mtc.stop) or stops itself (e.g. self.stop). The MTC may stop all parallel test components by using the all keyword, i.e. all component.stop.

Stopped components created with an alive clause in the create operation are not removed from the test system. They can be restarted by using a start statement. Variables, ports, constants and timers owned by such a component, i.e. declared and defined in the corresponding component type definition, keep their status. A stop operation for a component created without an alive clause is semantically equivalent to a kill operation. The component is removed from the test system.

A component to be stopped is identified by a component reference provided as expression, e.g. a value or value returning function. For simplicity, the keyword "all component" is considered to be special values of <component-expression>. The operations mtc and self are evaluated according to clauses 9.33 and 9.43.
The flow graph segment `<stop-component-op>` in figure 125 defines the execution of the stop component operation.

```plaintext
if (Entity.VALUE-STACK.top() == 'all component') {
    Entity.VALUE-STACK.pop(); // clean value stack
    if (Entity != MTC) { // 'all' not allowed
        ***DYNAMIC-ERROR***
    } else {
        Entity.NEXT-CONTROL(true);
    }
} else {
    Entity.NEXT-CONTROL(false);
}
```

```plaintext
if (ALL-ENTITY-STATES.member(Entity.VALUE-STACK.top())) {
    Entity.NEXT-CONTROL(true); // Component is not removed from the test system
} else {
    Entity.NEXT-CONTROL(false); // Component is killed
}
```

```plaintext
if (Entity.VALUE-STACK.top().KEEP-ALIVE == true) {
    Entity.NEXT-CONTROL(true); // Component is not removed from the test system
} else {
    Entity.NEXT-CONTROL(false); // Component is killed
}
```

Figure 125: Flow graph segment `<stop-component-op>`
9.49.1 Void

9.49.2 Flow graph segment <stop-alive-component>

The <stop-alive-component> flow graph segment in figure 126 describes the stopping of a parallel test component, i.e. not the MTC or module control, which has been created with an alive clause. The effect is that the test case verdict TC-VERDICT and the list of terminated test components (DONE) are updated and that the component changes its status to BLOCKED. The <stop-alive-component> flow graph assumes that the identifier of the component to be stopped is on top of the value stack of the component that executes the segment.

```
segment <stop-alive-component>
let { // local scope
    var {myEntity : Entity.VALUE-STACK.top();
        var compVarScope := copy(myEntity.DATA-STATE.first());
        var compTimerScope := copy(myEntity.TIMER-STATE.first());
        var compPortScope := copy(myEntity.PORT-REF.first());

    // for test continuation, if stop is executed by another component
    if (Entity != myEntity()) {
        Entity.VALUE-STACK.pop(); // clean value stack
        Entity.NEXT-CONTROL(True);
    }

    // Update test case verdict
    if (myEntity.E-VERDICT == fail or TC-VERDICT == fail) {
        TC-VERDICT := fail;
    }
    else {
        if (myEntity.E-VERDICT == inconc or TC-VERDICT == inconc) {
            TC-VERDICT := inconc;
        }
        else {
            if (myEntity.E-VERDICT == pass or TC-VERDICT == pass) {
                TC-VERDICT := pass;
            }
        }
    }

    // Update of DONE
    DONE.append((myEntity, E-VERDICT));  // Update of DONE

    // Update of component state
    myEntity.STATUS := BLOCKED;
    myEntity.CONTROL-STACK := NULL;
    myEntity.DEFAULT-LIST := NULL;
    myEntity.VALUE-STACK := NULL;
    myEntity.VALUE-STACK.push(MARK);  // for component scope
    myEntity.STATUS := IDLE;
    myEntity.TIMER-STATE := NULL;
    myEntity.TIMER-STATE.add(compTimerScope);
    myEntity.TIMER-REF := NULL;
    myEntity.TIMER-REF.add(compTimerScope);
    myEntity.SNAP-ALIVE := NULL;
    myEntity.SNAP-DONE := NULL;
    myEntity.SNAP-KILLED := NULL;
}

// End of local scope
RETURN;
```

Figure 126: Flow graph segment <stop-alive-component>
9.49.3 Flow graph segment <stop-all-comp>

The <stop-all-comp> flow graph segment in figure 127 describes the stopping of all parallel test components of a test case.

```plaintext
let // local scope
  var myEntity := ALL-ENTITY-STATES.next(MTC);

  Entity.VALUE-STACK.push(MARK)
  while (myEntity != NULL) {
    Entity.VALUE-STACK.push(myEntity);
    myEntity := ALL-ENTITY-STATES.next(myEntity);
  }

  Entity.NEXT-CONTROL(true);
  RETURN;

if (Entity.VALUE-STACK.top().KEEP-ALIVE == true) {
  Entity.NEXT-CONTROL(true);
} else {
  Entity.NEXT-CONTROL(false);
}
RETURN;
```

Figure 127: Flow graph segment <stop-all-comp>
9.50 Stop execution statement

The syntactical structure of the `stop` execution statement is:

```
stop
```

The effect of the `stop` execution statement depends on the entity that executes the `stop` execution statement:

a) If `stop` is performed by the module control, the test campaign ends, i.e. all test components and the module control disappear from the module state. This is semantically similar to the execution of a `kill` statement by the module control.

b) If the `stop` is executed by the MTC, the test case ends. All parallel test components and the MTC stop and are removed from the test system. The global test case verdict is updated and pushed onto the value stack of the module control. Control is given back to the module control. This is semantically similar to the execution of a `kill` statement by the MTC.

c) If the `stop` is executed by a test component, the global test case verdict `TC-VERDICT` and the global `DONE` list are updated. If the test component is created with an alive clause. The status of the component is set to BLOCKED and it may be started again. Otherwise the component is removed from the test system.

The flow graph segment `<stop-exec-stmt>` in figure 128 describes the execution of the `stop` statement.

