International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.164

(11/2007)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

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

Testing and Test Control Notation version 3: TTCN-3 operational semantics

ITU-T Recommendation Z.164

ITU-T Z-SERIES RECOMMENDATIONS

LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

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

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

ITU-T Recommendation Z.164

Testing and Test Control Notation version 3: TTCN-3 operational semantics

Summary

ITU-T Recommendation Z.164 defines the operational semantics of TTCN-3 (*Testing and Test Control Notation 3*). The operational semantics are necessary to unambiguously interpret the specifications made with TTCN-3. This Recommendation is based on the TTCN-3 core language defined in [ITU-T Z.161].

Source

ITU-T Recommendation Z.164 was approved on 13 November 2007 by ITU-T Study Group 17 (2005-2008) under the ITU-T Recommendation A.8 procedure.

FOREWORD

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

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

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

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

NOTE

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

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

INTELLECTUAL PROPERTY RIGHTS

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

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

© ITU 2008

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

CONTENTS

		Page
1	Scope	1
2	References	1
3	Definitions and abbreviations	1
	3.1 Definitions	1
	3.2 Abbreviations	1
4	Introduction	1
5	Structure of this Recommendation	2
6	Restrictions	2
7	Replacement of short forms	2
	7.1 Order of replacement steps	3
	7.2 Replacement of global constants and module parameters	3
	7.3 Embedding single receiving operations into alt statements	3
	7.4 Embedding stand-alone altstep calls into alt statements	4
	7.5 Replacement of interleave statements	5
	7.6 Replacement of trigger operations	16
	7.7 Replacement of select-case statements	17
8	Flow graph semantics of TTCN-3	18
	8.1 Flow graphs	18
	8.2 Flow graph representation of TTCN-3 behaviour	23
	8.3 State definitions for TTCN-3 modules	29
	8.4 Messages, procedure calls, replies and exceptions	41
	8.5 Call records for functions, altsteps and test cases	43
	8.6 The evaluation procedure for a TTCN-3 module	44
9	Flow graph segments for TTCN-3 constructs	46
	9.1 Action statement	47
	9.2 Activate statement	47
	9.2a Alive component operation	48
	9.3 Alt statement	51
	9.4 Altstep call	57
	9.5 Assignment statement	57
	9.6 Call operation	57
	9.7 Catch operation	64
	9.8 Check operation	64
	9.9 Clear port operation	67
	9.10 Connect operation	68
	9.11 Constant definition	69
	9.12 Create operation	69
	9.13 Deactivate statement	70

		Page
9.14	Disconnect operation	72
9.15	Do-while statement	73
9.16	Done component operation	73
9.17	Execute statement	75
9.18	Expression	78
9.19	Flow graph segment <finalise-component-init>	81
9.20	Flow graph segment <init-component-scope>	82
9.20a	Flow graph segment <init-scope-with-runs-on>	82
9.20b	Flow graph segment <init-scope-without-runs-on>	83
9.21	Flow graph segment <parameter-handling>	83
9.22	Flow graph segment <statement-block>	84
9.23	For statement	85
9.24	Function call	86
9.25	Getcall operation	91
9.26	Getreply operation	91
9.27	Getverdict operation	91
9.28	Goto statement	92
9.29	If-else statement	92
9.29a	Kill component operation	93
9.29b	Kill execution statement	97
9.29c	Killed component operation	98
9.30	Label statement	100
9.31	Log statement	100
9.32	Map operation	100
9.33	Mtc operation	101
9.34	Port declaration	101
9.35	Raise operation	102
9.36	Read timer operation	105
9.37	Receive operation	106
9.38	Repeat statement	110
9.39	Reply operation	110
9.40	Return statement	114
9.41	Running component operation	117
9.42	Running timer operation	120
9.43	Self operation	121
9.44	Send operation	122
9.45	Setverdict operation	125
9.46	Start component operation	126
9.47	Start port operation	128
9.48	Start timer operation	128

Page

9.49 Stop component operation 130
9.50 Stop execution statement 133
9.51 Stop port operation 134
9.52 Stop timer operation 135
9.53 System operation 136
9.54 Timer declaration 136
9.55 Timeout timer operation 137
9.56 Unmap operation 138
9.57 Variable declaration 139
9.58 While statement 140

10 Lists of operational semantic components 141
10.1 Functions and states 141
10.2 Special keywords 144
10.3 Flow graphs of TTCN-3 behaviour descriptions 144
10.4 Flow graph segments 144

ITU-T Recommendation Z.164

Testing and Test Control Notation version 3: TTCN-3 operational semantics

1 Scope

This Recommendation defines the operational semantics of TTCN-3. This Recommendation is based on the TTCN-3 core language defined in [ITU-T Z.161].

2 References

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

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

3 Definitions and abbreviations

3.1 Definitions

This Recommendation uses the terms and definitions given in [ITU-T Z.161].

3.2 Abbreviations

This Recommendation uses the following abbreviations and acronyms:

BNF Backus-Nauer Form

MTC Main Test Component

SUT System Under Test

TTCN Testing and Test Control Notation

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 shall 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 [ITU-T Z.161]. 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 this Recommendation

This Recommendation 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 this Recommendation.

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.

7 Replacement of short forms

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; and
- **`select-case`** statements.

In addition to the handling of short forms, the operational semantics requires a special handling for module parameters and global constants, i.e., constants that are defined in the module definitions part. All references to module parameters and global constants shall be replaced by concrete values. This means, it is assumed that the value of module parameters and global constants 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 of a module control part without stop statement.

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 is 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 to call 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 altstep 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 the restrictions 1-5, 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 line. This is shown in Figure 1.

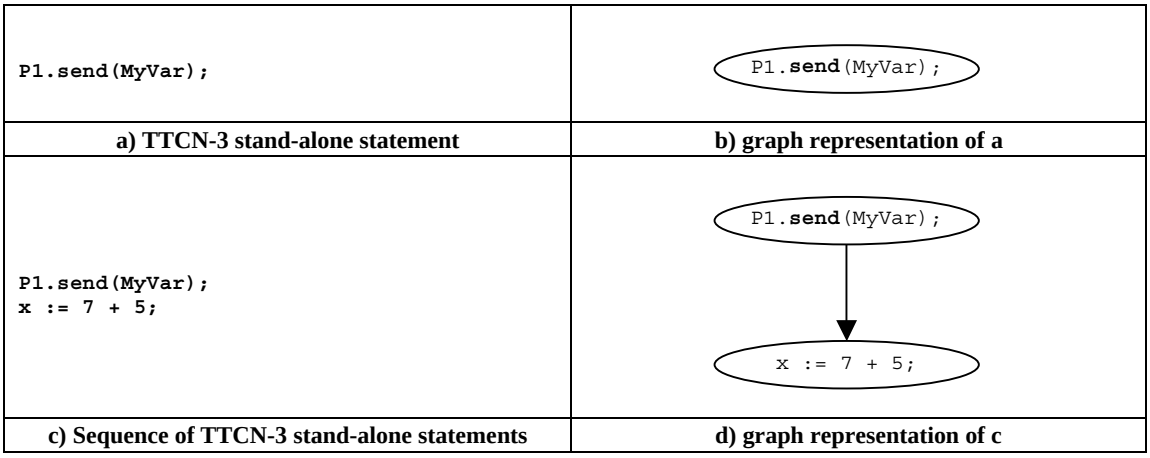


Figure 1 – Graph representation of TTCN-3 stand-alone statements

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.

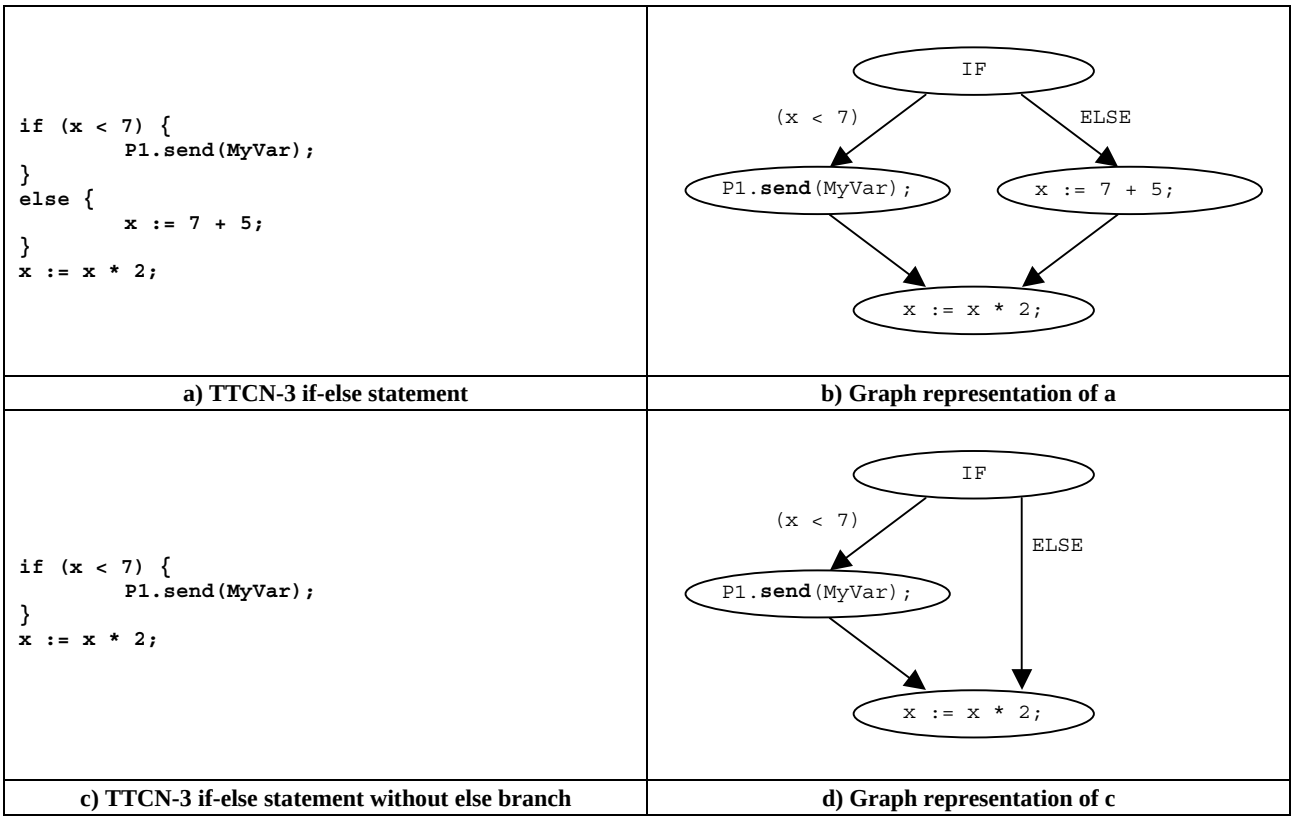
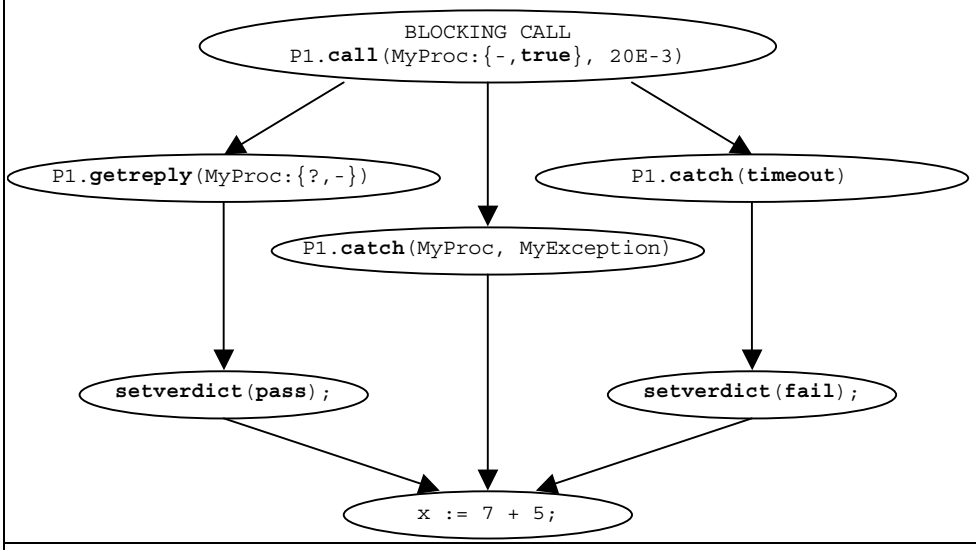


Figure 2 – Graph representation of TTCN-3 if-else statements

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.

```
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



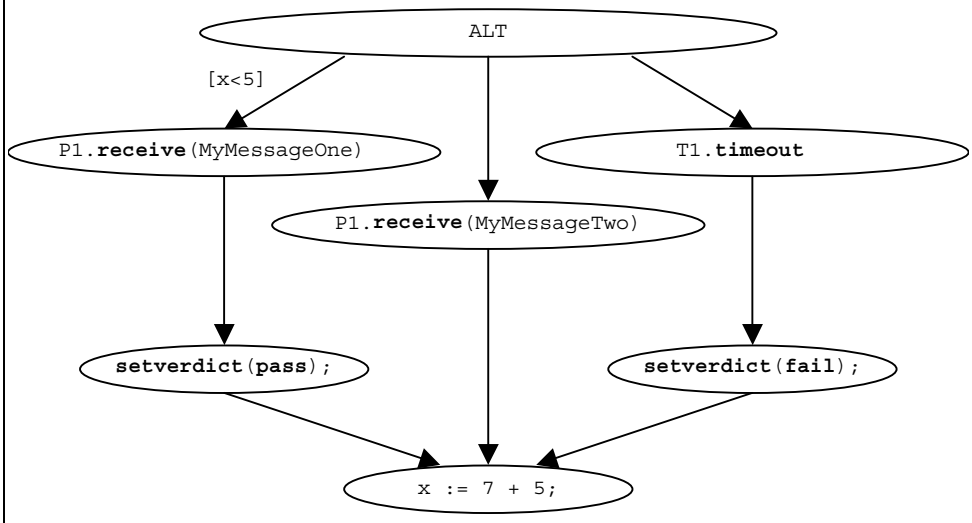
b) Graph representation of a

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



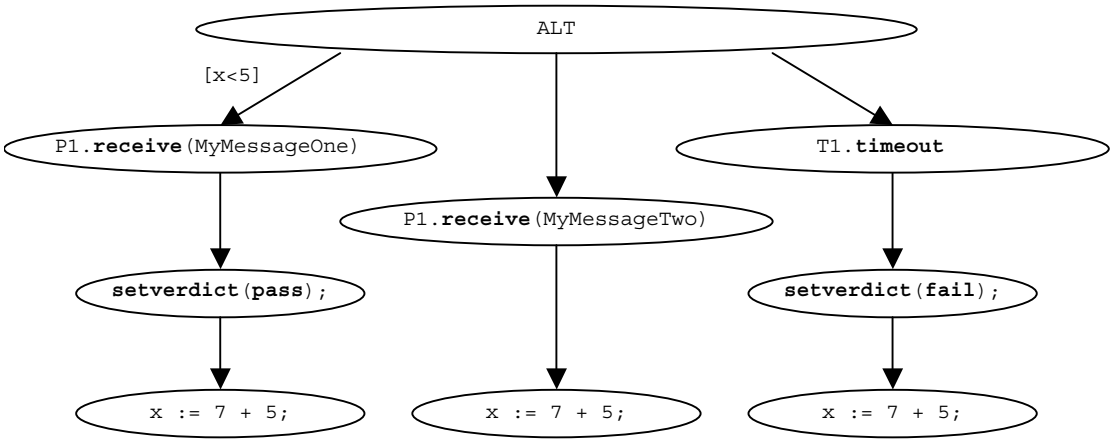
b) Graph representation of a

Figure 4 – Graph representation 1 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 representations. 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 2 of a TTCN-3 alt statement

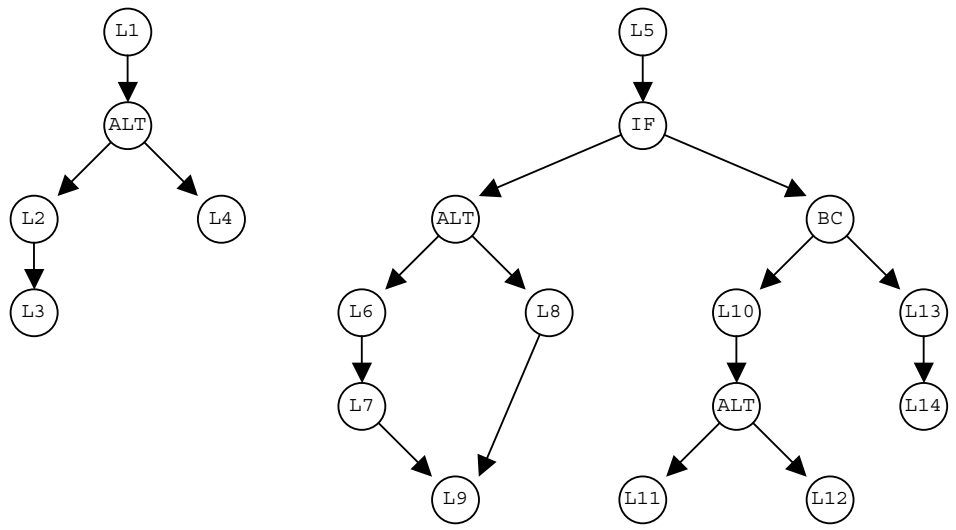
An **`interleave`** statement can be described by a graph that consists of a set of directed sub-graphs, each of which is constructed by means of stand-alone statements and the compound statements **`if-else`**, *blocking* **`call`** and **`alt`**. The directed sub-graphs describe the interleaved flows 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 {                                     // ALT
            [] P1.receive(M3) {                   // L2
                  setverdict(pass);               // L3
               }
            [] T1.timeout { }                        // L4
         }

      }
   [] P2.receive(M2) {                          // L5
         if (x < 5 ) {                             // IF
            alt {                                  // ALT
               [] P2.receive(M4) {                // L6
                     setverdict(pass);            // L7
                  }
               [] Comp1.done { }                  // L8
            }
            x := 7 + 5;                            // L9
         }
         else {
            P3.call(MyProcTempl, 20E-3) {    // BC (= BLOCKING CALL)
```

```
                [] P3.getreply(ReplyTempl) {     // L10
                      alt {                         // 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 $\boldsymbol{GI} = (\boldsymbol{St}, \boldsymbol{F})$ where:

$\boldsymbol{St}$ is the set of allowed TTCN-3 statements; and

$\boldsymbol{F} \subseteq (\boldsymbol{St} \times \boldsymbol{St})$ 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 <u>*REACHABLE*</u> function returns all statements that are reachable from a statement $\boldsymbol{s}$ in a graph $\boldsymbol{GI} = (\boldsymbol{St}, \boldsymbol{F})$:
  - <u>*REACHABLE*</u> $(\boldsymbol{s}, \boldsymbol{GI}) = \{ \boldsymbol{s} \} \cup \{ \boldsymbol{stmt} \mid \boldsymbol{stmt} \in \boldsymbol{St} \wedge \exists(\boldsymbol{s} = \boldsymbol{x_1}, \boldsymbol{x_2}, \ldots , \boldsymbol{x_n} = \boldsymbol{stmt})$ where $\boldsymbol{x_i} \in \boldsymbol{St}$, $i \in \{1\ldots n\} \wedge (\boldsymbol{x_i}, \boldsymbol{x_{i+1}}) \in \boldsymbol{F}\}$
- The <u>*SUCCESSORS*</u> function returns all successors of a statement $\boldsymbol{s}$ in a graph $\boldsymbol{GI} = (\boldsymbol{St}, \boldsymbol{F})$:
  - <u>*SUCCESSORS*</u> $(\boldsymbol{s}, \boldsymbol{GI}) = \{ \boldsymbol{stmt} \mid \boldsymbol{stmt} \in \boldsymbol{St} \wedge (\boldsymbol{s}, \boldsymbol{stmt}) \in \boldsymbol{F}\}$
- The <u>*ENABLED*</u> function returns all statements of a graph $\boldsymbol{GI} = (\boldsymbol{St}, \boldsymbol{F})$ which have no predecessors:
  - <u>*ENABLED*</u> $(\boldsymbol{GI}) = \{ \boldsymbol{stmt} \mid \boldsymbol{stmt} \in \boldsymbol{St} \wedge (\boldsymbol{F} \cap (\boldsymbol{S} \times \{\boldsymbol{s}\}) = \varnothing)\}$
- The <u>*KIND*</u> function returns the kind or type of a TTCN-3 statement in a graph representing an **`interleave`** statement.
- The <u>*DISCARD*</u> function deletes a statement $\boldsymbol{s}$ or a set of statements $\boldsymbol{S}$ from a graph $\boldsymbol{GI} = (\boldsymbol{St}, \boldsymbol{F})$ and returns the resulting graph $\boldsymbol{GI'} = (\boldsymbol{St'}, \boldsymbol{F'})$:

For single nodes:

– *DISCARD* ($\boldsymbol{s}$, $\boldsymbol{GI}$) = $\boldsymbol{GI'}$ where: $\boldsymbol{GI'}$ = ($\boldsymbol{St'}$, $\boldsymbol{F'}$), with $\boldsymbol{St'} = \boldsymbol{St} \backslash \{\boldsymbol{s}\}$ and $\boldsymbol{F'} = \boldsymbol{F} \cap (\boldsymbol{St} \backslash \{\boldsymbol{s}\} \times \boldsymbol{St} \backslash \{\boldsymbol{s}\})$.

For sets of nodes:

– *DISCARD* ($\boldsymbol{S}$, $\boldsymbol{GI}$) = $\boldsymbol{GI'}$ where: $\boldsymbol{GI'}$ = ($\boldsymbol{St'}$, $\boldsymbol{F'}$), with $\boldsymbol{St'} = \boldsymbol{St} \backslash \boldsymbol{S}$ and $\boldsymbol{F'} = \boldsymbol{F} \cap (\boldsymbol{St} \backslash \boldsymbol{S} \times \boldsymbol{St} \backslash \boldsymbol{S})$.

- The *RECEIVING* function takes a set of statements of a graph $\boldsymbol{GI}$ and returns all receiving statements:

  – *RECEIVING* (S) = { $stmt \mid stmt \in St \wedge KIND(stmt) \in$ {receive, trigger, getcall, getreply, catch, check, done, timeout}}

- The *RANDOM* function selects randomly an element $\boldsymbol{s}$ from a given set $\boldsymbol{S}$ and returns $\boldsymbol{s}$.

  – *RANDOM* ($\boldsymbol{S}$) = $\boldsymbol{s}$ where $\boldsymbol{s} \in \boldsymbol{S}$

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 == ∅) {   // We assume 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
            }
        }
        elseif (KIND(myStmt) == ALT) {
```

```
            for each (i in SUCCESSORS(myStmt, GI) {

            CONSTRUCT-SUCCESSORS(mystmt, DISCARD(REACHABLE(myStmt, GI)\REACHABLE(i, GI)));
                    // NEXT RECURSION STEP, the DISCARD(REACHABLE(myStmt, GI)\REACHABLE(i, GI))
                        // argument considers the branching of a flow of control due to different
                        // receiving events.
            }
        }
        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
                                        // interleaving is not influenced by an embedded alt statement
            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(S)
        }
    }
}
```