![Flow graph segment `<stop-exec-stmt>`](image-url)
9.51 Stop port operation

The syntactical structure of the **stop** port operation is:

```
<portId>.stop
```

The flow graph segment `<stop-port-op>` in figure 129 defines the execution of the **stop** port operation.

```plaintext
let { // Begin of local scope
  var portRef := NULL
  var portState := NULL;
  if (portId == "all port") {
    portState := ALL-PORT-STATES.first();
    while (portState != NULL) {
      if (portState_OWNER == Entity) {
        portState_STATUS := STOPPED
      }
      portState := ALL-PORT-STATES.next(portState);
    }
  } else {
    portRef := Entity.portId.COMP-PORT-NAME;
    GET-PORT(Entity, portRef).STATUS := STOPPED;
  } // End of scope
Entity.NEXT-CONTROL(true);
RETURN
}
```

Figure 129: Flow graph segment `<stop-port-op>`
9.52 Stop timer operation

The syntactical structure of the stop timer operation is:

`<timerId>.stop`

The flow graph segment `<stop-timer-op>` in figure 130 defines the execution of the stop timer operation.

The all keyword is handled as a special value of timerId.

```plaintext
segment <stop-timer-op>

stop-timer-op (timerId)

// The timer reference <timerId> is copied
// into the node attribute 'timerId'

if (timerId == 'all timer') {
  Entity.TIMER-STATE.change.change(TIMER-SET(), STATUS, IDLE));
  Entity.TIMER-STATE.change.change(TIMER-SET(), ACT-DURATION, 0.0);
  Entity.TIMER-STATE.change.change(TIMER-SET(), TIME-LEFT, 0.0);
  // Note, the first parameter of the TIMER-SET function is
  // omitted, because it is applied to all timers in the
  // actual scope unit.
}
else {
  Entity.TIMER-SET(timerId, STATUS, IDLE);
  Entity.TIMER-SET(timerId, ACT-DURATION, 0.0);
  Entity.TIMER-SET(timerId, TIME-LEFT, 0.0);
}

Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 130: Flow graph segment `<stop-timer-op>`

9.53 System operation

The syntactical structure of the system operation is:

`system`

The flow graph segment `<system-op>` in figure 131 defines the execution of the system operation.

```plaintext
segment <system-op>

system-op

Entity.VALUE-STACK.push(system);
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 131: Flow graph segment `<system-op>`
9.53α Test case stop operation

The syntactical structure of the test case stop operation is:

\[
\text{testcase.stop} \ (<\text{informal-description}>)
\]

The behaviour of the test case stop operation is identical to the execution of a log statement (clause 9.31) followed by a dynamic error (clause 9.17.3). Flow graph segment \text{<test-case-stop-op>} in figure 131a defines the execution of the test case stop operation.

![Figure 131a: Flow graph segment <test-case-stop-op>](image)

9.54 Timer declaration

9.54.0 General

The syntactical structure of a timer declaration is:

\[
\text{timer} \ <\text{timerId}> \ [::= <\text{float-expression}>]
\]

The effect of a timer declaration is the creation of a new timer binding. The declaration of a default duration is optional. The default value is considered to be an expression that evaluates to a value of the type float.

The flow graph segment \text{<timer-declaration>} in figure 132 defines the execution of a timer declaration.

![Figure 132: Flow graph segment <timer-declaration>](image)
9.54.1 Flow graph segment <timer-decl-default>

The flow graph segment `<timer-decl-default>` in figure 133 defines the execution of a timer declaration where a default duration in form of an expression is provided.

```
segment <timer-decl-default>

<expression>

// The expression shall evaluate
// to a value of type float

timer-decl-default (timerId)

Entity.INIT-TIMER(timerId, Entity.VALUE-STACK.top());
Entity.VALUE-STACK.pop(); // clean VALUE-STACK

Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 133: Flow graph segment `<timer-decl-default>`

9.54.2 Flow graph segment <timer-decl-no-def>

The flow graph segment `<timer-decl-no-def>` in figure 134 defines the execution of a timer declaration where no default duration is provided, i.e. the default duration of the timer is undefined.

```
segment <timer-decl-no-def>

timer-decl-no-def (timerId)

Entity.INIT-TIMER(timerId, NONE);
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 134: Flow graph segment `<timer-decl-no-def>`
9.55 Timeout timer operation

The syntactical structure of the **timeout** timer operation is:

```xml
<timerId>.timeout
```

The flow graph segment `<timeout-timer-op>` in figure 135 defines the execution of the **timeout** timer operation.

```javascript
// The timer reference <timerId> is copied // into the node attribute 'timerId'
let { // local scope for variable myTimerList
    var myTimerList; // to store a list of timer Bindings
    if (timerId == 'any timer') {
        myTimerList := Entity.TIMER-STATE.first();
        timerId := NULL;
        while (myTimerList != NULL && timerId == NULL) {
            timerId := myTimerList.random(SNAP-STATUS == TIMEOUT);
            myTimerList := Entity.TIMER-STATE.next(myTimerList);
        }
    }
    if (timerId != NULL && Entity.timerId.SNAP-STATUS == TIMEOUT) {
        Entity.TIMER-SET(timerId, STATUS, IDLE);
        Entity.TIMER-SET(timerId, ACT-DURATION, 0.0);
        Entity.TIMER-SET(timerId, TIME-LEFT, 0.0);
        Entity.STATUS := ACTIVE;
        Entity.NEXT-CONTROL(true);
    } else {
        Entity.NEXT-CONTROL(false);
    }
} // end of local scope
RETURN;
```

**NOTE 1:** A **timeout** operation is embedded in an **alt** statement. Its evaluation is based on the actual snapshot, i.e. the decision is based on the **SNAP-STATUS** entry in the timer binding. If the timeout operation is successful, i.e. **SNAP-STATUS** == **TIMEOUT**, the timer is set into an IDLE state and the component state changes from **SNAPSHOT** to **ACTIVE**.

**NOTE 2:** When the **timeout** evaluates to **true** or **false**, either execution continues with the statement that follows the **timeout** operation (**true** branch), or the next alternative in the **alt** statement has to be checked (**false** branch).