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.

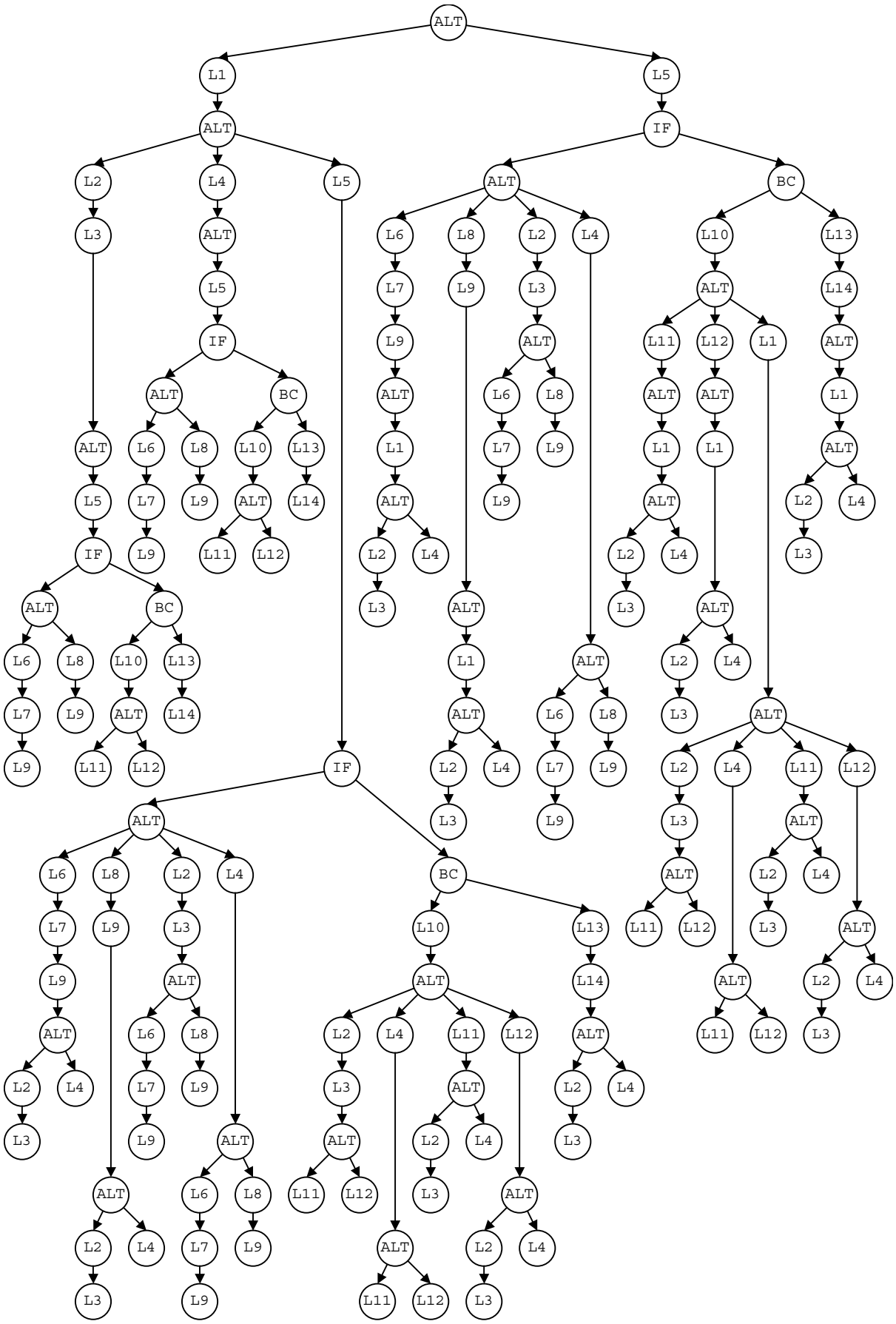


Figure 8 – Result of applying the algorithm in Figure 7 to the interleave statement in Figure 6

```
alt {                                                                   // ALT
    []  P1.receive(M1} {                                                // L1
            alt {                                                       // ALT
                []  P1.receive(M3) {                                    // L2
                        setverdict(pass);                               // L3
                        alt {                                           // ALT
                            []  P2.receive(M2) {                        // L5
                                    if (x < 5 ) {                       // IF
                                        alt {                           // ALT
                                                []  P2.receive(M4) {    // L6
                                                        setverdict(pass);// L7
                                                        x := 7 + 5;     // L9
                                                }
                                                []  Comp1.done {        // L8
                                                        x := 7 + 5;     // L9
                                    }   }           }
                                    else {
                                        P3.call(MyProcTempl, 20E-3) { // BC (= BLOCKING CALL)
                                            []  P3.getreply(ReplyTempl) {// L10
                                                    alt {               // ALT
                                                        []  P2.receive(M5) { }// L11
                                                        []  P2.receive(M6) { }// L12
                                                }   }
                                            []  P3.catch(timeout) {     // L13
                                                    setverdict(fail);   // L14
                    }   }       }   }   }       }
                []  T1.timeout {                                        // L4
                        alt {                                           // ALT
                            []  P2.receive(M2) {                        // L5
                                    if (x < 5 ) {                       // IF
                                        alt {                           // ALT
                                                []  P2.receive(M4) {    // L6
                                                        setverdict(pass);// L7
                                                        x := 7 + 5;     // L9
                                                }
                                                []  Comp1.done {        // L8
                                                        x := 7 + 5;     // L9
                                    }   }           }
                                    else {
                                        P3.call(MyProcTempl, 20E-3) { // BC (= BLOCKING CALL)
                                            []  P3.getreply(ReplyTempl) {// L10
                                                    alt {               // ALT
                                                        []  P2.receive(M5) { }// L11
                                                        []  P2.receive(M6) { }// L12
                                                }   }
                                            []  P3.catch(timeout) {     // L13
                                                    setverdict(fail);   // L14
                    }   }       }   }   }       }
                []  P2.receive(M2) {                                    // L5
                        if (x < 5 ) {                                   // IF
                            alt {                                       // ALT
                                []  P2.receive(M4) {                    // L6
                                        setverdict(pass);               // L7
                                        x := 7 + 5;                     // L9
                                        alt {                           // ALT
                                            []  P1.receive(M3) {        // L2
                                                    setverdict(pass);   // L3
                                                }
                                            []  T1.timeout { }          // L4
                                    }   }
                                []  Comp1.done {                        // L8
                                        x := 7 + 5;                     // L9
                                        alt {                           // ALT
                                            []  P1.receive(M3) {        // L2
                                                    setverdict(pass);   // L3
                                                }
                                            []  T1.timeout { }          // L4
                                    }   }
                                []  P1.receive(M3) {                    // L2
                                        setverdict(pass);               // L3
                                        alt {                           // ALT
                                            []  P2.receive(M4) {        // L6
                                                    setverdict(pass);   // L7
                                                    x := 7 + 5;         // L9
                                                }
                                            []  Comp1.done {            // L8
                                                    x := 7 + 5;         // L9
                                    }   }       }
                                []  T1.timeout {                        // L4
                                        alt {                           // ALT
```

```
                                        []  P2.receive(M4) {                        // L6
                                                setverdict(pass);                   // L7
                                                x := 7 + 5;                         // L9
                                            }
                                        []  Comp1.done {                            // L8
                                                x := 7 + 5;                         // L9
                        }   }       }   }       }
                        else {
                            P3.call(MyProcTempl, 20E-3) {           // BC (= BLOCKING CALL)
                                []  P3.getreply(ReplyTempl) {                       // L10
                                        alt {                                       // 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
                                                    []  P2.receive(M6) { }          // L12
                                            }   }
                                        []  P2.receive(M5) {                        // L11
                                                alt {                               // ALT
                                                    []  P1.receive(M3) {            // L2
                                                            setverdict(pass);       // L3
                                                        }
                                                    []  T1.timeout { }              // L4
                                            }   }
                                        []  P2.receive(M6) {                        // L12
                                                alt {                               // ALT
                                                    []  P1.receive(M3) {            // L2
                                                            setverdict(pass);       // L3
                                                        }
                                                    []  T1.timeout { }              // L4
                                    }   }       }   }
                                []  P3.catch(timeout) {                             // L13
                                        setverdict(fail);                           // L14
                                        alt {                                       // ALT
                                        []  P1.receive(M3) {                        // L2
                                                setverdict(pass);                   // L3
                                            }
                                        []  T1.timeout { }                          // L4
    }   }       }   }   }       }   }
[]  P2.receive(M2) {                                                                // L5
        if (x < 5 ) {                                                               // IF
            alt {                                                                   // ALT
                []  P2.receive(M4) {                                                // L6
                        setverdict(pass);                                           // L7
                        x := 7 + 5;                                                 // L9
                        alt {                                                       // ALT
                            []  P1.receive(M1} {                                    // L1
                                    alt {                                           // ALT
                                    []  P1.receive(M3) {                            // L2
                                            setverdict(pass);                       // L3
                                        }
                                    []  T1.timeout { }                              // L4
                        }   }       }   }
                []  Comp1.done {                                                    // L8
                        x := 7 + 5;                                                 // L9
                        alt {                                                       // ALT
                            []  P1.receive(M1} {                                    // L1
                                    alt {                                           // ALT
                                    []  P1.receive(M3) {                            // L2
                                            setverdict(pass);                       // L3
                                        }
                                    []  T1.timeout { }                              // L4
                        }   }       }   }
                []  P1.receive(M3) {                                                // L2
                        setverdict(pass);                                           // L3
                        alt {                                                       // ALT
                            []  P2.receive(M4) {                                    // L6
                                    setverdict(pass);                               // L7
                                    x := 7 + 5;                                     // L9
                                }
                            []  Comp1.done {                                        // L8
                                    x := 7 + 5;                                     // L9
                        }   }       }
                []  T1.timeout {                                                    // L4
                        alt {                                                       // ALT
```

```
                                    []  P2.receive(M4) {                                    // L6
                                            setverdict(pass);                               // L7
                                            x := 7 + 5;                                     // L9
                                        }
                                    []  Comp1.done {                                        // L8
                                            x := 7 + 5;                                     // L9
            }   }       }   }       }
            else {
                P3.call(MyProcTempl, 20E-3) {                       // BC (= BLOCKING CALL)
                    []  P3.getreply(ReplyTempl) {                                           // L10
                            alt {                                                           // ALT
                                    []  P2.receive(M5) {                                    // L11
                                            alt {                                           // ALT
                                                    []  P1.receive(M1} {                    // L1
                                                            alt {                           // ALT
                                                                    []  P1.receive(M3) {    // L2
                                                                            setverdict(pass);// L3
                                                                    }
                                                                    []  T1.timeout { }      // L4
                                            }   }       }   }
                                    []  P2.receive(M6) {                                    // L12
                                            alt {                                           // ALT
                                                    []  P1.receive(M1} {                    // L1
                                                            alt {                           // ALT
                                                                    []  P1.receive(M3) {    // L2
                                                                            setverdict(pass);// L3
                                                                    }
                                                                    []  T1.timeout { }      // L4
                                            }   }       }   }
                                    []  P1.receive(M1} {                                    // L1
                                            alt {                                           // 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
                                                                    []  P2.receive(M6) { }  // L12
                                                            }   }
                                                    []  P2.receive(M5) {                    // L11
                                                            alt {                           // ALT
                                                                    []  P1.receive(M3) {    // L2
                                                                            setverdict(pass);// L3
                                                                    }
                                                                    []  T1.timeout { }      // L4
                                                            }   }
                                                    []  P2.receive(M6) {                    // L12
                                                            alt {                           // ALT
                                                                    []  P1.receive(M3) {    // L2
                                                                            setverdict(pass);// L3
                                                                    }
                                                                    []  T1.timeout { }      // L4
                                            }   }       }   }       }   }
                    []  P3.catch(timeout) {                                                 // L13
                            setverdict(fail);                                               // L14
                            alt {                                                           // ALT
                                    []  P1.receive(M1} {                                    // L1
                                            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 to one or several other values. Therefore, semantics of a **`select-case`** statement can be described by its replacement with a set of nested **`if-else`** statements. The operational semantics assumes that this replacement is done on the syntactical level.

Schematically the **`select-case`** statement looks as follows:

```
select (<expression>) {
    case (<templateInst1a>, …, <templateInst1n>)
        <statementblock1>
    case (<templateInst2a>, …, <templateInst2n>)
        <statementblock2>
     …
    case (<templateInstxa>, …, <templateInstxn>)
        <statementblockx>
    case else
            <statementblockx+1>
    }
```

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

```
if (match(<expression>, <templateInst1a>) or … or match(<expression>, <templateInst1n>))
    <statementblock1>
else {
    if (match(<expression>, <templateInst2a>) or … or match(<expression>, <templateInst2n>))
        <statementblock2>
    else {
        …
        if (match(<expression>, <templateInstxa>) or … or match(<expression>, <templateInstxn>))
            <statementblockx>
        else
            <statementblockx+1>
        …
    }
}
```

Example:

```
    // The following select-case statement:

    select (MyModulePar) { // where MyModulePar is of charstring type
        case ("firstValue") {
                log ("The first branch is selected");
            }
        case (MyCharstingVar, MyCharstringConst) {
                log ("The second branch is selected");
            }
        case else {
                log ("The else branch is selected");
            }
    }