**NOTE 3:** The **any** keyword is treated like as special value of **timerId**.

---

**Figure 135:** Flow graph segment `<timeout-timer-op>`
9.56 Unmap operation

9.56.0 General

The syntactical structure of the unmap operation is:

\[
\text{unmap}(<\text{component-expression}>:<\text{portId1}> \ [,\text{system:<portId2>}] )
\]

The identifiers \(<\text{portId1}>\) and \(<\text{portId2}>\) are considered to be port identifiers of the corresponding test component and test system interface. The components to which the \(<\text{portId1}>\) belongs is referenced by means of the component reference \(<\text{component-expression}>\). The reference may be stored in variables or is returned by a function, i.e. it is an expression, which evaluates to a component reference. The value stack is used for storing the component reference.

The unmap operation can be used with one parameter pair and with two parameter pairs. The usage of the unmap operation with one parameter pair may unmap port mappings for one component or, if executed by the MTC for all components. The usage of the unmap operation with two parameter pairs allows to unmap one specific mapped port.

The operational semantics does not model the ports in the abstract test system interface. Therefore, only the parameter pair that identifies the component (or components, if the all component keyword is used) and the corresponding port (or ports, if the all port keyword is used) has to be considered here.

In the flow graph segment three cases are distinguished:

1) the mtc unmaps all mapped ports of all components;
2) all mapped ports of one component are unmapped; and
3) one port of one component is unmapped.
The execution of the **unmap** operation is defined by the flow graph segment `<unmap-op>` shown in figure 136.

```plaintext
// The Expression shall evaluate to a component reference. The result is pushed onto VALUE-STACK
if (Entity.VALUE-STACK.top() == "all component") {
    if {(Entity != MTC) OR (Entity == MTC && portId != "all port");
        ***DYNAMIC-ERROR***
    } else {
        Entity.VALUE-STACK.pop();
        Entity.NEXT-CONTROL(true);
    }
} else {
    Entity.VALUE-STACK.push(portId);
    Entity.NEXT-CONTROL(false);
}
RETURN;
```

```plaintext
if (Entity.VALUE-STACK.top() != "all port") {
    Entity.VALUE-STACK.pop();
    Entity.NEXT-CONTROL(true);
} else {
    Entity.NEXT-CONTROL(false);
}
RETURN;
```

Figure 136: Flow graph segment `<unmap-op>`
9.56.1 Flow graph segment <unmap-all>

The flow segment <unmap-all> defines the unmapping of all components at all mapped ports.

```
segment <unmap-all>

unmap-all

let { // local scope
  var port := ALL-PORT-STATES.first();
  var connection;
  while (port != NULL) {
    connection := port.CONNECTIONS.first();
    while (connection != NULL) {
      if (connection.REMOTE-ENTITY == system) {
        port.CONNECTIONS.delete(connection);
        connection := port.CONNECTIONS.first();
      } else {
        connection := NULL; // connected port
      }
    }
    port := ALL-PORT-STATES.next(port)
  }
} // End of local scope

Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 136a: Flow graph segment <unmap-all>
9.56.2 Flow graph segment <unmap-comp>

The flow segment <unmap-comp> defines the unmapping of all mapped ports of a specified component.

```
let { // local scope
  var comp := Entity.VALUE-STACK.top();
  var connection;
  var port := ALL-PORT-STATES.first();
  while (port != NULL) {
    if (port.OWNER == comp) { // port of comp
      connection := port.CONNECTIONS.first();
      if (connection.REMOTE-ENTITY == system) { // mapped port of comp
        port.CONNECTIONS.delete(connection);
      }
    }
    port := ALL-PORT-STATES.next(port);
  }
  Entity.VALUE-STACK.pop(); // clear value stack
} // End of local scope
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 136b: Flow graph segment <unmap-comp>
9.56.3 Flow graph segment `<unmap-port>`

The flow segment `<unmap-port>` defines the **unmap** operation for a specific mapped port.

```plaintext
segment `<unmap-port>`

let [ // local scope
  var portId;
  var comp;
  var port;
  var connection;

  portId := Entity.VALUE-STACK.top();
  Entity.VALUE-STACK.pop();
  comp := Entity.VALUE-STACK.top();
  Entity.VALUE-STACK.pop();
  port := GET-PORT(comp, portId);

  connection := port.CONNECTIONS.first();
  if (connection.REMOTE-ENTITY != SYSTEM) {
    ***DYNAMIC-ERROR*** // port is not a mapped port
  } else if (connection != NULL) {
    port.CONNECTIONS.delete(connection);
  } else { // do nothing, port is neither connected nor mapped
    Entity.NEXT-CONTROL(true);
    RETURN;
  }
```

**Figure 136c: Flow graph segment `<unmap-port>`**

9.57 Variable declaration

9.57.0 General

The syntactical structure of a variable declaration is:

```
var <varType> <varId> [:= <varType-expression>]
```

The initialization of a variable by providing an initial value (in form of an expression) is optional. The initial value is considered to be an expression that evaluates to a value of the type of the variable.

**NOTE:** Variables may be declared to be lazy or fuzzy. This operational semantics does not model lazy and fuzzy evaluation. It assumes that the correct evaluation of such variables is done internally.
The flow graph segment `<variable-declaration>` in figure 137 defines the execution of the declaration of a variable.

9.57.1 Flow graph segment `<var-declaration-init>`

The flow graph segment `<var-declaration-init>` in figure 138 defines the execution of a variable declaration where an initial value in form of an expression is provided.
9.57.2 Flow graph segment <var-declaration-undef>

The flow graph segment <var-declaration-undef> in figure 139 defines the execution of a variable declaration where no initial value is provided, i.e. the value of the variable is undefined.

![Flow graph segment <var-declaration-undef>](image)

**Figure 139: Flow graph segment <var-declaration-undef>**

9.58 While statement

The syntactical structure of the while statement is:

```
while (<boolean-expression>) <statement-block>
```

The execution of a while statement is defined by the flow graph segment <while-stmt> shown in figure 140.