    // is semantically equivalent to:

    if (match(MyModulePar, "firstValue")) {
        log ("The first branch is selected");
    }
    else {
        if (match(MyModulePar, MyCharstingVar) or match(MyModulePar, MyCharstingConst)) {
            log ("The second branch is selected");
        }
        else {
                log ("The else branch is selected");
        }
    }
```

8 Flow graph semantics of TTCN-3

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

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.

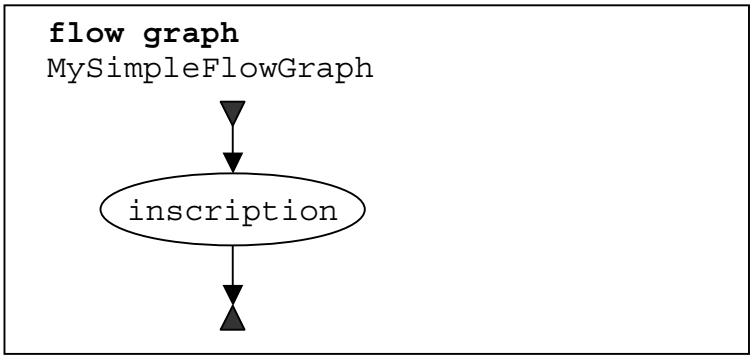


Figure 10 – A simple flow graph

8.1.2 Flow graph nodes

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.

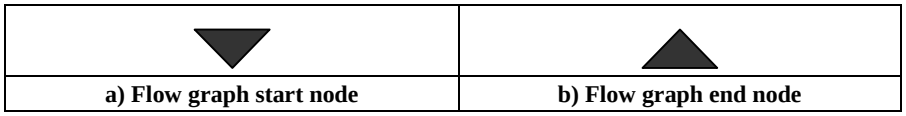


Figure 11 – Start and end nodes

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.

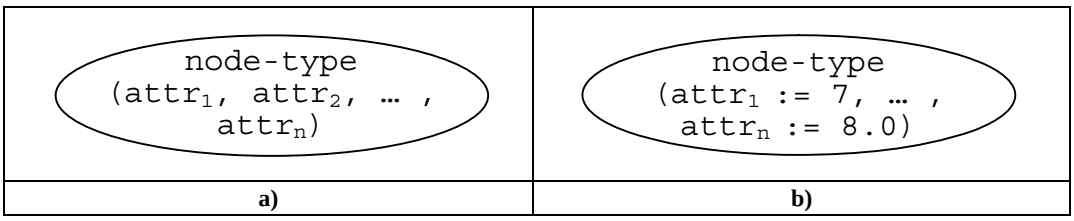


Figure 12 – Basic nodes with attributes

8.1.2.4 Reference nodes

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.

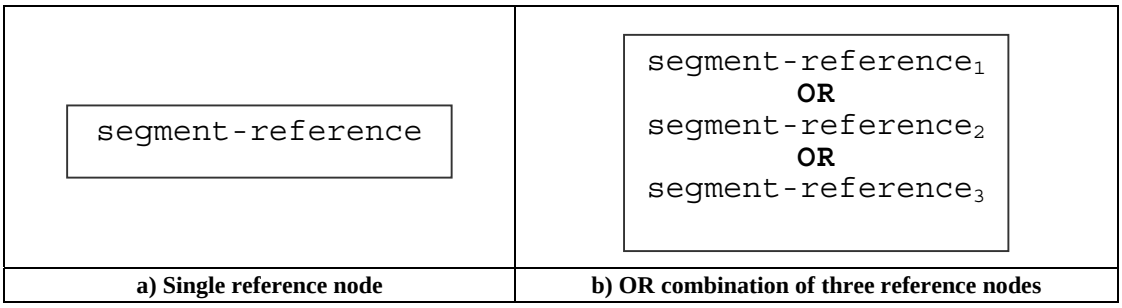


Figure 13 – Reference node

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).

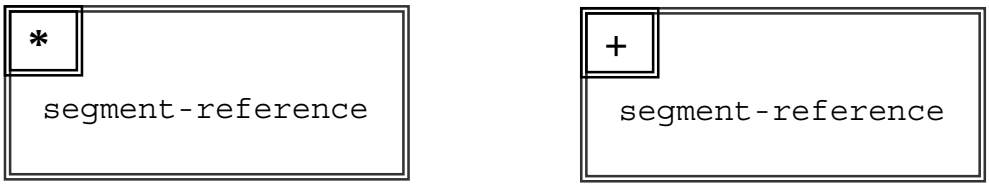


Figure 14 – Repetition of reference nodes

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.

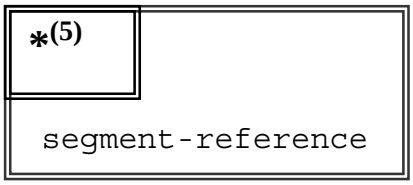
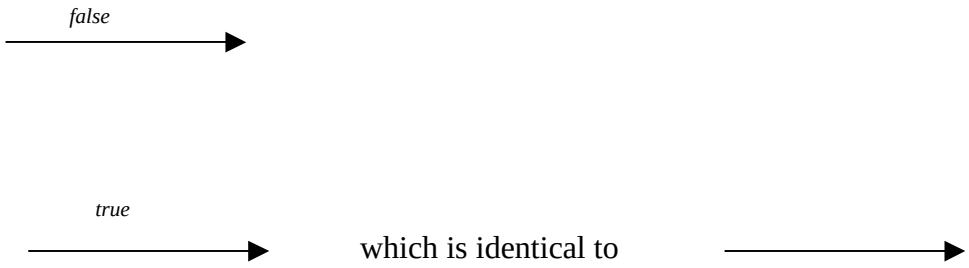


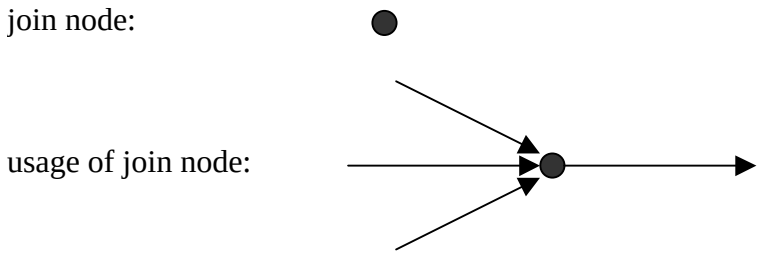
Figure 15 – Restricted repetition of a reference node

8.1.3 Flow lines

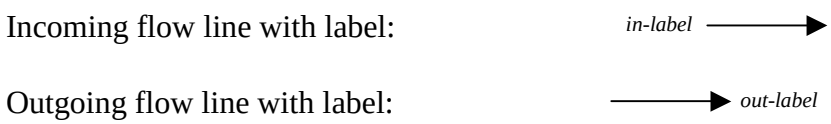
Flow lines are represented by means of arrows. A flow line has an inscription of *true* or *false* which indicates a condition under which the flow line is chosen during the flow graph interpretation. As a short hand notation it is allowed to omit the true inscription. Examples of flow lines are shown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



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 below:



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 below:



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 unlabelled incoming and one or none unlabelled 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.

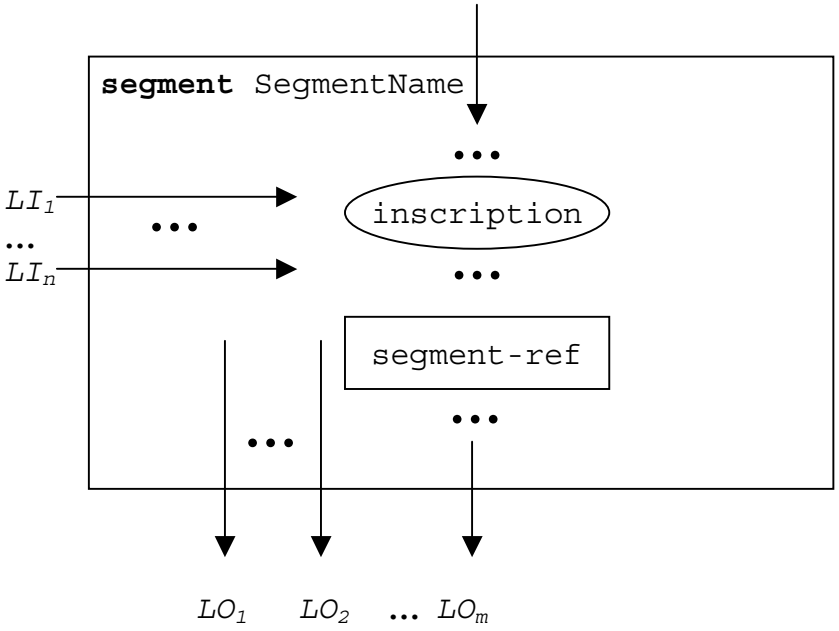


Figure 16 – Structure of a flow graph segment description

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.

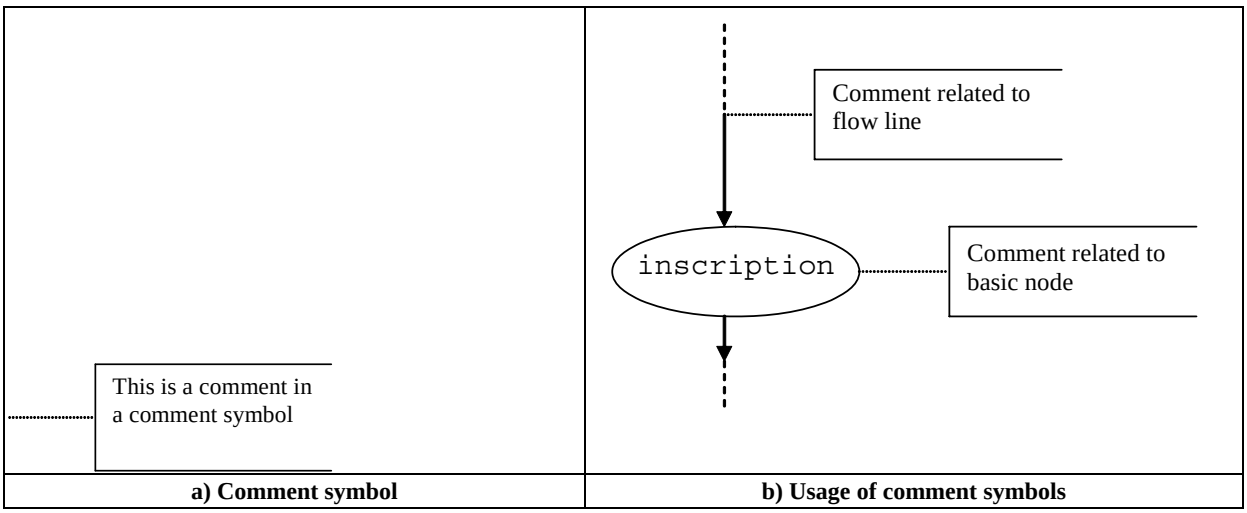


Figure 17 – Flow graph representation of comments

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:

actualNodeRef.*NEXT*(*bool*) := `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

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 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 this Recommendation 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:

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

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

```
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.

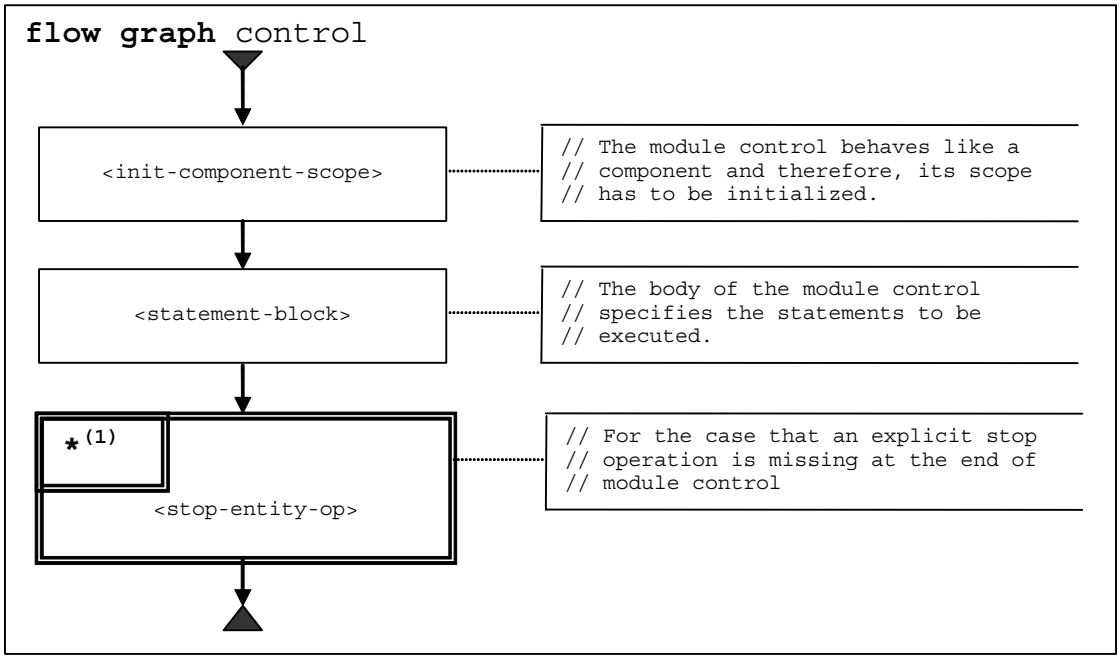


Figure 18 – Flow graph representation of module control

8.2.3 Flow graph representation of test cases

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

```
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 `<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 `<stop-entity-op>` covers the case where no explicit **`stop`** operation for the MTC is specified, i.e., the operational semantics assumes that a **`stop`** operation is implicitly added.

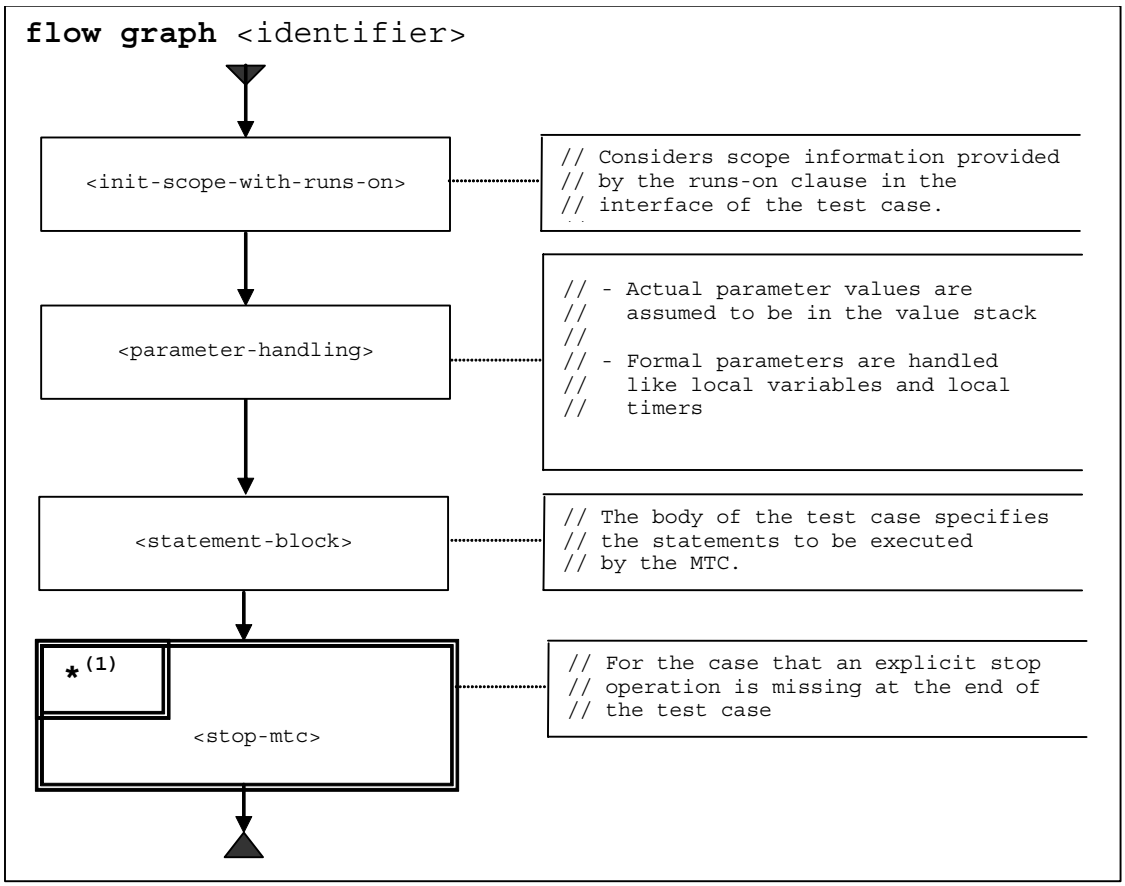


Figure 19 – Flow graph representation of test cases

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 `<function-interface>` above refers to the (optional) **`runs on`** and the (optional) **`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 `<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 **`runs on`** clause in the `<function-interface>`. A missing **`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 `<init-scope-with-runs-on>` and `<init-scope-without-runs-on>`. The reference node `<return-without-value>` covers the case where no explicit **`return`** statement is specified, i.e., the operational semantics assumes that a **`return`** statement is implicitly added.

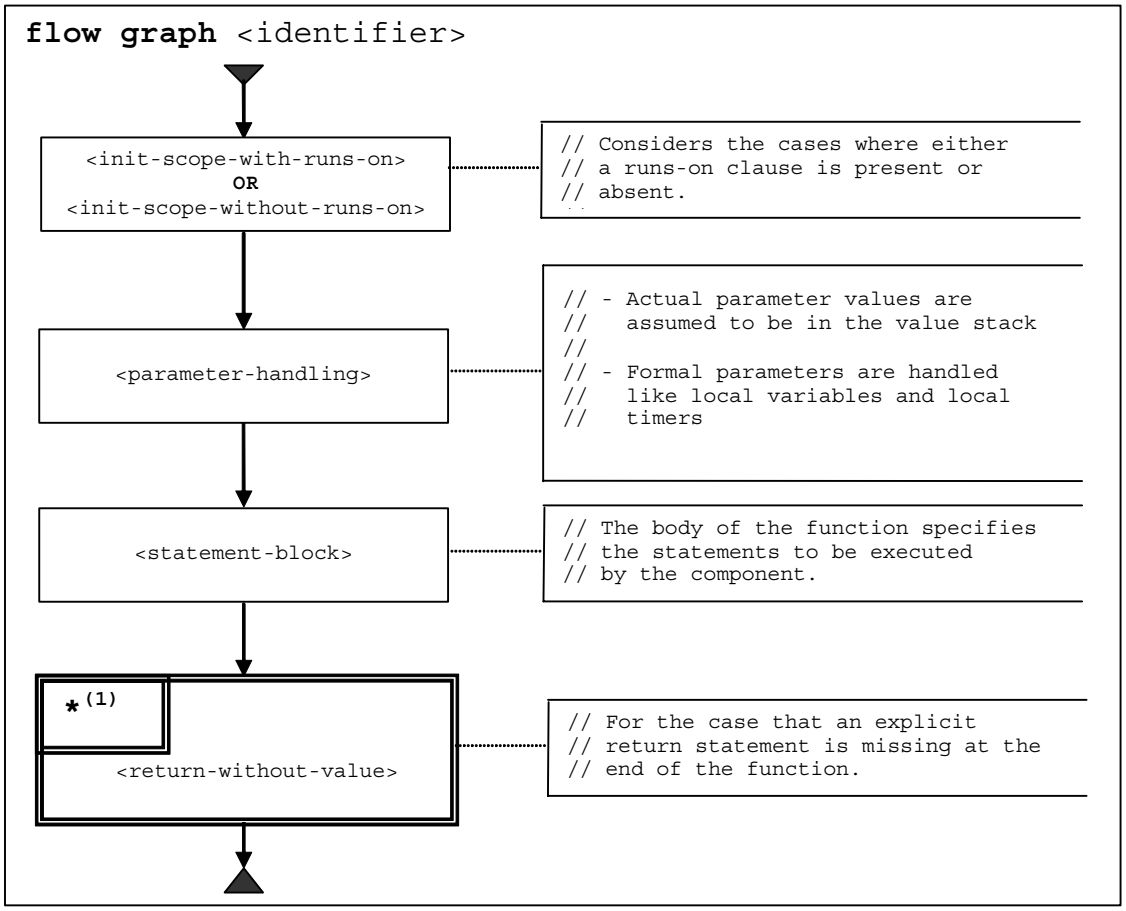


Figure 20 – Flow graph representation of functions

8.2.5 Flow graph representation of altsteps

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

```
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.