![Flow graph segment <while-stmt>](image)

**Figure 140: Flow graph segment <while-stmt>**
10 Lists of operational semantic components

10.1 Functions and states

Table 1

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACT-DURATION</td>
<td>Duration with which an active timer has been started</td>
<td>8.3.2.4</td>
</tr>
<tr>
<td>add</td>
<td>List operation: adds an item as first element to a list</td>
<td>8.3.1a.1</td>
</tr>
<tr>
<td>ADD-CON</td>
<td>Adds a connection to a port state</td>
<td>8.3.3.2</td>
</tr>
<tr>
<td>ALL-ENTITY-STATES</td>
<td>Component states in module state</td>
<td>8.3.1.1</td>
</tr>
<tr>
<td>ALL-PORT-STATES</td>
<td>Port states in module state</td>
<td>8.3.1</td>
</tr>
<tr>
<td>append</td>
<td>List operation: appends an item as last element to a list</td>
<td>8.3.1a.1</td>
</tr>
<tr>
<td>APPLY-OPERATOR</td>
<td>Application of operators like +, -, or /</td>
<td>8.6.2</td>
</tr>
<tr>
<td>change</td>
<td>List operation: changes all elements of a list</td>
<td>8.3.1a.1</td>
</tr>
<tr>
<td>clear</td>
<td>Stack operation &quot;clear&quot;: clears a stack</td>
<td>8.3.2.1</td>
</tr>
<tr>
<td>clear-until</td>
<td>Stack operation &quot;clear-until&quot;: pops items until a specific item is top element in the stack</td>
<td>8.3.2.1</td>
</tr>
<tr>
<td>CONNECTIONS-LIST</td>
<td>List of connections of a port</td>
<td>8.3.3</td>
</tr>
<tr>
<td>CONTINUE-COMPONENT</td>
<td>The actual component continues its execution</td>
<td>8.6.2</td>
</tr>
<tr>
<td>CONTROL-STACK</td>
<td>Stack of flow graph nodes denoting the actual control state of an entity</td>
<td>8.3.2</td>
</tr>
<tr>
<td>DATA-STATE</td>
<td>Data state in an entity state</td>
<td>8.3.2</td>
</tr>
<tr>
<td>DEF-DURATION</td>
<td>Default Duration of a timer</td>
<td>8.3.2.4</td>
</tr>
<tr>
<td>DEFAULT-LIST</td>
<td>List of active defaults in an entity state</td>
<td>8.3.2</td>
</tr>
<tr>
<td>DEFAULT-POINTER</td>
<td>Points to the actual default during the default evaluation</td>
<td>8.3.2</td>
</tr>
<tr>
<td>DEL-CON</td>
<td>Deletes a connection from a port state</td>
<td>8.3.3.2</td>
</tr>
<tr>
<td>DEL-ENTITY</td>
<td>Deletes an entity from a module state</td>
<td>8.3.4</td>
</tr>
<tr>
<td>DEL-TIMER-SCOPE</td>
<td>Deletes a timer scope</td>
<td>8.3.2.5</td>
</tr>
<tr>
<td>DEL-VAR-SCOPE</td>
<td>Deletes a variable scope</td>
<td>8.3.2.3</td>
</tr>
<tr>
<td>delete</td>
<td>List operation: deletes an item from a list</td>
<td>8.3.1a.1</td>
</tr>
<tr>
<td>dequeue</td>
<td>Queue operation: deletes the first element from a queue</td>
<td>8.3.3.2</td>
</tr>
<tr>
<td>DONE</td>
<td>Identifiers of terminated test components (part of module state)</td>
<td>8.3.1</td>
</tr>
<tr>
<td>E-VERDICT</td>
<td>Local test verdict of a test component</td>
<td>8.3.2</td>
</tr>
<tr>
<td>enqueue</td>
<td>Queue operation: puts an item as last element into a queue</td>
<td>8.3.3.2</td>
</tr>
<tr>
<td>first</td>
<td>Queue operation &quot;first&quot;: returns the first element of a queue</td>
<td>8.3.3.2</td>
</tr>
<tr>
<td>first</td>
<td>List operation: returns the first element of a list</td>
<td>8.3.1a.1</td>
</tr>
<tr>
<td>GET-FLOW-GRAph</td>
<td>Retrieves the start node of a flow graph</td>
<td>8.2.7</td>
</tr>
<tr>
<td>GET-PORT</td>
<td>Retrieves a port reference</td>
<td>8.3.3.2</td>
</tr>
<tr>
<td>GET-REMOTE-PORT</td>
<td>Retrieves the reference of a remote port</td>
<td>8.3.3.2</td>
</tr>
<tr>
<td>GET-TIMER-LOC</td>
<td>Retrieves location of a timer</td>
<td>8.3.2.5</td>
</tr>
<tr>
<td>GET-UNIQUE-ID</td>
<td>Returns a new unique identifier when it is called</td>
<td>8.6.2</td>
</tr>
<tr>
<td>GET-VAR-LOC</td>
<td>Retrieves location of a variable</td>
<td>8.3.2.3</td>
</tr>
<tr>
<td>INIT-CALL-RECORD</td>
<td>Initializes variables for parameters for procedure-based communication in the actual scope unit of the test component</td>
<td>8.5.1</td>
</tr>
<tr>
<td>INIT-FLOW-GRAPHS</td>
<td>Initializes the flow graph handling</td>
<td>8.6.2</td>
</tr>
<tr>
<td>INIT-TIMER</td>
<td>Creates a new timer binding</td>
<td>8.3.2.5</td>
</tr>
<tr>
<td>INIT-TIMER-LOC</td>
<td>Creates a new timer binding with an existing location</td>
<td>8.3.2.5</td>
</tr>
<tr>
<td>INIT-TIMER-SCOPE</td>
<td>Initializes a new timer scope</td>
<td>8.3.2.5</td>
</tr>
<tr>
<td>INIT-VAR</td>
<td>Creates a new variable binding</td>
<td>8.3.2.3</td>
</tr>
<tr>
<td>INIT-VAR-LOC</td>
<td>Creates a new variable binding with an existing location</td>
<td>8.3.2.3</td>
</tr>
<tr>
<td>INIT-VAR-SCOPE</td>
<td>Initializes a new variable scope</td>
<td>8.3.2.3</td>
</tr>
<tr>
<td>length</td>
<td>List operation: returns the length of a list</td>
<td>8.3.1a.1</td>
</tr>
<tr>
<td>M-CONTROL</td>
<td>Identifier of module control in module state</td>
<td>8.3.1</td>
</tr>
<tr>
<td>MATCH-ITEM</td>
<td>Checks if a received message, call, reply or exception matches with a receiving operation</td>
<td>8.4.5</td>
</tr>
<tr>
<td>member</td>
<td>List operation: checks if an item is element of a list</td>
<td>8.3.1a.1</td>
</tr>
<tr>
<td>MTC</td>
<td>Reference to MTC in module state</td>
<td>8.3.1</td>
</tr>
<tr>
<td>NEW-CALL-RECORD</td>
<td>Creates a call record for a function call</td>
<td>8.5.1</td>
</tr>
<tr>
<td>NEW-ENTITY</td>
<td>Creates a new entity state</td>
<td>8.3.2.1</td>
</tr>
<tr>
<td>NEW-PORT</td>
<td>Creates a new port</td>
<td>8.3.2.2</td>
</tr>
</tbody>
</table>
### 10.2 Special keywords