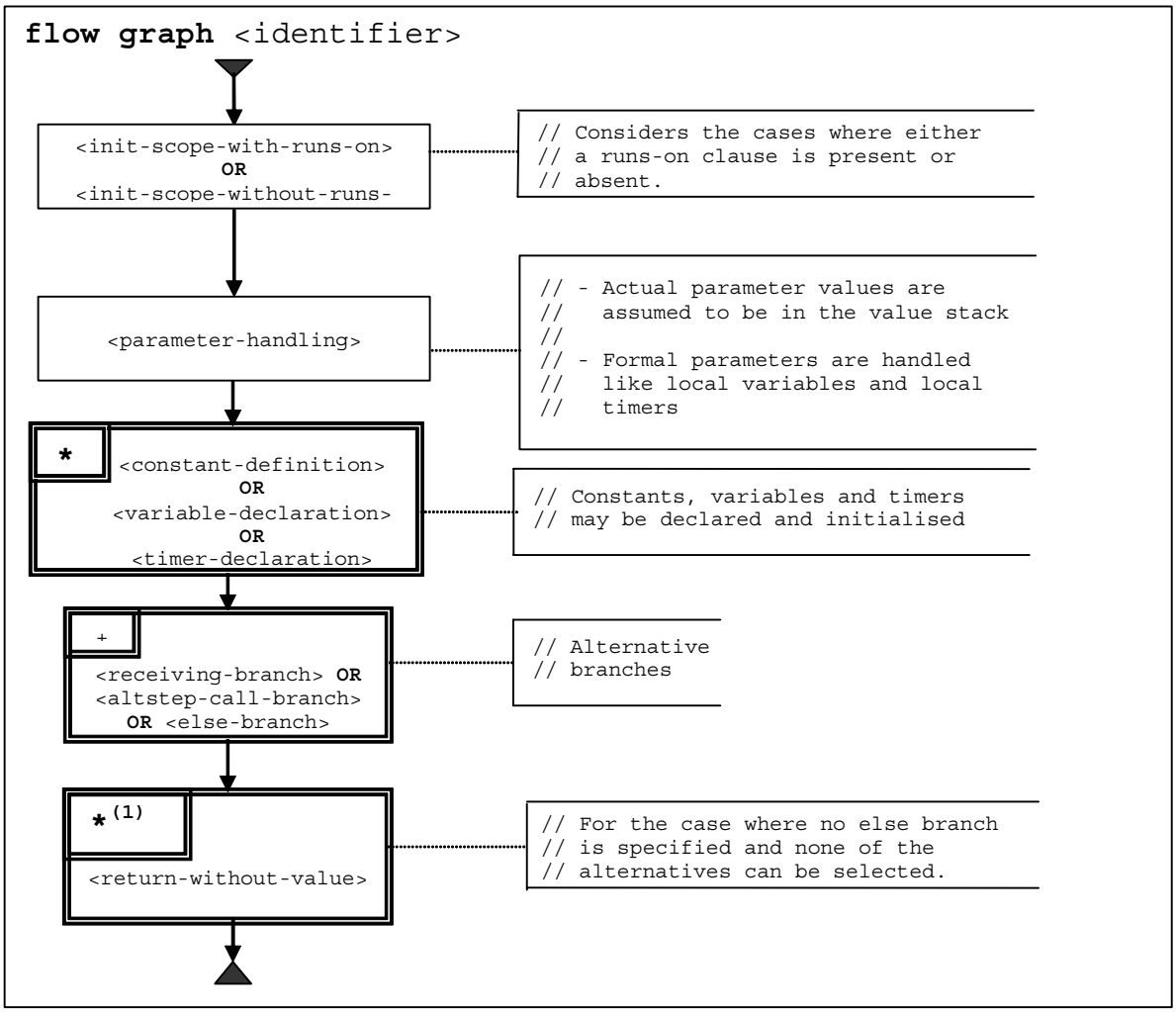


Figure 21 – Flow graph representation of altsteps

8.2.6 Flow graph representation of component type definitions

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

```
type component <identifier> <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 `<identifier>` refers to the name of the represented component type.

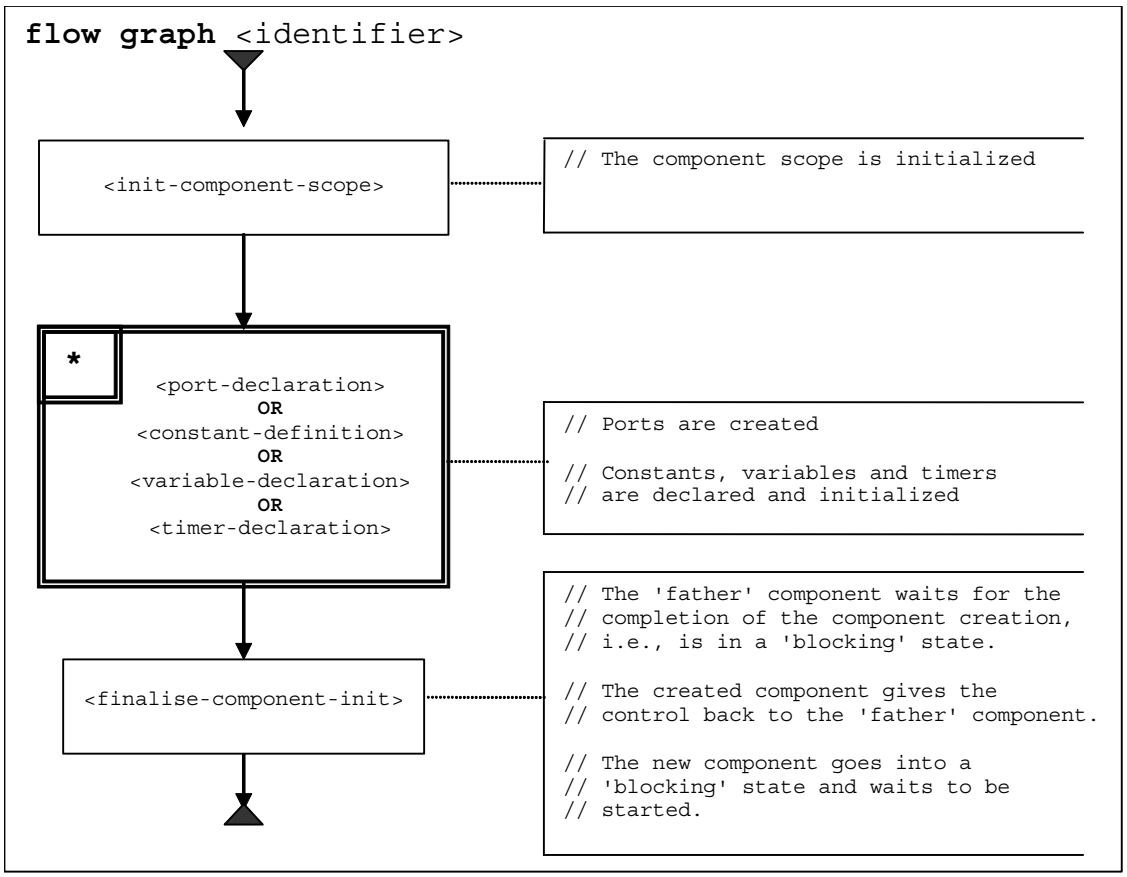


Figure 22 – Flow graph representation of component type definitions

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 and to component type definitions.

8.3 State definitions for TTCN-3 modules

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 test components and ports. Module 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

As shown in Figure 23 a module state is structured into *ALL-ENTITY-STATES*, *ALL-PORT-STATES*, *MTC*, *TC-VERDICT*, *DONE* and *KILLED*. *ALL-ENTITY-STATES* describes the state of the module control and during the execution of a test case the states of the instantiated test components. *ALL-PORT-STATES*, the *MTC* reference and the *TC-VERDICT* are only relevant during test case execution. *ALL-PORT-STATES* describes the states of the different ports. *MTC* provides a reference to the Main Test Component (MTC), *TC-VERDICT* stores the actual global test verdict of a test case, *DONE* is a list of all currently stopped test components during test case execution and *KILLED* is a list of all terminated test components during test case execution.

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 *KILLED* and the *DONE* list when it is killed.

The behaviour of module control (*M-CONTROL* in Figure 23) is handled like a normal test component and its state is the first element in *ALL-ENTITY-STATES* of a module state.

ALL-ENTITY-STATES					ALL-PORT-STATES					MTC	TC-VERDICT	DONE	KILLED
	M-CONTROL	ES_1	…	ES_n		P_1	…	P_n					

Figure 23 – Structure of a module state

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 a module state.

8.3.1.1 Accessing the module state

The *MTC* and *TC-VERDICT* are parts of a module state and handled like global variables, i.e., the keywords *MTC* and *TC-VERDICT* can be used to retrieve and to change the values of the corresponding module state.

NOTE 1 – There only exists one module state during the interpretation of a TTCN-3 module. Therefore, the keywords *MTC* and *TC-VERDICT* can be considered as globally unique identifiers for the evaluation procedure.

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 2 – 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.

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

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

<identifier>
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 <*identifier*> is a unique identifier of an entity, i.e., module control of test component, in the test system. Such unique identifiers are created implicitly for the module control, the **`mtc`** and the test **`system`** when a module starts execution or a test case is executed by means of the **`execute`** statement. The identifier is used to identify and address entities in the test system, e.g., in case of **`send`** operations with **`to`** clauses or **`receive`** operations with **`from`** clauses.

The *STATUS* describes whether the module control or a test component is **`ACTIVE`**, **`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 *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 we define the special value **`MARK`** to be part 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-KILLED*, i.e., *SNAP-KILLED* 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 <*identifier*> is the unique identifier of an entity state, which can be used to access the component represented by entity state and the different parts of the entity state.

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, we assume that we can use the "dot" notation by using references and unique identifiers. 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*, *last*, *length*, *next*, *random* and *change*. The meaning of these list operations is defined in clause 8.3.1.1.

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

- *NEW-ENTITY* (*entityIdentifier*, *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:

- <identifier> is set to *entityIdentifier* and shall be a globally unique identifier;
- *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 *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 *CONTROL-STACK*. This series of stack operations is encapsulated in the *NEXT-CONTROL* function:

```
myEntity.NEXT-CONTROL(myBool) {
     successorNode := myEntity.CONTROL-STACK.NEXT(myBool).top();
     myEntity.CONTROL-STACK.pop();
     myEntity.CONTROL-STACK.push(successorNode);
}
```

8.3.2.2 Data state and variable binding

As shown in Figure 25, the data state *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 **`return`** statement.

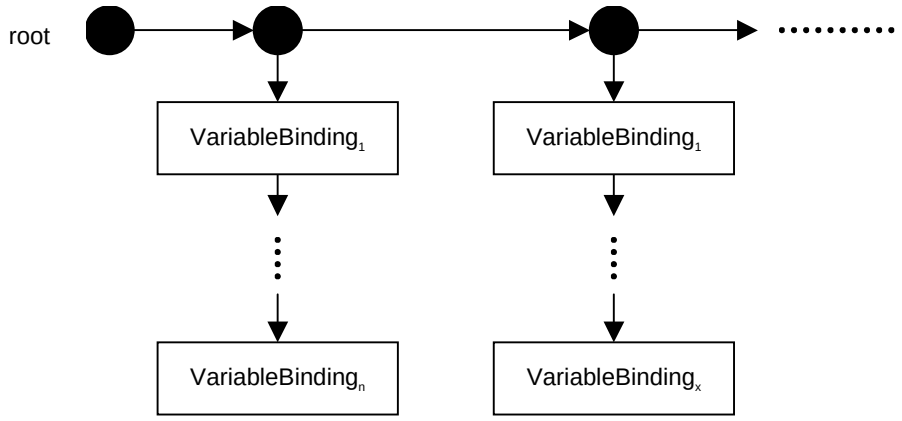


Figure 25 – Structure of the *DATA-STATE* part of an entity state

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

NOTE – Unique location identifiers shall be provided automatically when a variable is declared.

VAR-NAME	<location>	VALUE

Figure 26 – Structure of a variable binding

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:

a) 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 *VAR-NAME*, receives a new location and gets the value that is passed into the function or test case.

b) 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 *VAR-NAME*, but receives no new location and no new value. The new variable binding gets a copy of <*location*> and *VALUE* of the variable that is passed in by reference.

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 my*Val* 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.

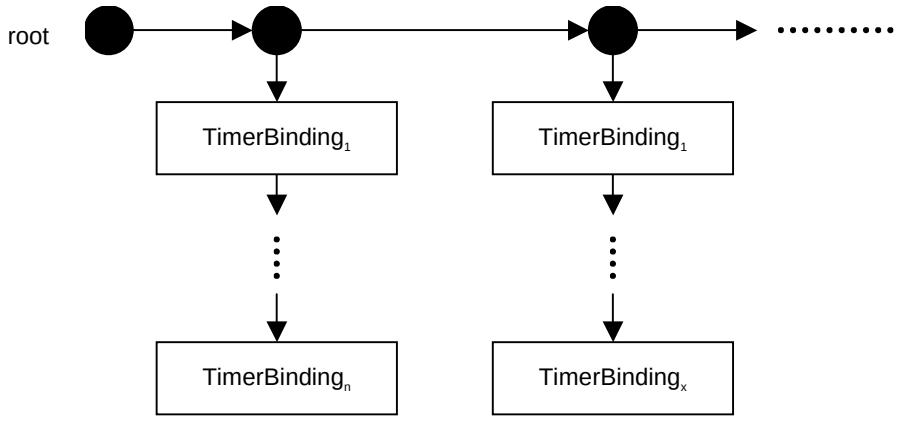


Figure 27 – Structure of the *TIMER-STATE* part of an entity state

The structure of a timer binding is shown in Figure 28. The meaning of *TIMER-NAME* and <*location*> is similar to the meaning of *VAR-NAME* and <*location*> for a variable binding (Figure 26).

TIMER-NAME	<location>	STATUS	DEF-DURATION	ACT-DURATION	TIME-LEFT	SNAP-VALUE	SNAP-STATUS

Figure 28 – Structure of a timer binding

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

NOTE – *DEF-DURATION* is undefined if a timer is declared without default duration. *ACT-DURATION* and *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 *DEF-DURATION* is copied into *ACT-DURATION*. A dynamic error occurs if a timer is started without a defined duration.

SNAP-VALUE and *SNAP-STATUS* are needed to support the snapshot semantics of TTCN-3. When taking a snapshot, *SNAP-VALUE* gets the actual value of *ACT-DURATION* – *TIME-LEFT*. And *SNAP-STATUS* gets the same value as *STATUS*. The evaluation of a snapshot will only be based on the values in *SNAP-VALUE* and *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 <*location*>, *STATUS*, *DEF-DURATION*, *ACT-DURATION*, *TIME-LEFT*, *SNAP-VALUE* and *SNAP-STATUS* from the timer that is passed in by reference. When updating a timer all timer bindings with the same <*location*> value are updated at the same time.

8.3.2.5 Accessing timer states

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

- myEntity.myTimer.STATUS;
- myEntity.myTimer.DEF-DURATION;
- myEntity.myTimer.ACT-DURATION;

- myEntity.myTimer.TIME-LEFT;
- myEntity.myTimer.SNAP-VALUE;
- myEntity.myTimer.SNAP-STATUS.

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

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

- *myEntity.TIMER-SET(myTimer, STATUS, myVal).*

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

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

a) The *INIT-TIMER* function: *myEntity.INIT-TIMER* (*myTimer*, *myDuration*)

creates a new timer binding for a timer *myTimer* with the default duration *myDuration* in the actual scope of an entity *myEntity*. Using the keyword **NONE** as *myDuration* means that a timer without default duration is created.

b) The *GET-TIMER-LOC* function: *myEntity.GET-TIMER-LOC* (*myTimer*)

retrieves the location of timer *myTimer* owned by *myEntity*.

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

creates a new timer binding for a timer *myTimer* with the location *myLocation* in the actual scope unit of *myEntity*. The timer will be initialized with the values of *STATUS*, *DEF-DURATION*, *ACT-DURATION* and *TIME-LEFT* of another timer with the location *<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 8.3.2.4 all timers with the same location will have identical values for *STATUS*, *DEF-DURATION*, *ACT-DURATION* and *TIME-LEFT* during their lifetime.

d) The *INIT-TIMER-SCOPE* function: *myEntity.INIT-TIMER-SCOPE* ()

initializes a new timer scope in the timer state of entity *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: *myEntity.DEL-TIMER-SCOPE* ()

deletes a timer scope of the timer state of entity *myEntity*, i.e., the first list in the list of lists of timer bindings is deleted.

f) The *SNAP-TIMER* function: *myEntity.SNAP-TIMER* ()

makes an update of *SNAP-VALUE* and *SNAP-STATUS*, in all timers owned by *myEntity*, i.e.:

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

8.3.2.6 Port references and port binding

As shown in Figures 28-a, 27 and 25 the port references *PORT-REF*, the timer state *TIMER-STATE* and the data state *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.

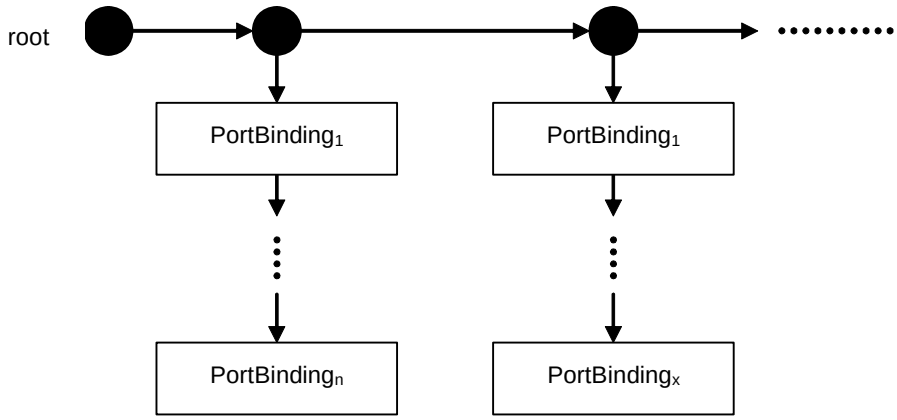


Figure 28-a – Structure of the *PORT-REF* part of an entity state

The structure of a port binding is shown in Figure 28-b. A port has two names. *PORT-NAME* identifies a port in a scope unit. *COMP-PORT-NAME* is the port name given in the component type to a port.

PORT-NAME	COMP-PORT-NAME

Figure 28-b – Structure of a port binding

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 altsteps, 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

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** 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**.

COMP-PORT-NAME	OWNER	STATUS	CONNECTIONS-LIST	VALUE-QUEUE	SNAP-VALUE

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 indicates that the port is mapped onto the port *myPort* in the test system interface.

REMOTE-ENTITY	REMOTE-PORT-NAME

Figure 30 – Structure of a connection

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 exits 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:

GET-PORT (*myEntity*, *myPort*).*STATUS*

e) The *ADD-CON* function: *ADD-CON* (*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: *DEL-CON* (*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: *SNAP-PORTS* (*myEntity*)

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