Table 2

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
<th>Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVE</td>
<td>STATUS of an entity state</td>
<td>8.3.2</td>
</tr>
<tr>
<td>BLOCKED</td>
<td>STATUS of an entity state</td>
<td>8.3.2</td>
</tr>
<tr>
<td>BREAK</td>
<td>STATUS of an entity state</td>
<td>8.3.2</td>
</tr>
<tr>
<td>HALTED</td>
<td>STATUS of a port</td>
<td>8.3.3</td>
</tr>
<tr>
<td>HALT-MARKER</td>
<td>Used as marker in a port queue</td>
<td>8.3.3, 9.28a</td>
</tr>
<tr>
<td>IDLE</td>
<td>STATUS of a timer state</td>
<td>8.3.2</td>
</tr>
<tr>
<td>MARK</td>
<td>Used as mark for VALUE-STACK</td>
<td>8.3.2</td>
</tr>
<tr>
<td>NONE</td>
<td>Used to describe an undefined value</td>
<td>8.3.2.3, 8.3.2.5, 8.3.3.2</td>
</tr>
<tr>
<td>NULL</td>
<td>Symbolic value for pointer and pointer-like types to indicate that nothing is addressed</td>
<td>8.3.1a.1, 8.3.2.1, 8.3.3, 8.3.3.2, 8.6.1.1</td>
</tr>
<tr>
<td>REPEAT</td>
<td>STATUS of an entity state</td>
<td>8.3.2</td>
</tr>
<tr>
<td>RUNNING</td>
<td>STATUS of a timer state</td>
<td>8.3.2.4</td>
</tr>
<tr>
<td>SNAPSHOT</td>
<td>STATUS of an entity state</td>
<td>8.3.2</td>
</tr>
</tbody>
</table>

**ETSI**
### 10.3 Flow graphs of TTCN-3 behaviour descriptions

**Table 3**

<table>
<thead>
<tr>
<th>Reference</th>
<th>Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module control</td>
<td>8.2.2</td>
</tr>
<tr>
<td>Test cases</td>
<td>8.2.3</td>
</tr>
<tr>
<td>Functions</td>
<td>8.2.4</td>
</tr>
<tr>
<td>Altsteps</td>
<td>8.2.5</td>
</tr>
<tr>
<td>Component type definitions</td>
<td>8.2.6</td>
</tr>
</tbody>
</table>