```
SNAP-PORTS (myEntity) {
    for all ports p         /* in the module state */ {
        if (p.OWNER == myEntity) {
            if (p.STATUS == STOPPED) {
                p. SNAP-VALUE := NULL;
            }
            else {
                p. SNAP-VALUE := p.first()
            }
        }
    }
}
```

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

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 this Recommendation, *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 objects 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.

sender	type	value

Figure 31 – Structure of a message

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 (see clause 8.4.1), 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.

sender	procedure-reference	parameter-part				
			in-or-inout-parameter$_1$	…	in-or-inout-parameter$_n$	

Figure 32 – Structure of a procedure call

sender	procedure-reference	parameter-part					value
			inout-or-out-parameter$_1$	…	inout-or-out-parameter$_n$		

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.

sender	procedure-reference	type	value

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 fulfil 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:

MATCH-ITEM (*myItem*, `<matching-part>`, `<sender>`)

returns **`true`** if *myItem* fulfils 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:

RETRIEVE-INFO (*myItem*, `<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

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 <$parId_1$> … <$parId_m$> for the formal parameters <$parId_1$> … <$parId_m$>. 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.

behaviour-id	value-parameters					variable and timer reference-parameters					port reference-parameters				
		$parId_1$	…	$parId_m$			$parId_n$	…	$parId_r$			$parId_s$	…	$parId_z$	
		$value_1$	…	$value_m$			loc_n	…	loc_r			$name_s$	…	$name_z$	

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*.$parId_n$ or *myCallRecord*.*behaviour-id* where *myCallRecord* is a pointer to a call record.

For the construction of a call, the function *NEW-CALL-RECORD* is assumed to be available:

NEW-CALL-RECORD(*myBehaviour*)

creates a new call record for function, altstep or test case *myBehaviour* and returns a pointer to the new record. The parameter fields of the new call record have undefined values.

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 *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 *COMP-PORT-NAME* field.

8.6 The evaluation procedure for a TTCN-3 module

8.6.1 Evaluation phases

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

1) initialization phase;
2) update phase;
3) selection phase; and
4) 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:

a) *Declaration and initialization of variables*:

– *INIT-FLOW-GRAPHS*(); // Initialization of flow graph handling.
// INIT-FLOW-GRAPHS is explained in clause 8.6.2

– *Entity*:= **NULL**; // *Entity* will be used to refer to an entity state. An entity //state either represents module control or a test //component.

NOTE – The following global variables *ALL-ENTITY-STATES*, *ALL-PORT-STATES*, *MTC*, *TC-VERDICT*, *DONE*, *KILLED* and *SNAP-ACTIVE* form the module state that is manipulated during the interpretation of a TTCN-3 module (see clause 8.3.1).

- *ALL-ENTITY-STATES*:= **NULL**;
- *ALL-PORT-STATES*:= **NULL**;
- *MTC*:= **NULL**;
- *TC-VERDICT*:= **none**;
- *DONE*:= **NULL**;
- *KILLED*:= **NULL**;

b) *Creation and initialization of module control*:

- *Entity:=* *NEW-ENTITY* (*GET-UNIQUE-ID*(),*GET-FLOW-GRAPH* (<*moduleId*>), **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*>. *GET-UNIQUE-ID* is // explained in clause 8.6.2. The Boolean parameter indicates that // module control cannot be restarted after it is stopped.
- *Entity*.*INIT-VAR-SCOPE*(); // New variable scope
- *Entity*.*INIT-TIMER-SCOPE*(); // New timer scope
- *Entity*.*VALUE-STACK*.*push*(**MARK**); // A mark is pushed onto the value stack
- *ALL-ENTITY-STATES*.*append*(*Entity*); // The new entity is put into the module state.

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 *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., *TIME-LEFT* is set to 0.0 and *STATUS* is set to `TIMEOUT`;

NOTE 1 – The update of timers includes the update of all running *TIMER-GUARD* timers in module states. *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.

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 the *STATUS* value **ACTIVE** or **SNAPSHOT**;

b) *Storage:* Store the identifier of the selected entity in the global variable *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 *CONTROL-STACK* of *Entity*;

b) *Check termination criterion:* Stop execution if module control has terminated, i.e., the list of entity states is empty, otherwise continue with Phase II.

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 functions *INIT-FLOW-GRAPHS* and *GET-UNIQUE-ID*:

a) *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.

b) *GET-UNIQUE-ID* is assumed to be a function that returns a unique identifier each time it is called. The unique identifier may be implemented in form of a counter variable that is increased and returned each time *GET-UNIQUE-ID* is called.

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

a) *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.

b) *RETURN*: returns the control back to the module evaluation procedure described in this clause. The *RETURN* is the last action of the '*execution step of the selected entity*' of the *execution* phase.

c) **`***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 **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 <dynamic-error> (clause 9.18.5).

   NOTE – The occurrence of a dynamic error is related to test behaviour. A dynamic error as specified by the operational semantics denotes a problem in the usage of TTCN-3, e.g., wrong usage or race condition.

d) *`APPLY-OPERATOR`* used as generic function for describing the evaluation of operators (e.g., +, *, / or -) in expressions (see clause 9.18.4).

9 Flow graph segments for TTCN-3 constructs

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 **action** statement is:

```
action (<informal description>)
```

The flow graph segment <action-stmt> in Figure 36 defines the execution of the **action** statement.

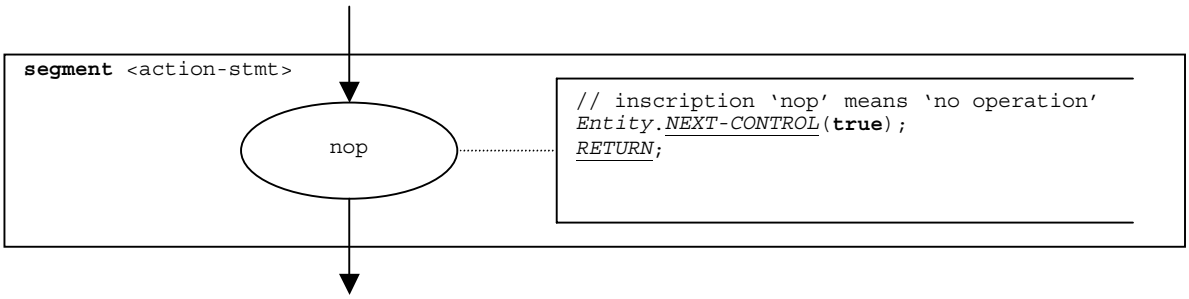


Figure 36 – Flow graph segment <action-stmt>

9.2 Activate statement

The syntactical structure of the **activate** statement is:

```
activate(<altstep-name>([<act-par-desc1>, … , <act-par-descn>]))
```

The <altstep-name> denotes 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`$_1$`>` the corresponding formal parameter identifier `<f-par-Id`$_1$`>` is known, i.e., we can extend the syntactical structure above to:

```
activate(<altstep-name>((<f-par-Id1>,<act-par-desc1>), … , (<f-par-Idn>,<act-par-descn>)))
```

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.

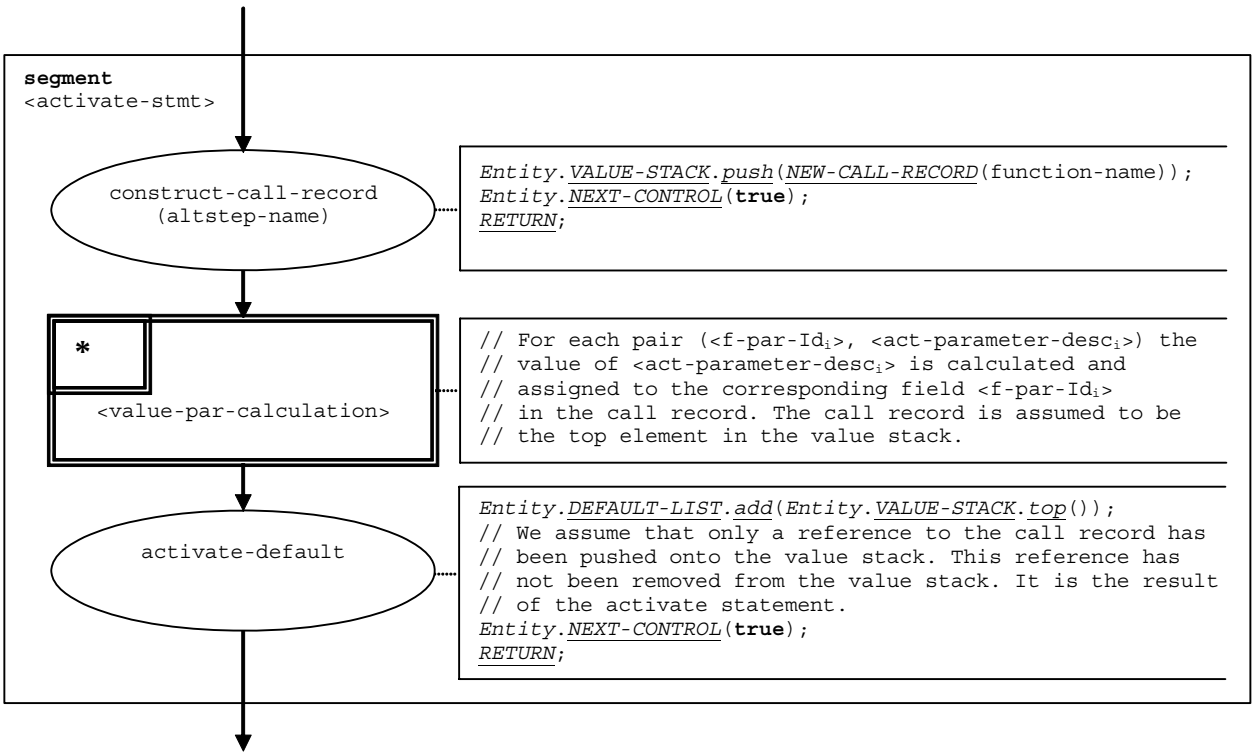


Figure 37 – Flow graph segment <activate-stmt>

9.2a Alive component operation

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 37-a defines the execution of the **running** component operation.

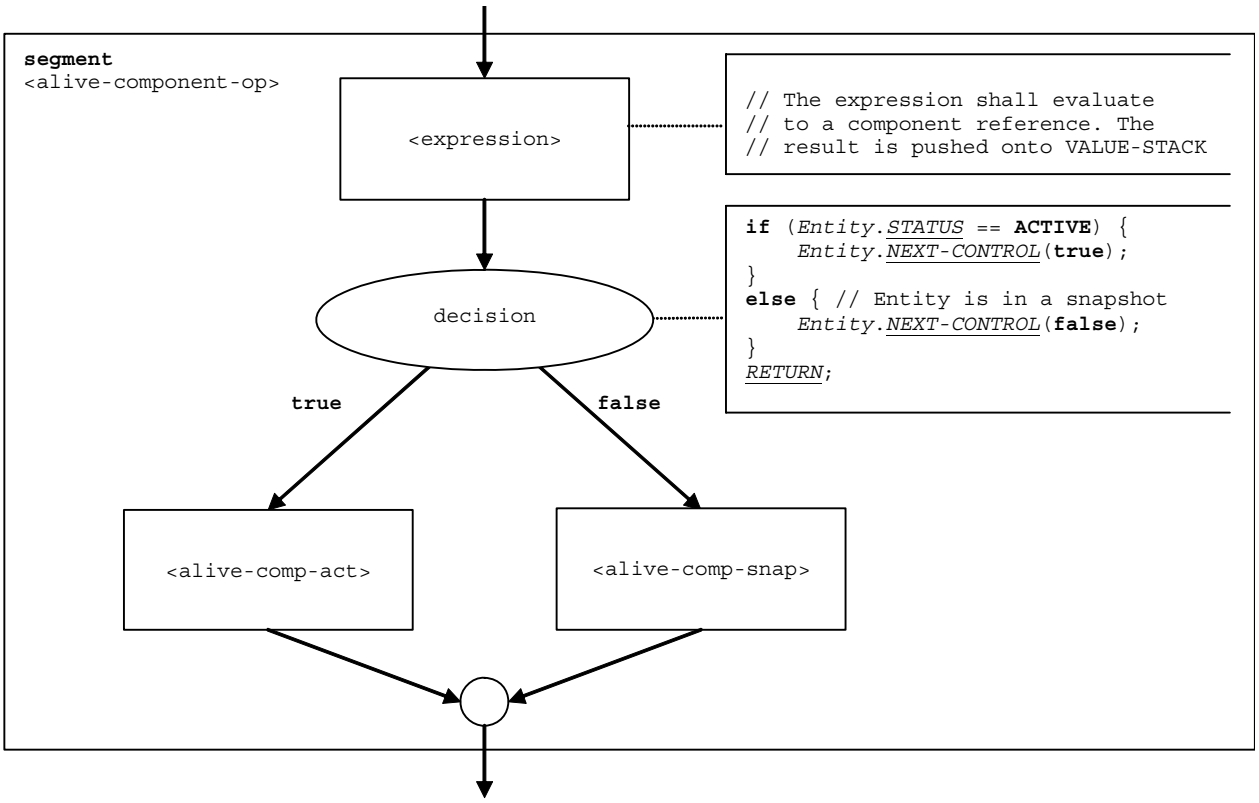


Figure 37-a – Flow graph segment <alive-component-op>

9.2a.1 Flow graph segment <alive-comp-act>

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

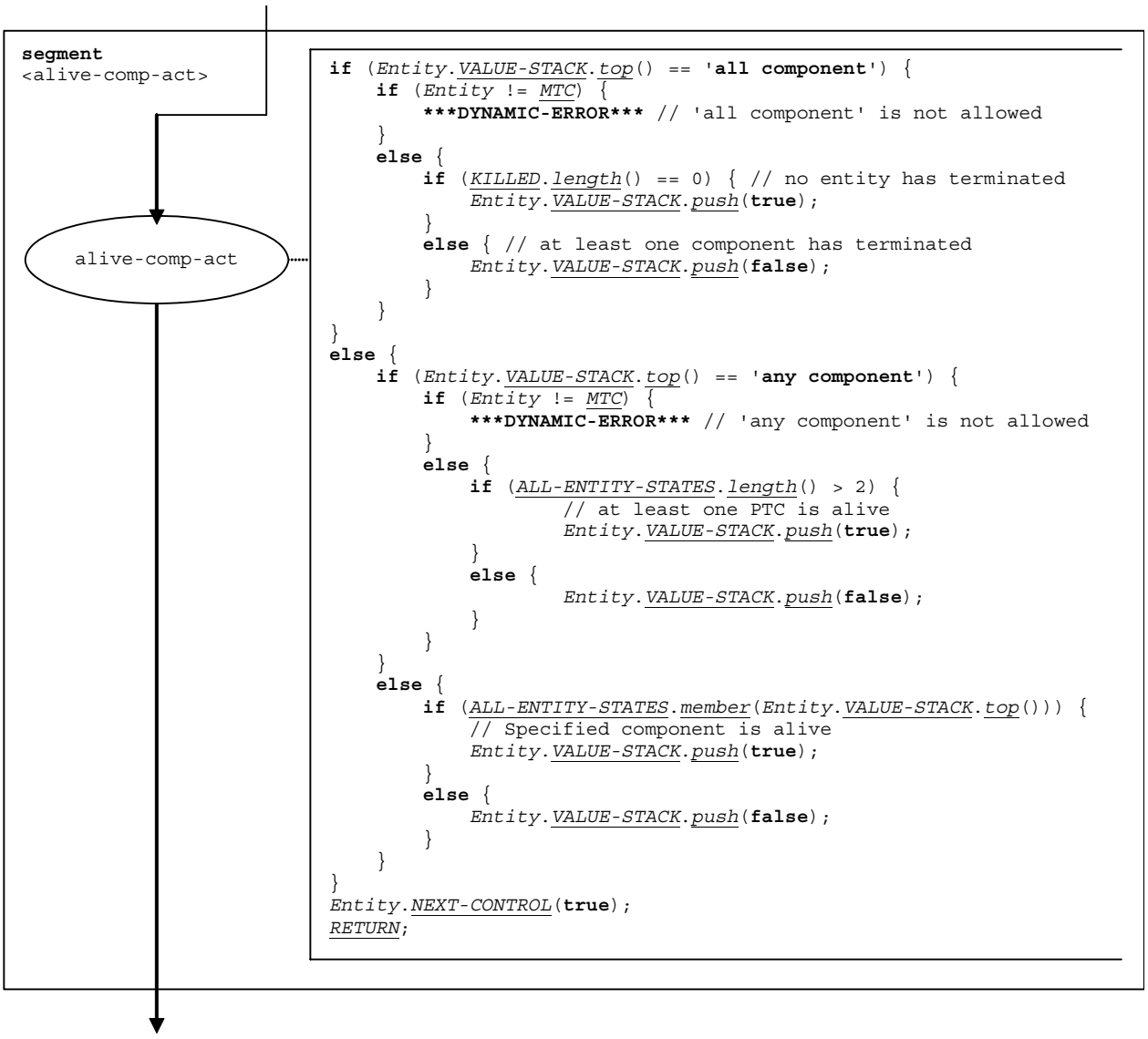


Figure 37-b – Flow graph segment <alive-comp-act>

9.2a.2 Flow graph segment <alive-comp-snap>

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

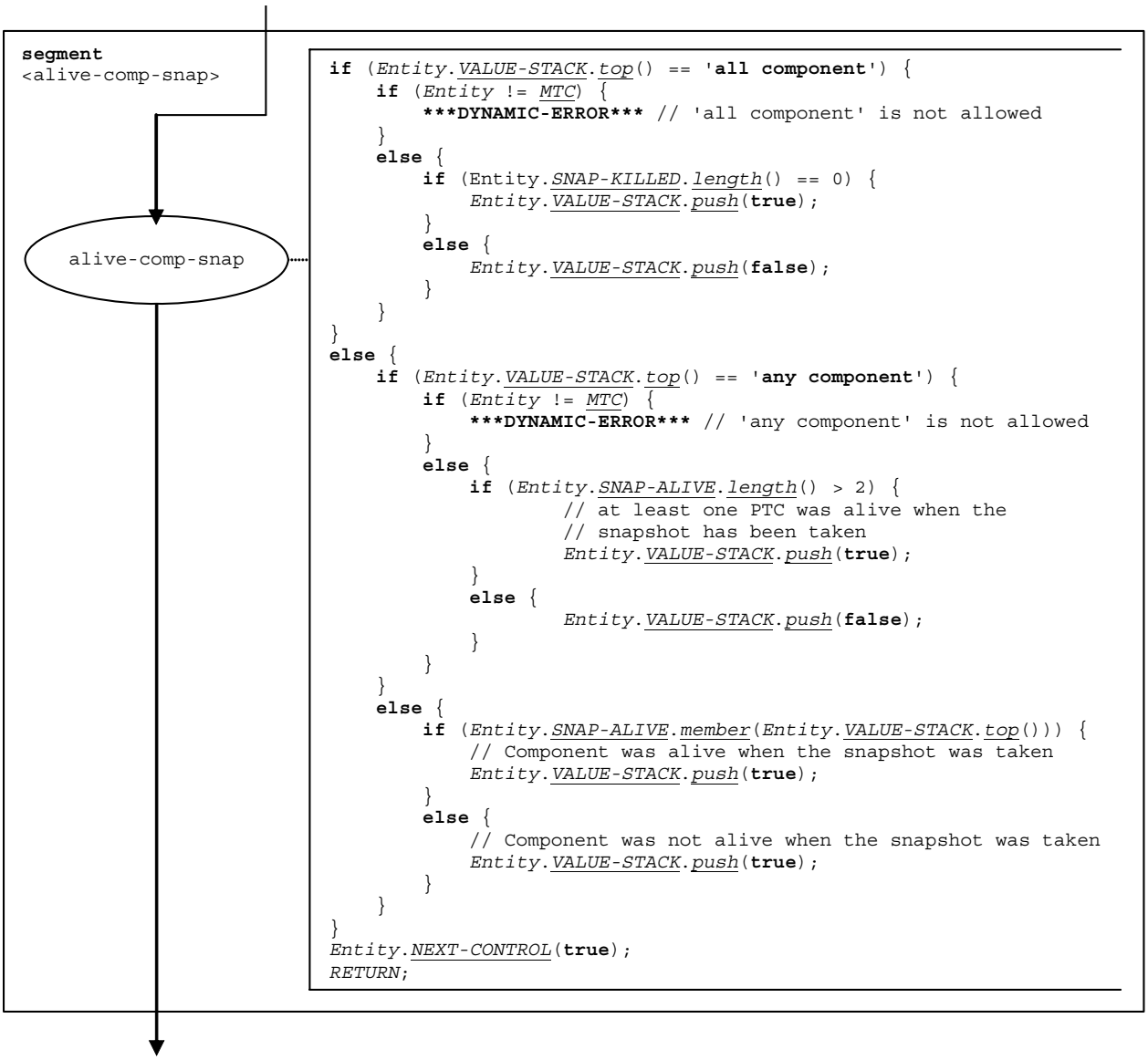


Figure 37-c – Flow graph segment <alive-comp-snap>

9.3 Alt statement

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 is provided 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>`.

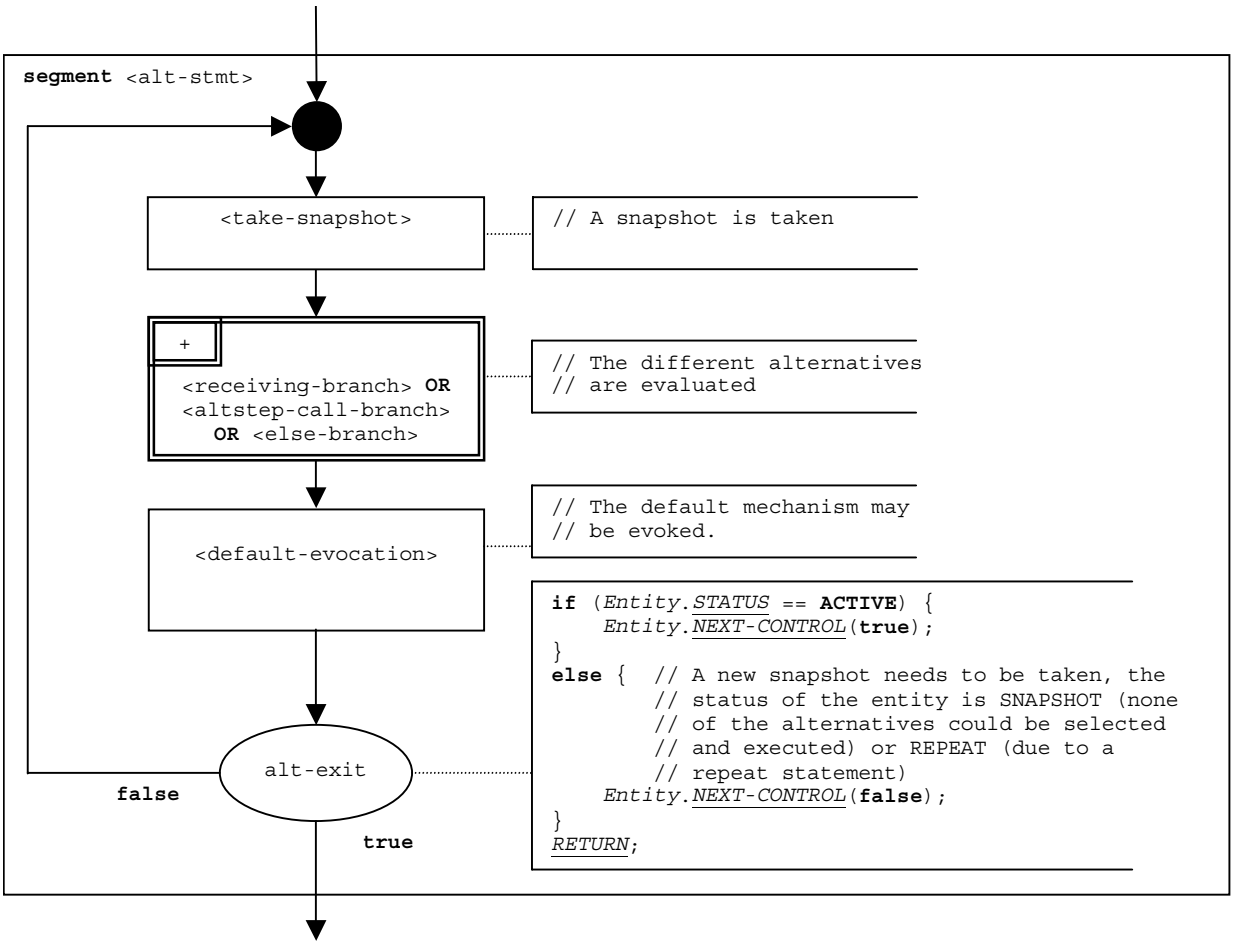


Figure 38 – Flow graph segment <alt-stmt>

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.

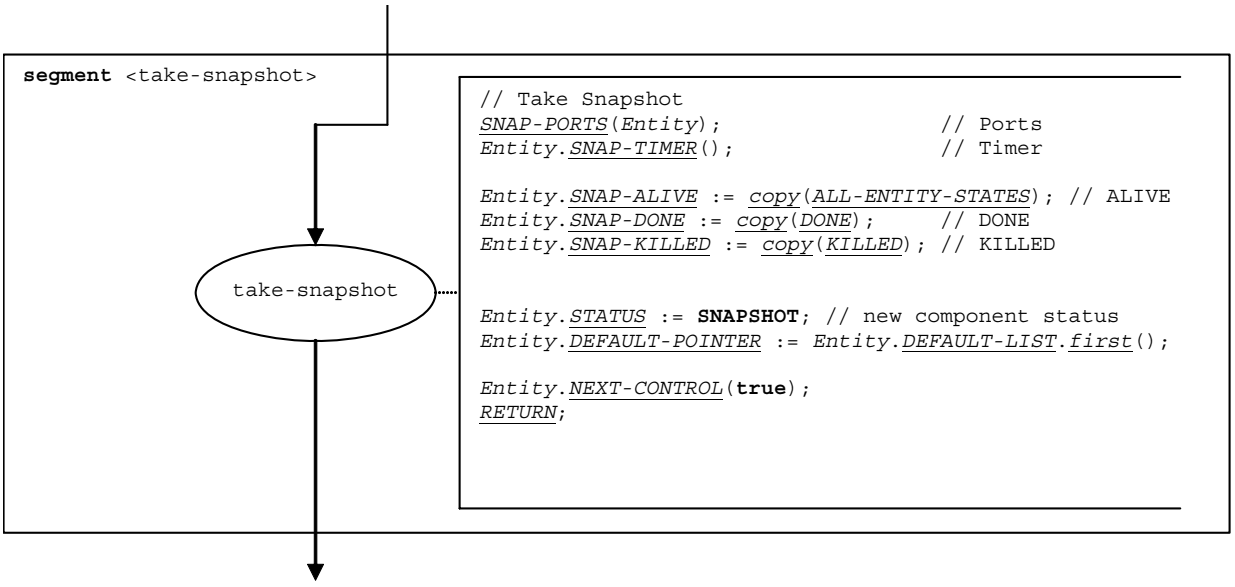


Figure 39 – Flow graph segment <take-snapshot>

9.3.2 Flow graph segment <receiving-branch>

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

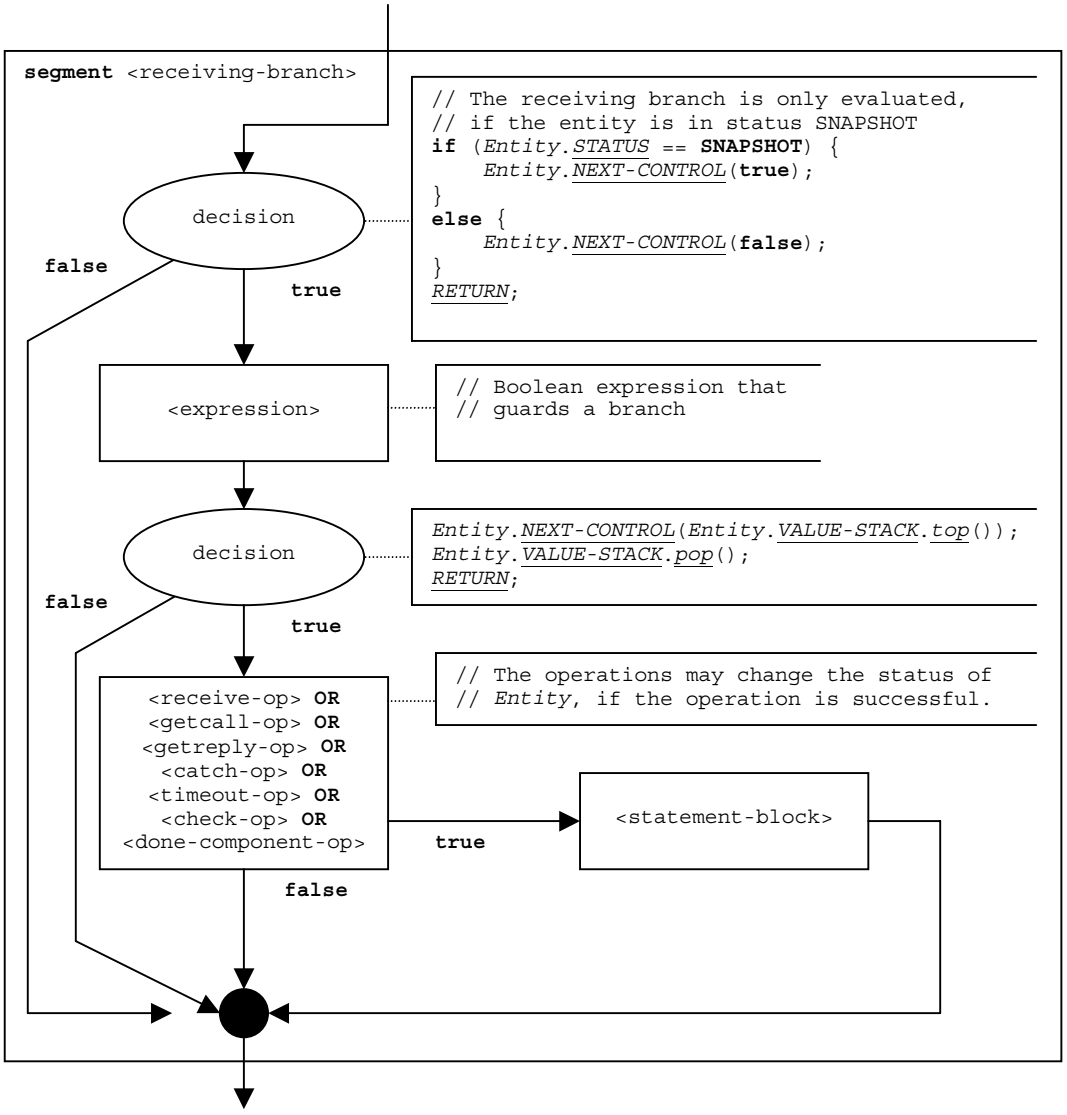


Figure 40 – Flow graph segment <receiving-branch>

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.

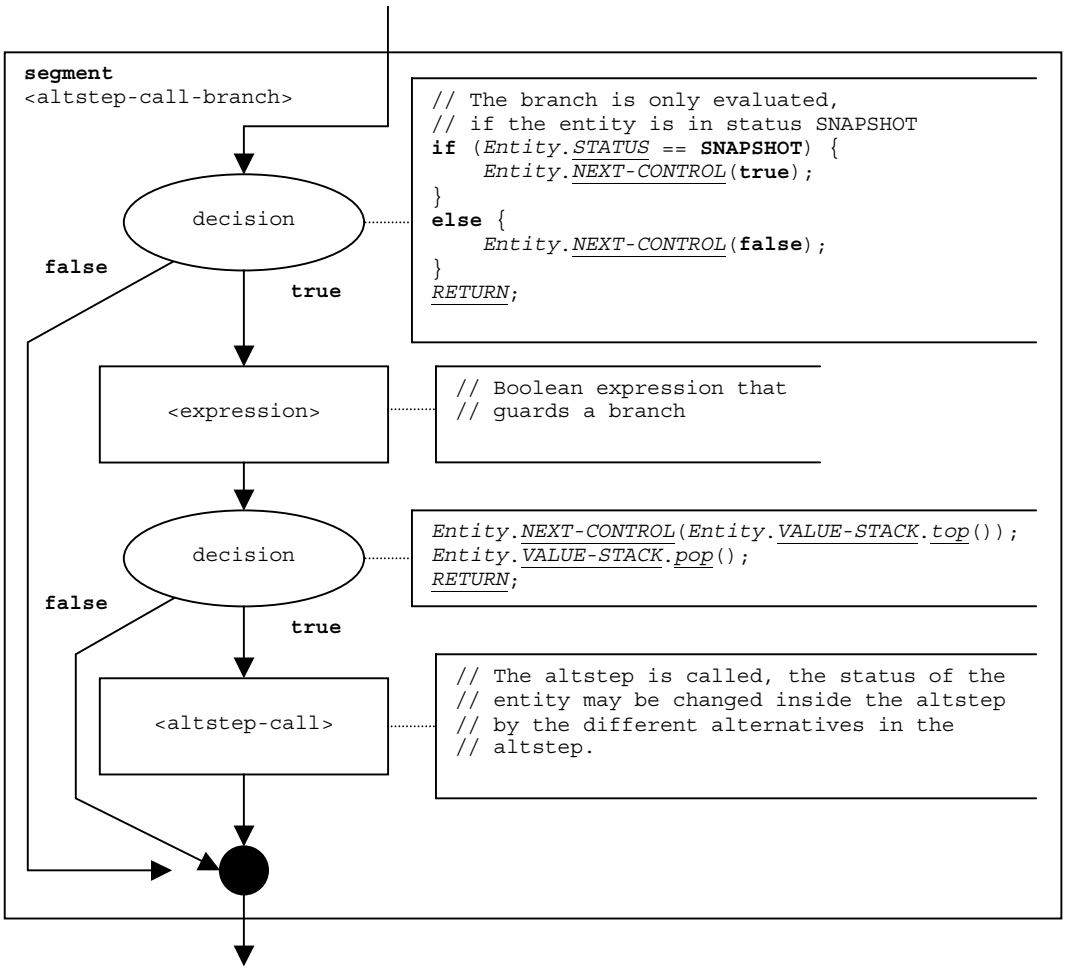


Figure 41 – Flow graph segment <altstep-call-branch>

9.3.4 Flow graph segment <else-branch>

The execution of an **`else`** branch within an **`alt`** statement is described by the flow graph segment `<else-branch>` in Figure 42.

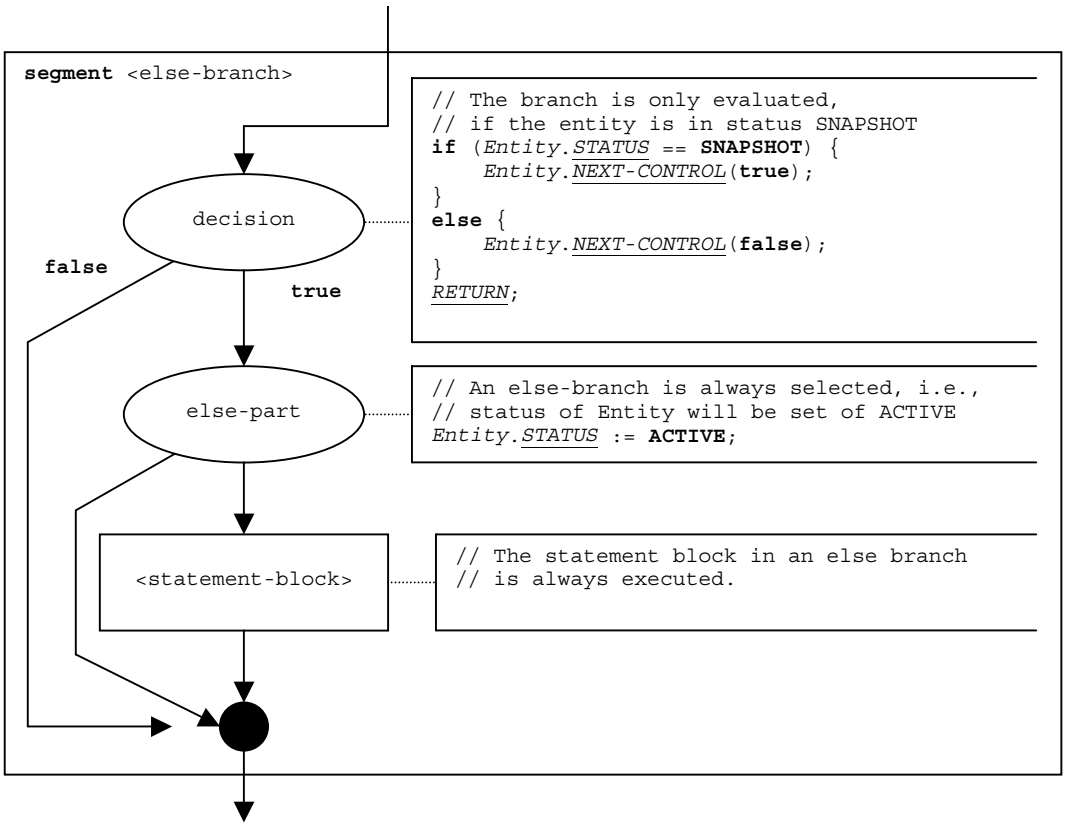


Figure 42 – Flow graph segment <else-branch>

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.

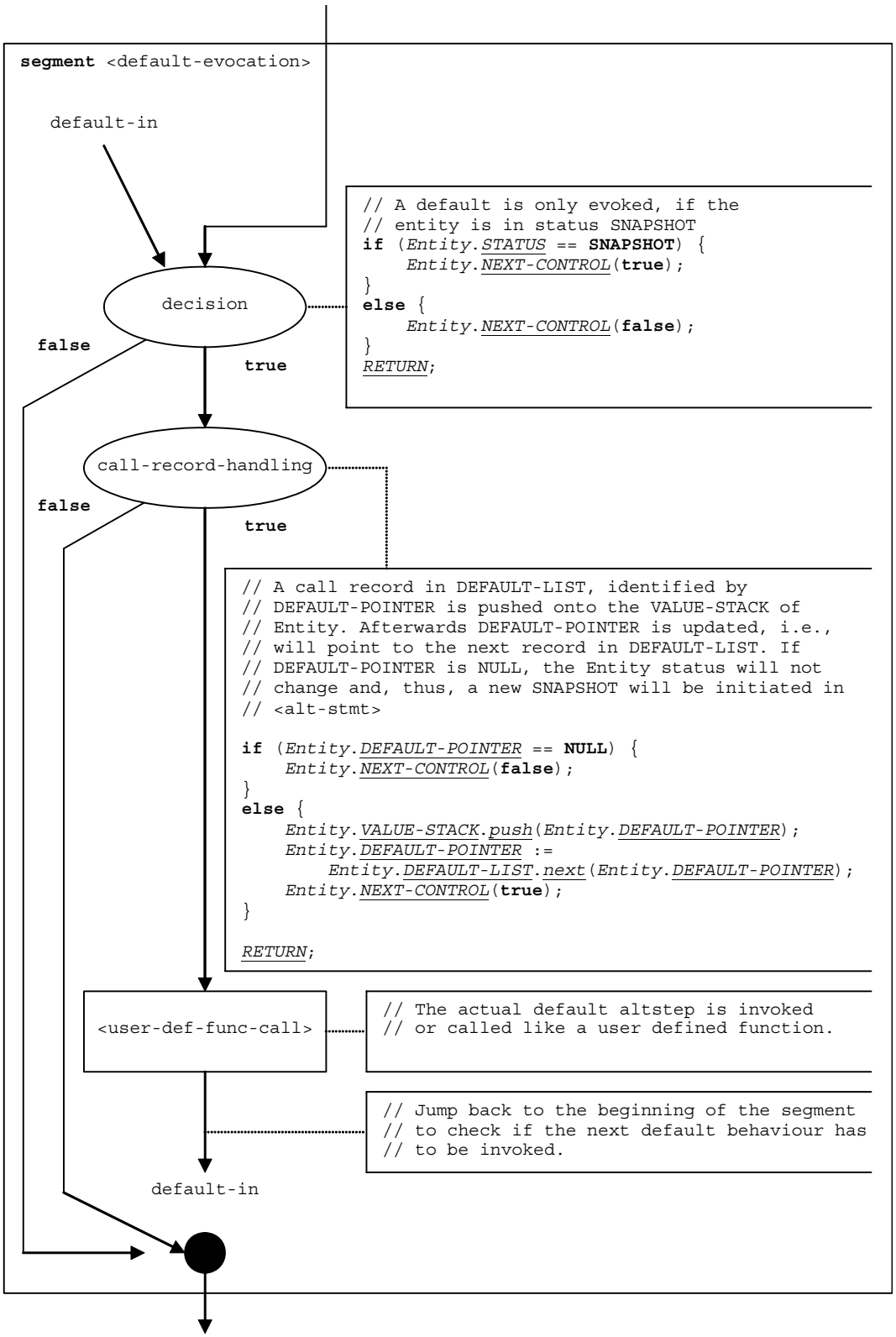


Figure 43 – Flow graph segment <default-evocation>

9.4 Altstep call

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

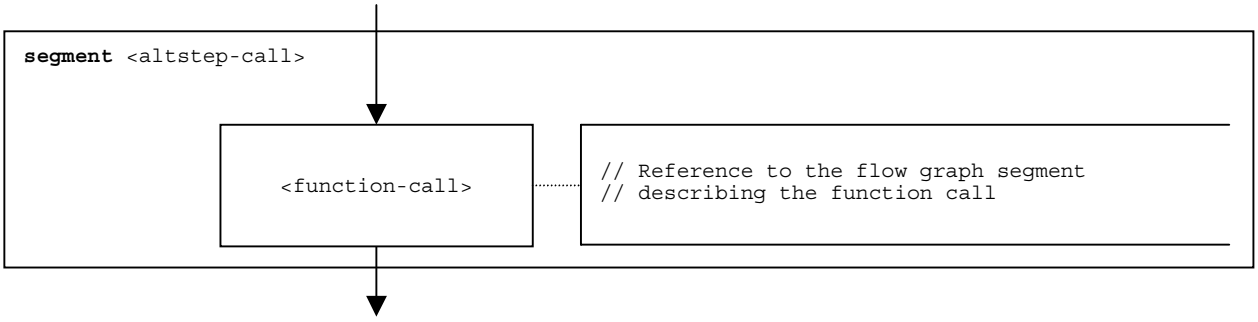


Figure 44 – Flow graph segment <altstep-call>

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.

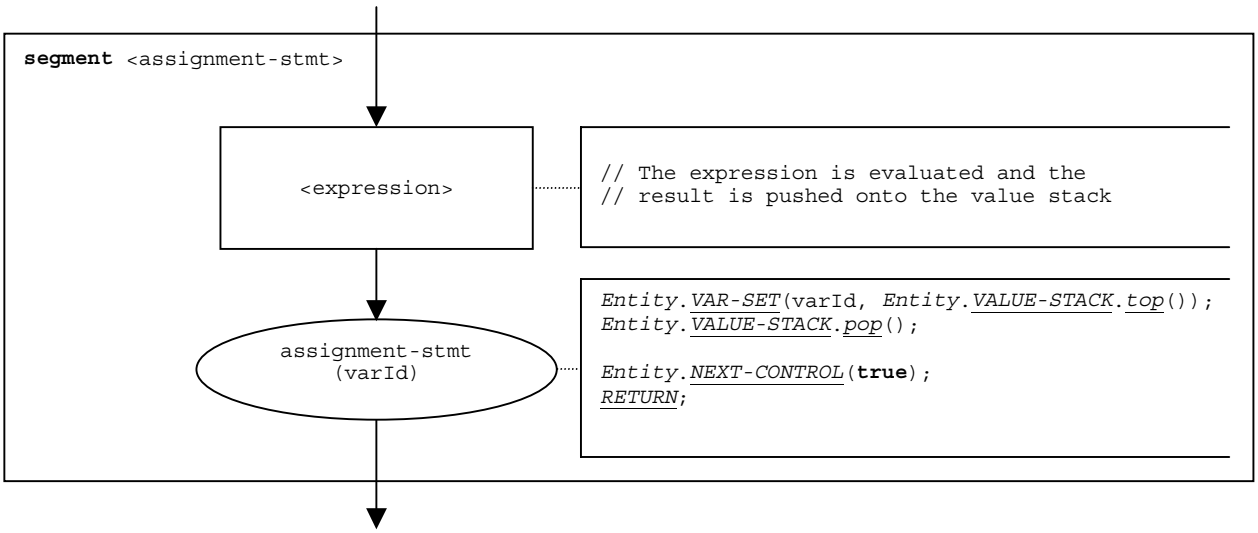


Figure 45 – Flow graph segment <assignment-stmt>

9.6 Call operation

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

```
<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.

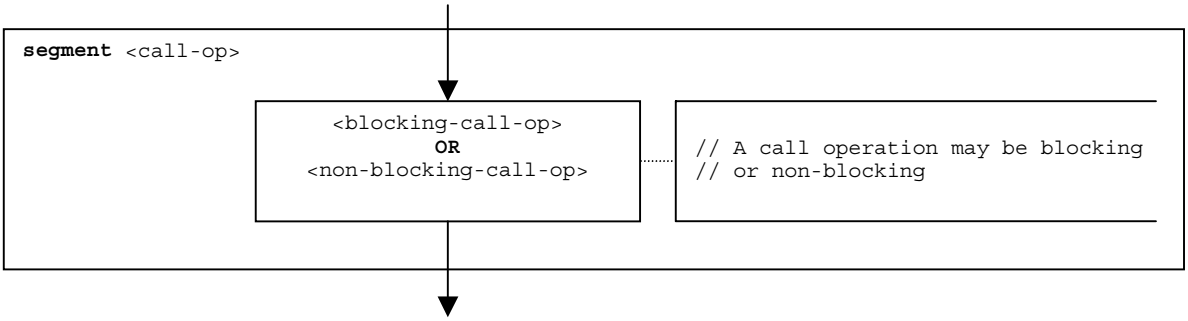


Figure 46 – Flow graph segment <call-op>

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.

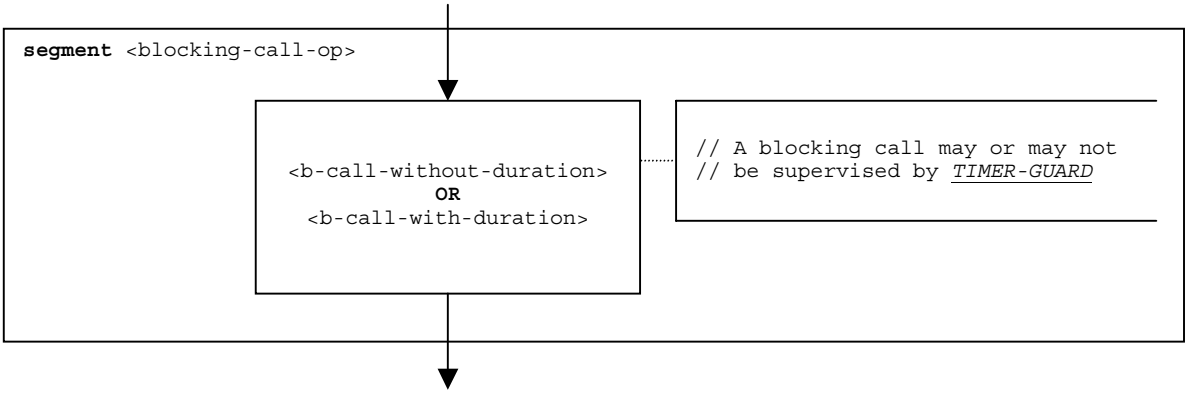


Figure 47 – Flow graph segment <blocking-call-op>

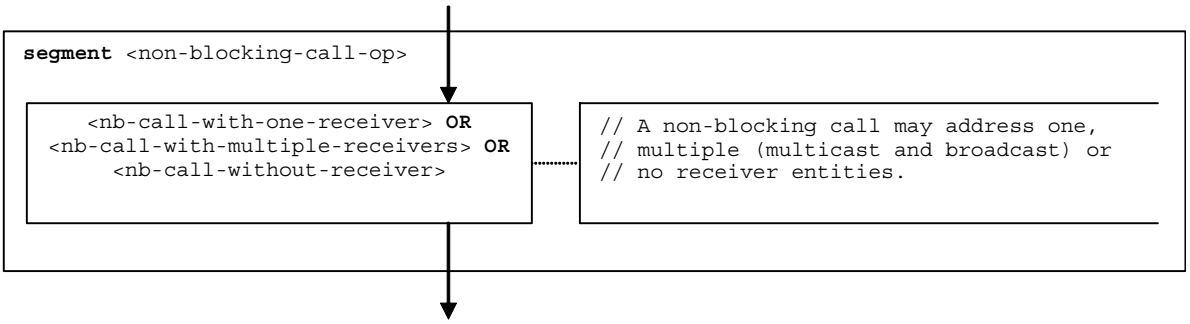


Figure 48 – Flow graph segment <non-blocking-call-op>

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.

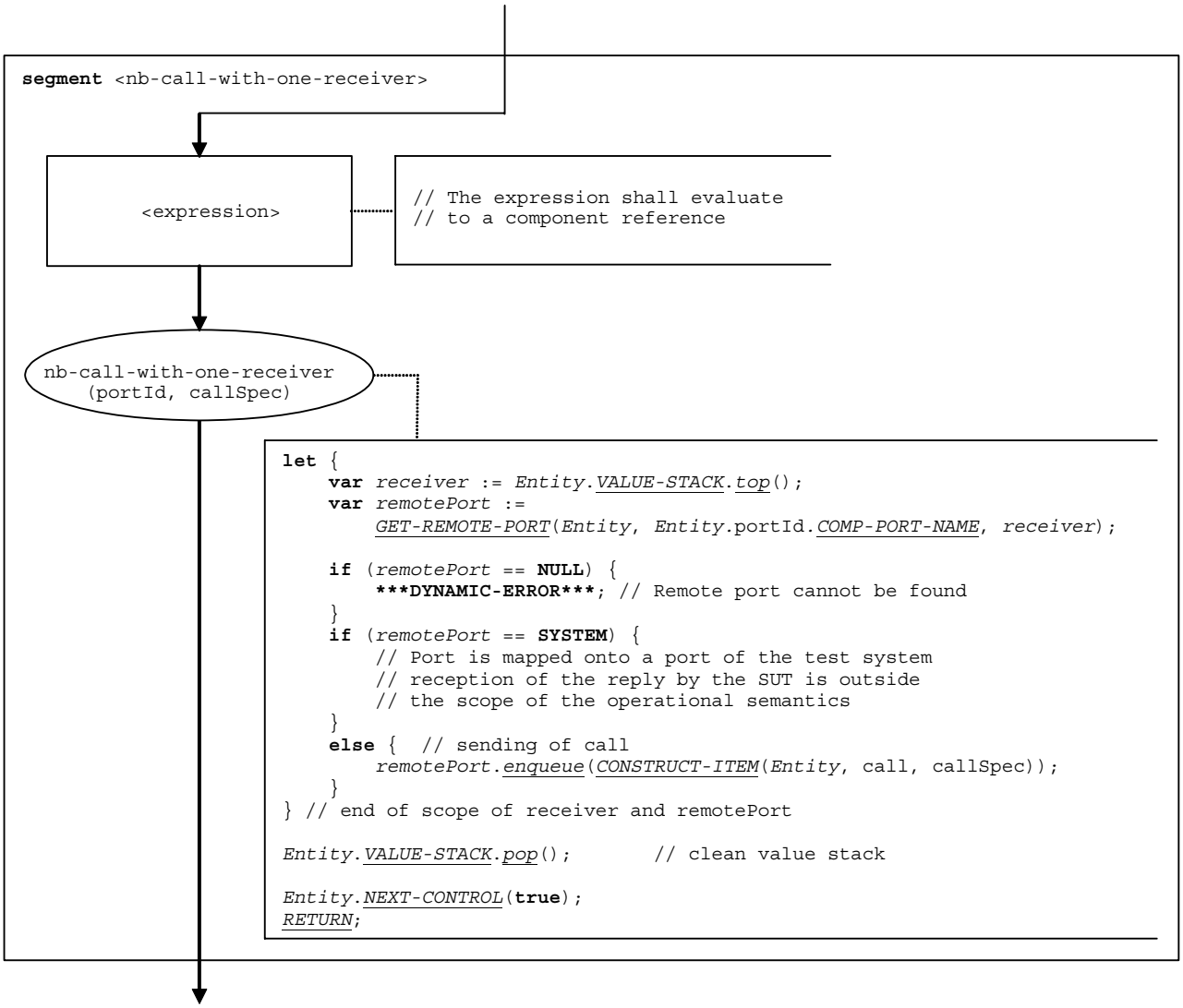


Figure 49 – Flow graph segment <nb-call-with-one-receiver>

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

The flow graph segment `<nb-call-with-multiple-receivers>` in Figure 49-a 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.

The component references 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 `nb-call-with-multiple-receivers` in Figure 49-a. The `number` parameter is 1 in case of broadcast communication, i.e., the keyword **`all component`** is top element in the value stack.

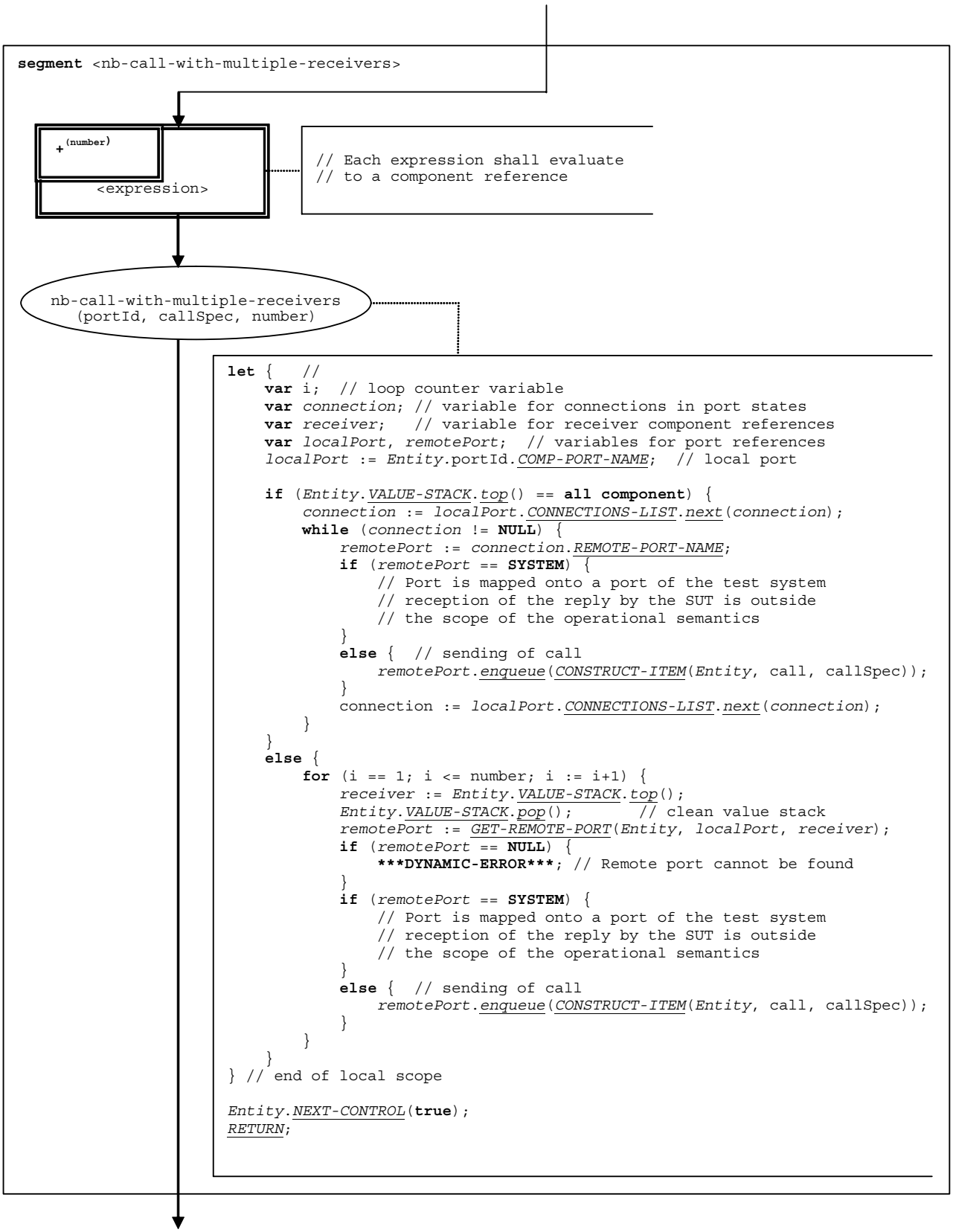


Figure 49-a – Flow graph segment <nb-call-with-multiple-receivers>

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.