### 10.4 Flow graph segments

**Table 4**

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Related TTCN-3 construct</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;action-stmt&gt;</td>
<td>action statement</td>
<td>36</td>
</tr>
<tr>
<td>&lt;activate-stmt&gt;</td>
<td>activate statement</td>
<td>37</td>
</tr>
<tr>
<td>&lt;alive-component-op&gt;</td>
<td>alive component operation</td>
<td>37a</td>
</tr>
<tr>
<td>&lt;alive-comp-act&gt;</td>
<td>alive component operation</td>
<td>37b</td>
</tr>
<tr>
<td>&lt;alive-comp-snap&gt;</td>
<td>alive component operation</td>
<td>37c</td>
</tr>
<tr>
<td>&lt;alt-stmt&gt;</td>
<td>alt statement</td>
<td>38</td>
</tr>
<tr>
<td>&lt;altstep-call&gt;</td>
<td>invocation of an altstep</td>
<td>44</td>
</tr>
<tr>
<td>&lt;altstep-call-branch&gt;</td>
<td>alt statement</td>
<td>41</td>
</tr>
<tr>
<td>&lt;assignment-stmt&gt;</td>
<td>assignment</td>
<td>45</td>
</tr>
<tr>
<td>&lt;b-call-with-duration&gt;</td>
<td>call operation</td>
<td>52</td>
</tr>
<tr>
<td>&lt;b-call-without-duration&gt;</td>
<td>call operation</td>
<td>51</td>
</tr>
<tr>
<td>&lt;blocking-call-op&gt;</td>
<td>call operation</td>
<td>47</td>
</tr>
<tr>
<td>&lt;break-altstep-stmt&gt;</td>
<td>break statement (leaving an altstep)</td>
<td>45a</td>
</tr>
<tr>
<td>&lt;call-op&gt;</td>
<td>call operation</td>
<td>46</td>
</tr>
<tr>
<td>&lt;call-reception-part&gt;</td>
<td>call operation</td>
<td>53</td>
</tr>
<tr>
<td>&lt;catch-op&gt;</td>
<td>catch operation</td>
<td>55</td>
</tr>
<tr>
<td>&lt;catch-timeout-exception&gt;</td>
<td>call operation</td>
<td>54</td>
</tr>
<tr>
<td>&lt;check-op&gt;</td>
<td>check operation</td>
<td>56</td>
</tr>
<tr>
<td>&lt;check-with-sender&gt;</td>
<td>check operation</td>
<td>57</td>
</tr>
<tr>
<td>&lt;check-without-sender&gt;</td>
<td>check operation</td>
<td>58</td>
</tr>
<tr>
<td>&lt;checkstate-port-op&gt;</td>
<td>checkstate operation</td>
<td>58a</td>
</tr>
<tr>
<td>&lt;check-port-status&gt;</td>
<td>checkstate operation</td>
<td>58b</td>
</tr>
<tr>
<td>&lt;check-port-connection&gt;</td>
<td>checkstate operation</td>
<td>58c</td>
</tr>
<tr>
<td>&lt;clear-port-op&gt;</td>
<td>clear port operation</td>
<td>59</td>
</tr>
<tr>
<td>&lt;connect-op&gt;</td>
<td>connect operation</td>
<td>60</td>
</tr>
<tr>
<td>&lt;constant-definition&gt;</td>
<td>constant definition</td>
<td>61</td>
</tr>
<tr>
<td>&lt;create-op&gt;</td>
<td>create operation</td>
<td>62</td>
</tr>
<tr>
<td>&lt;deactivate-all-defaults&gt;</td>
<td>deactivate statement</td>
<td>63c</td>
</tr>
<tr>
<td>&lt;deactivate-one-default&gt;</td>
<td>deactivate statement</td>
<td>63b</td>
</tr>
<tr>
<td>&lt;deactivate-stmt&gt;</td>
<td>deactivate statement</td>
<td>63a</td>
</tr>
<tr>
<td>&lt;default-evocation&gt;</td>
<td>alt statement</td>
<td>43</td>
</tr>
<tr>
<td>&lt;disconnect-op&gt;</td>
<td>disconnect operation</td>
<td>64</td>
</tr>
<tr>
<td>&lt;disconnect-one-par-pair&gt;</td>
<td>disconnect operation</td>
<td>64a</td>
</tr>
<tr>
<td>&lt;disconnect-all&gt;</td>
<td>disconnect operation</td>
<td>64b</td>
</tr>
<tr>
<td>&lt;disconnect-comp&gt;</td>
<td>disconnect operation</td>
<td>64c</td>
</tr>
<tr>
<td>Identifier</td>
<td>Related TTCN-3 construct</td>
<td>Reference</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>&lt;disconnect-port&gt;</td>
<td>disconnect operation</td>
<td>64d 9.14.4</td>
</tr>
<tr>
<td>&lt;disconnect-two-par-pairs&gt;</td>
<td>disconnect operation</td>
<td>64e 9.14.5</td>
</tr>
<tr>
<td>&lt;do-while-stmt&gt;</td>
<td>do-while statement</td>
<td>65 9.15</td>
</tr>
<tr>
<td>&lt;done-op&gt;</td>
<td>done component operation</td>
<td>66 9.16</td>
</tr>
<tr>
<td>dynamic-error</td>
<td>execute statement</td>
<td>69a 19.17.3</td>
</tr>
<tr>
<td>&lt;else-branch&gt;</td>
<td>alt statement</td>
<td>42 9.3.4</td>
</tr>
<tr>
<td>&lt;execute-stmt&gt;</td>
<td>execute statement</td>
<td>67 9.17</td>
</tr>
<tr>
<td>&lt;execute-timeout&gt;</td>
<td>execute statement</td>
<td>69 9.17.2</td>
</tr>
<tr>
<td>&lt;execute-without-timeout&gt;</td>
<td>execute statement</td>
<td>68 9.17.1</td>
</tr>
<tr>
<td>&lt;expression&gt;</td>
<td>expression</td>
<td>70 9.18</td>
</tr>
<tr>
<td>&lt;final-ize-component-init&gt;</td>
<td>used in component type definitions</td>
<td>75 9.19</td>
</tr>
<tr>
<td>&lt;for-stmt&gt;</td>
<td>for statement</td>
<td>79 9.23</td>
</tr>
<tr>
<td>&lt;func-op-call&gt;</td>
<td>expression</td>
<td>73 9.18.3</td>
</tr>
<tr>
<td>&lt;function-call&gt;</td>
<td>call of a function</td>
<td>80 9.24</td>
</tr>
<tr>
<td>&lt;getcall-op&gt;</td>
<td>getcall operation</td>
<td>86 9.25</td>
</tr>
<tr>
<td>&lt;getreply-op&gt;</td>
<td>getreply operation</td>
<td>87 9.26</td>
</tr>
<tr>
<td>&lt;getverdict-op&gt;</td>
<td>getverdict operation</td>
<td>88 9.27</td>
</tr>
<tr>
<td>&lt;goto-stmt&gt;</td>
<td>goto statement</td>
<td>89 9.28</td>
</tr>
<tr>
<td>&lt;halt-port-op&gt;</td>
<td>halt port operation</td>
<td>89a 9.28a</td>
</tr>
<tr>
<td>&lt;if-else-stmt&gt;</td>
<td>if-else statement</td>
<td>90 9.29</td>
</tr>
<tr>
<td>&lt;init-component-scope&gt;</td>
<td>used in component type definitions</td>
<td>76 9.20</td>
</tr>
<tr>
<td>&lt;init-scope-with-runs-on&gt;</td>
<td>used in function and altstep definitions</td>
<td>76a 9.20a</td>
</tr>
<tr>
<td>&lt;init-scope-without-runs-on&gt;</td>
<td>used in function and altstep definitions</td>
<td>76b 9.20b</td>
</tr>
<tr>
<td>&lt;kill-all-comp&gt;</td>
<td>kill component operation</td>
<td>90d 9.29a.3</td>
</tr>
<tr>
<td>&lt;kill-component&gt;</td>
<td>kill component operation</td>
<td>90c 9.29a.2</td>
</tr>
<tr>
<td>&lt;kill-component-op&gt;</td>
<td>kill component operation</td>
<td>90a 9.29a</td>
</tr>
<tr>
<td>&lt;kill-control&gt;</td>
<td>kill execution statement</td>
<td>90f 9.29b.1</td>
</tr>
<tr>
<td>&lt;kill-exec-stmt&gt;</td>
<td>kill execution statement</td>
<td>90e 9.29b</td>
</tr>
<tr>
<td>&lt;kill-mtc&gt;</td>
<td>kill component operation</td>
<td>90b 9.29a.1</td>
</tr>
<tr>
<td>&lt;killed-op&gt;</td>
<td>killed component operation</td>
<td>90g 9.29c</td>
</tr>
<tr>
<td>&lt;label-stmt&gt;</td>
<td>label statement</td>
<td>91 9.30</td>
</tr>
<tr>
<td>&lt;lit-value&gt;</td>
<td>expression</td>
<td>71 9.18.1</td>
</tr>
<tr>
<td>&lt;log-stmt&gt;</td>
<td>log statement</td>
<td>92 9.31</td>
</tr>
<tr>
<td>&lt;map-op&gt;</td>
<td>map operation</td>
<td>93 9.32</td>
</tr>
<tr>
<td>&lt;mtc-op&gt;</td>
<td>mtc operation</td>
<td>94 9.33</td>
</tr>
<tr>
<td>&lt;nb-call-without-receiver&gt;</td>
<td>call operation</td>
<td>50 9.6.2</td>
</tr>
<tr>
<td>&lt;nb-call-with-one-receiver&gt;</td>
<td>call operation</td>
<td>49 9.6.1</td>
</tr>
<tr>
<td>&lt;nb-call-with-multiple-receivers&gt;</td>
<td>call operation</td>
<td>49a 9.6.1a</td>
</tr>
<tr>
<td>&lt;non-blocking-call-op&gt;</td>
<td>call operation</td>
<td>48 9.6</td>
</tr>
<tr>
<td>&lt;operator-appl&gt;</td>
<td>expression</td>
<td>74 9.18.4</td>
</tr>
<tr>
<td>&lt;parameter-handling&gt;</td>
<td>handling of parameters of functions, altsteps and test cases</td>
<td>77 9.21</td>
</tr>
<tr>
<td>&lt;port-declaration&gt;</td>
<td>port declaration</td>
<td>95 9.34</td>
</tr>
<tr>
<td>&lt;predef-ext-func-call&gt;</td>
<td>call of a function (call of a pre-defined or external function)</td>
<td>85 9.24.5</td>
</tr>
<tr>
<td>&lt;raise-op&gt;</td>
<td>raise operation</td>
<td>96 9.35</td>
</tr>
<tr>
<td>&lt;raise-with-one-receiver-op&gt;</td>
<td>raise operation</td>
<td>97 9.35.1</td>
</tr>
<tr>
<td>&lt;raise-with-multiple-receivers-op&gt;</td>
<td>raise operation</td>
<td>97a 9.35.1a</td>
</tr>
<tr>
<td>&lt;raise-without-receiver-op&gt;</td>
<td>raise operation</td>
<td>98 9.35.2</td>
</tr>
<tr>
<td>&lt;read-timer-op&gt;</td>
<td>read timer operation</td>
<td>99 9.36</td>
</tr>
<tr>
<td>&lt;receive-assignment&gt;</td>
<td>receive operation</td>
<td>103 9.37.3</td>
</tr>
<tr>
<td>&lt;receive-op&gt;</td>
<td>receive operation</td>
<td>100 9.37</td>
</tr>
<tr>
<td>&lt;receive-with-sender&gt;</td>
<td>receive operation</td>
<td>101 9.37.1</td>
</tr>
<tr>
<td>&lt;receive-without-sender&gt;</td>
<td>receive operation</td>
<td>102 9.37.2</td>
</tr>
<tr>
<td>&lt;receiving-branch&gt;</td>
<td>alt statement</td>
<td>40 9.3.2</td>
</tr>
<tr>
<td>&lt;ref-par-port-calc&gt;</td>
<td>call of a function (handling of port parameters)</td>
<td>83a 9.24.3a</td>
</tr>
<tr>
<td>&lt;ref-par-timer-calc&gt;</td>
<td>call of a function (handling of timer parameters)</td>
<td>83 9.24.3</td>
</tr>
<tr>
<td>&lt;ref-par-var-calc&gt;</td>
<td>call of a function (handling of reference parameters)</td>
<td>82 9.24.2</td>
</tr>
<tr>
<td>&lt;repeat-stmt&gt;</td>
<td>repeat statement</td>
<td>104 9.38</td>
</tr>
<tr>
<td>&lt;reply-op&gt;</td>
<td>reply operation</td>
<td>105 9.39</td>
</tr>
<tr>
<td>Identifier</td>
<td>Related TTCN-3 construct</td>
<td>Reference</td>
</tr>
<tr>
<td>-----------------------------------------------------</td>
<td>--------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>&lt;reply-with-one-receiver-op&gt;</td>
<td>reply operation</td>
<td>106</td>
</tr>
<tr>
<td>&lt;reply-with-multiple-receivers-op&gt;</td>
<td>reply operation</td>
<td>106a</td>
</tr>
<tr>
<td>&lt;reply-without-receiver-op&gt;</td>
<td>reply operation</td>
<td>107</td>
</tr>
<tr>
<td>&lt;return-stmt&gt;</td>
<td>return statement</td>
<td>108</td>
</tr>
<tr>
<td>&lt;return-with-value&gt;</td>
<td>return statement</td>
<td>109</td>
</tr>
<tr>
<td>&lt;return-without-value&gt;</td>
<td>return statement</td>
<td>110</td>
</tr>
<tr>
<td>&lt;running-component-op&gt;</td>
<td>component running operation</td>
<td>111</td>
</tr>
<tr>
<td>&lt;running-comp-act&gt;</td>
<td>component running operation</td>
<td>112</td>
</tr>
<tr>
<td>&lt;running-comp-snap&gt;</td>
<td>component running operation</td>
<td>113</td>
</tr>
<tr>
<td>&lt;running-timer-op&gt;</td>
<td>timer running operation</td>
<td>114</td>
</tr>
<tr>
<td>&lt;self-op&gt;</td>
<td>self operation</td>
<td>115</td>
</tr>
<tr>
<td>&lt;send-op&gt;</td>
<td>send operation</td>
<td>116</td>
</tr>
<tr>
<td>&lt;send-with-one-receiver-op&gt;</td>
<td>send operation</td>
<td>117</td>
</tr>
<tr>
<td>&lt;send-with-multiple-receivers-op&gt;</td>
<td>send operation</td>
<td>117a</td>
</tr>
<tr>
<td>&lt;send-without-receiver-op&gt;</td>
<td>send operation</td>
<td>118</td>
</tr>
<tr>
<td>&lt;setverdict-op&gt;</td>
<td>setverdict operation</td>
<td>119</td>
</tr>
<tr>
<td>&lt;start-component-op&gt;</td>
<td>start component operation</td>
<td>120</td>
</tr>
<tr>
<td>&lt;start-port-op&gt;</td>
<td>start port operation</td>
<td>121</td>
</tr>
<tr>
<td>&lt;start-timer-op&gt;</td>
<td>start timer operation</td>
<td>122</td>
</tr>
<tr>
<td>&lt;start-timer-op-default&gt;</td>
<td>start timer operation</td>
<td>123</td>
</tr>
<tr>
<td>&lt;start-timer-op-duration&gt;</td>
<td>start timer operation</td>
<td>124</td>
</tr>
<tr>
<td>&lt;statement-block&gt;</td>
<td>block of statements in compound statements</td>
<td>78</td>
</tr>
<tr>
<td>&lt;stop-component-op&gt;</td>
<td>stop component operation</td>
<td>125</td>
</tr>
<tr>
<td>&lt;stop-all-comp&gt;</td>
<td>stop component operation (all component.stop)</td>
<td>126</td>
</tr>
<tr>
<td>&lt;stop-exec-stmt&gt;</td>
<td>stop execution statement</td>
<td>128</td>
</tr>
<tr>
<td>&lt;stop-port-op&gt;</td>
<td>stop port operation</td>
<td>129</td>
</tr>
<tr>
<td>&lt;stop-timer-op&gt;</td>
<td>stop timer operation</td>
<td>130</td>
</tr>
<tr>
<td>&lt;system-op&gt;</td>
<td>system operation</td>
<td>131</td>
</tr>
<tr>
<td>&lt;take-snapshot&gt;</td>
<td>alt statement</td>
<td>39</td>
</tr>
<tr>
<td>&lt;test-case-stop-op&gt;</td>
<td>test case stop operation</td>
<td>131a</td>
</tr>
<tr>
<td>&lt;timer-declaration&gt;</td>
<td>timer declaration</td>
<td>132</td>
</tr>
<tr>
<td>&lt;timer-decl-default&gt;</td>
<td>timer declaration</td>
<td>133</td>
</tr>
<tr>
<td>&lt;timer-decl-no-def&gt;</td>
<td>timer declaration</td>
<td>134</td>
</tr>
<tr>
<td>&lt;timeout-timer-op&gt;</td>
<td>timeout operation</td>
<td>135</td>
</tr>
<tr>
<td>&lt;unmap-op&gt;</td>
<td>unmap operation</td>
<td>136</td>
</tr>
<tr>
<td>&lt;unmap-all&gt;</td>
<td>unmap operation</td>
<td>136a</td>
</tr>
<tr>
<td>&lt;unmap-comp&gt;</td>
<td>unmap operation</td>
<td>136b</td>
</tr>
<tr>
<td>&lt;unmap-port&gt;</td>
<td>unmap operation</td>
<td>136c</td>
</tr>
<tr>
<td>&lt;user-def-func-call&gt;</td>
<td>call of a function (call of a user-defined function)</td>
<td>84</td>
</tr>
<tr>
<td>&lt;value-par-calculation&gt;</td>
<td>call of a function (handling of value parameters)</td>
<td>81</td>
</tr>
<tr>
<td>&lt;var-declaration-init&gt;</td>
<td>variable declaration</td>
<td>138</td>
</tr>
<tr>
<td>&lt;var-declaration-undef&gt;</td>
<td>variable declaration</td>
<td>139</td>
</tr>
<tr>
<td>&lt;var-value&gt;</td>
<td>expression</td>
<td>72</td>
</tr>
<tr>
<td>&lt;variable-declaration&gt;</td>
<td>variable declaration</td>
<td>137</td>
</tr>
<tr>
<td>&lt;while-stmt&gt;</td>
<td>while statement</td>
<td>140</td>
</tr>
</tbody>
</table>
## History

<table>
<thead>
<tr>
<th>Document history</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>V2.2.1</strong></td>
</tr>
<tr>
<td><strong>V3.1.1</strong></td>
</tr>
<tr>
<td><strong>V3.2.1</strong></td>
</tr>
<tr>
<td><strong>V3.3.1</strong></td>
</tr>
<tr>
<td><strong>V3.4.1</strong></td>
</tr>
<tr>
<td><strong>V4.1.1</strong></td>
</tr>
<tr>
<td><strong>V4.2.1</strong></td>
</tr>
<tr>
<td><strong>V4.4.1</strong></td>
</tr>
<tr>
<td><strong>V4.5.1</strong></td>
</tr>
<tr>
<td><strong>V4.6.1</strong></td>
</tr>
<tr>
<td><strong>V4.6.1</strong></td>
</tr>
</tbody>
</table>