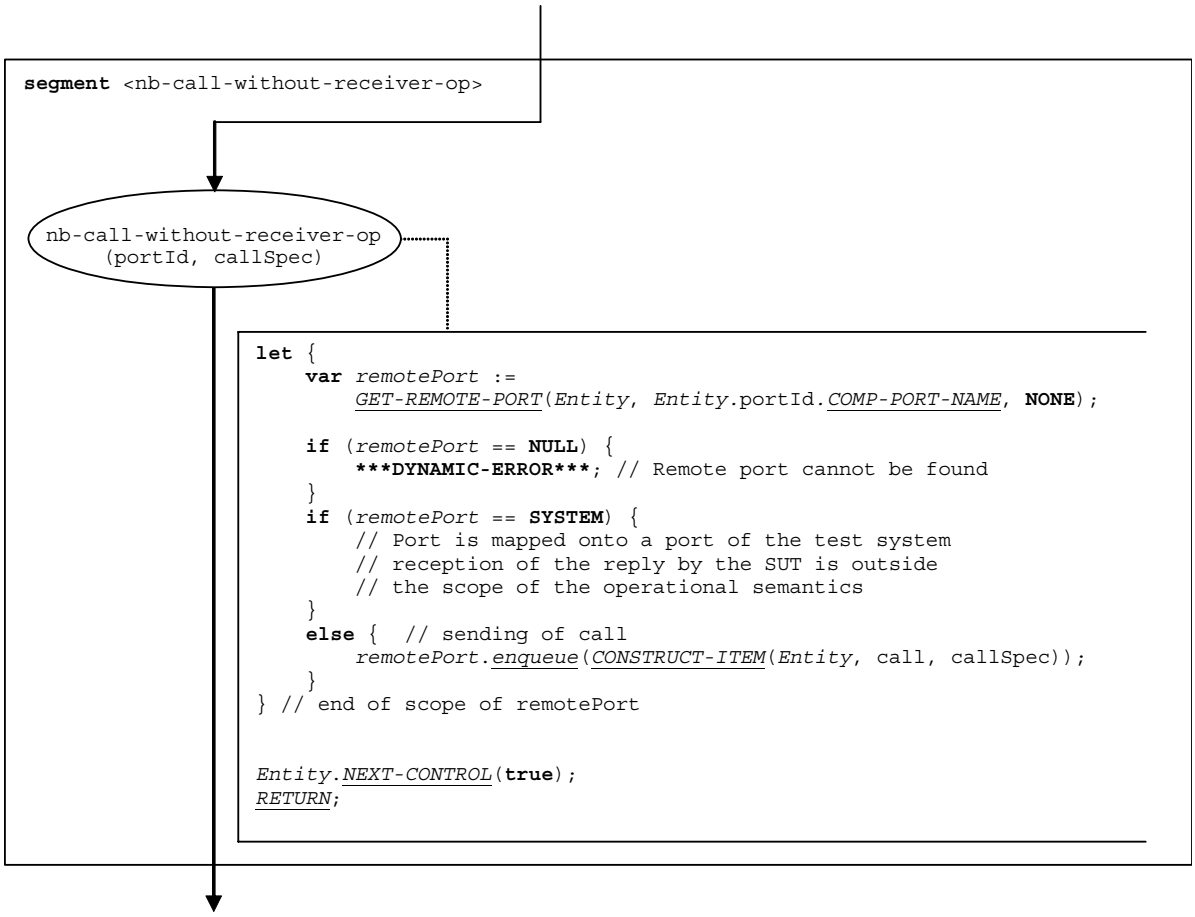


Figure 50 – Flow graph segment <nb-call-without-receiver>

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.

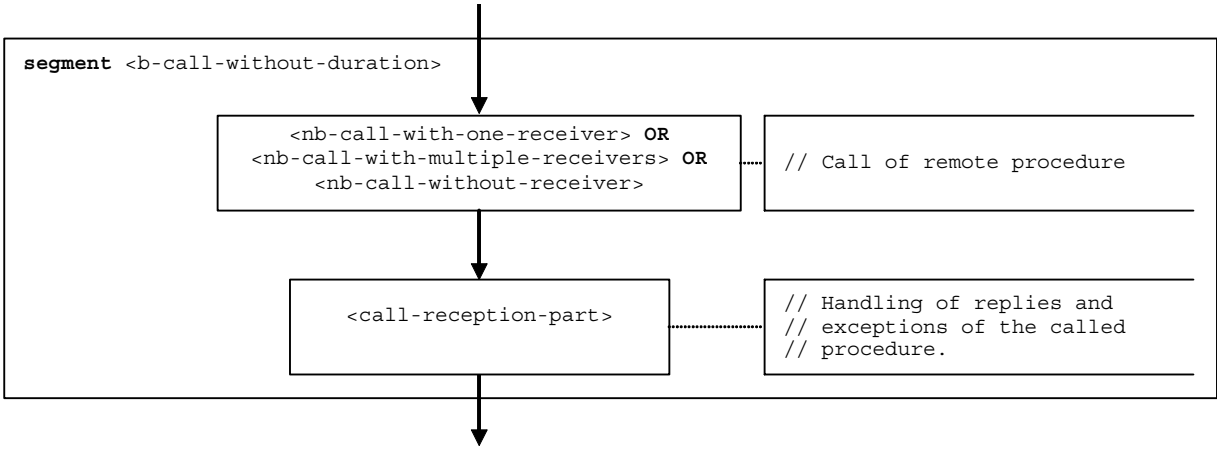


Figure 51 – Flow graph segment <b-call-without-duration>

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.

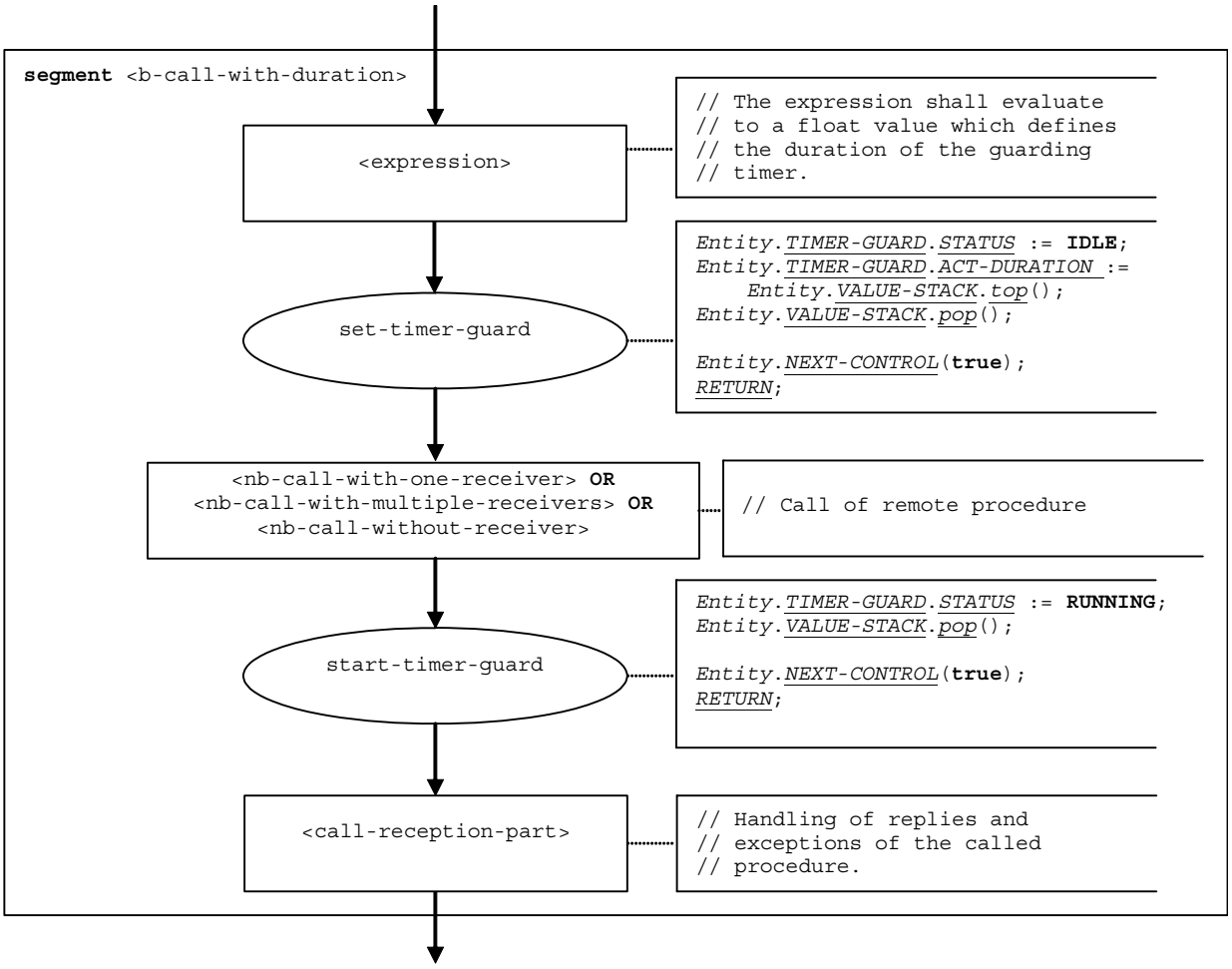


Figure 52 – Flow graph segment <b-call-with-duration>

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.

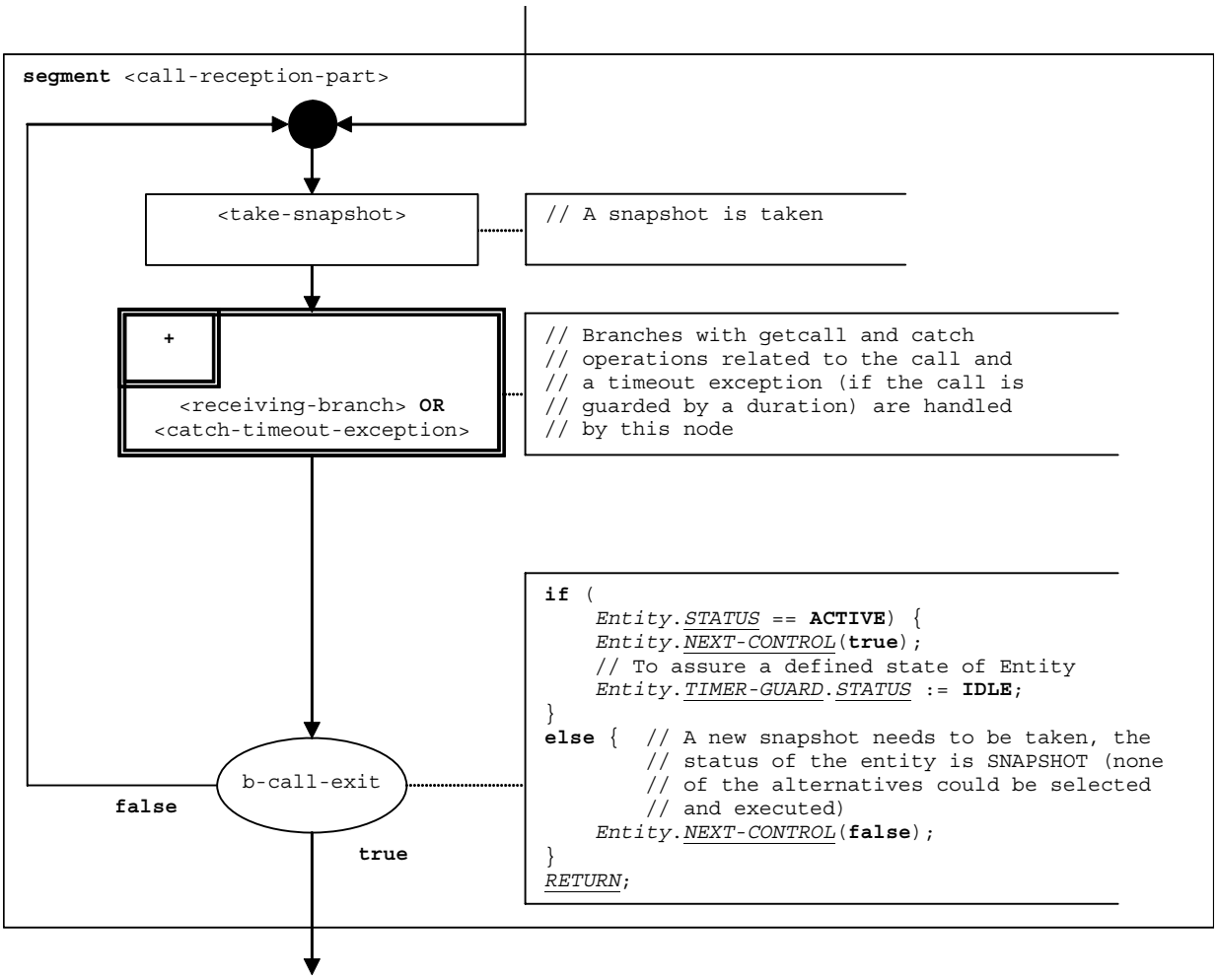


Figure 53 – Flow graph segment <call-reception-part>

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.

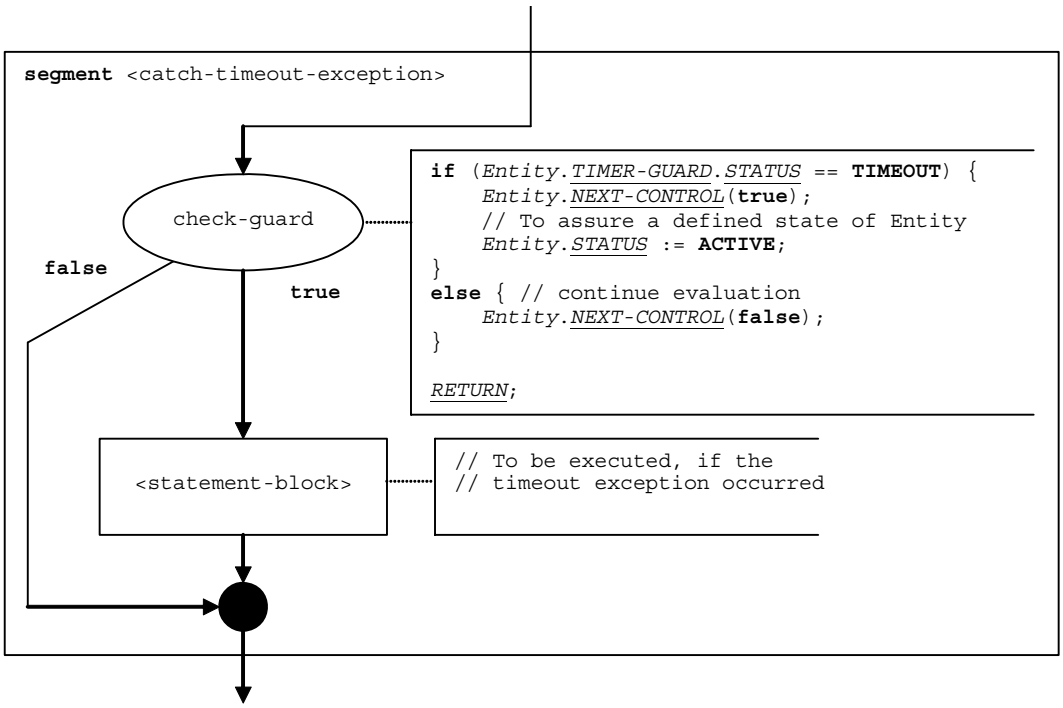


Figure 54 – Flow graph segment <catch-timeout-exception>

9.7 Catch operation

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

```
<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. This figure refers to flow graph segments related to the **receive** operation (see clause 9.37).

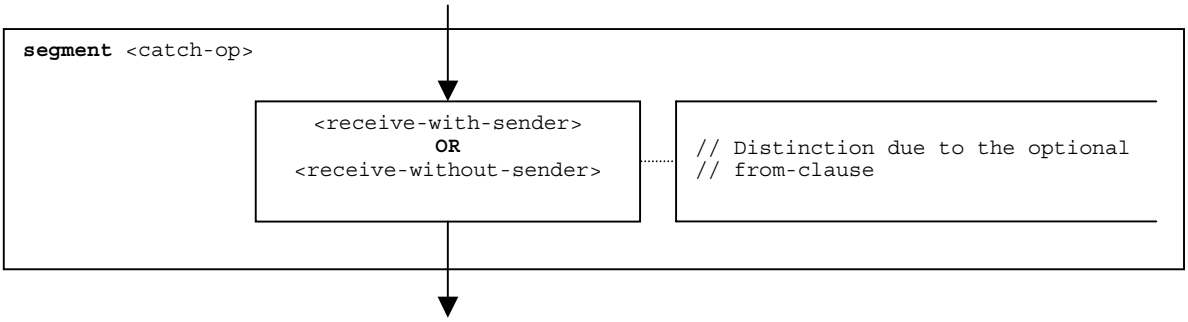


Figure 55 – Flow graph segment <catch-op>

9.8 Check operation

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

```
<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.

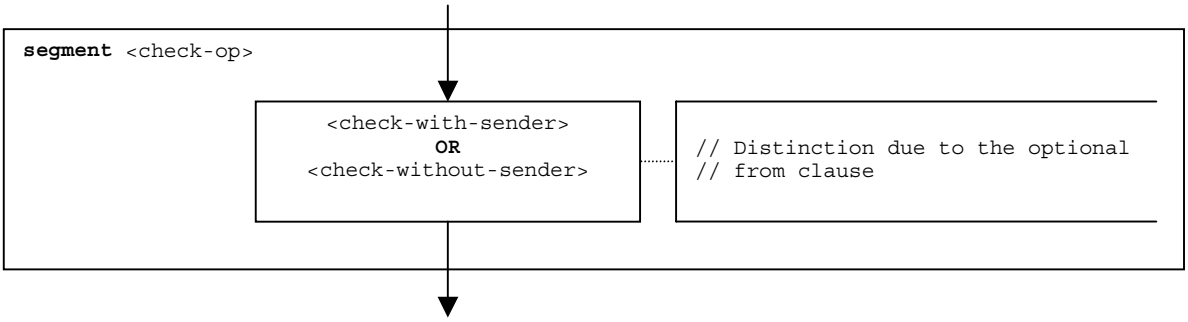


Figure 56 – Flow graph segment <check-op>

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 is specified in form of an expression.

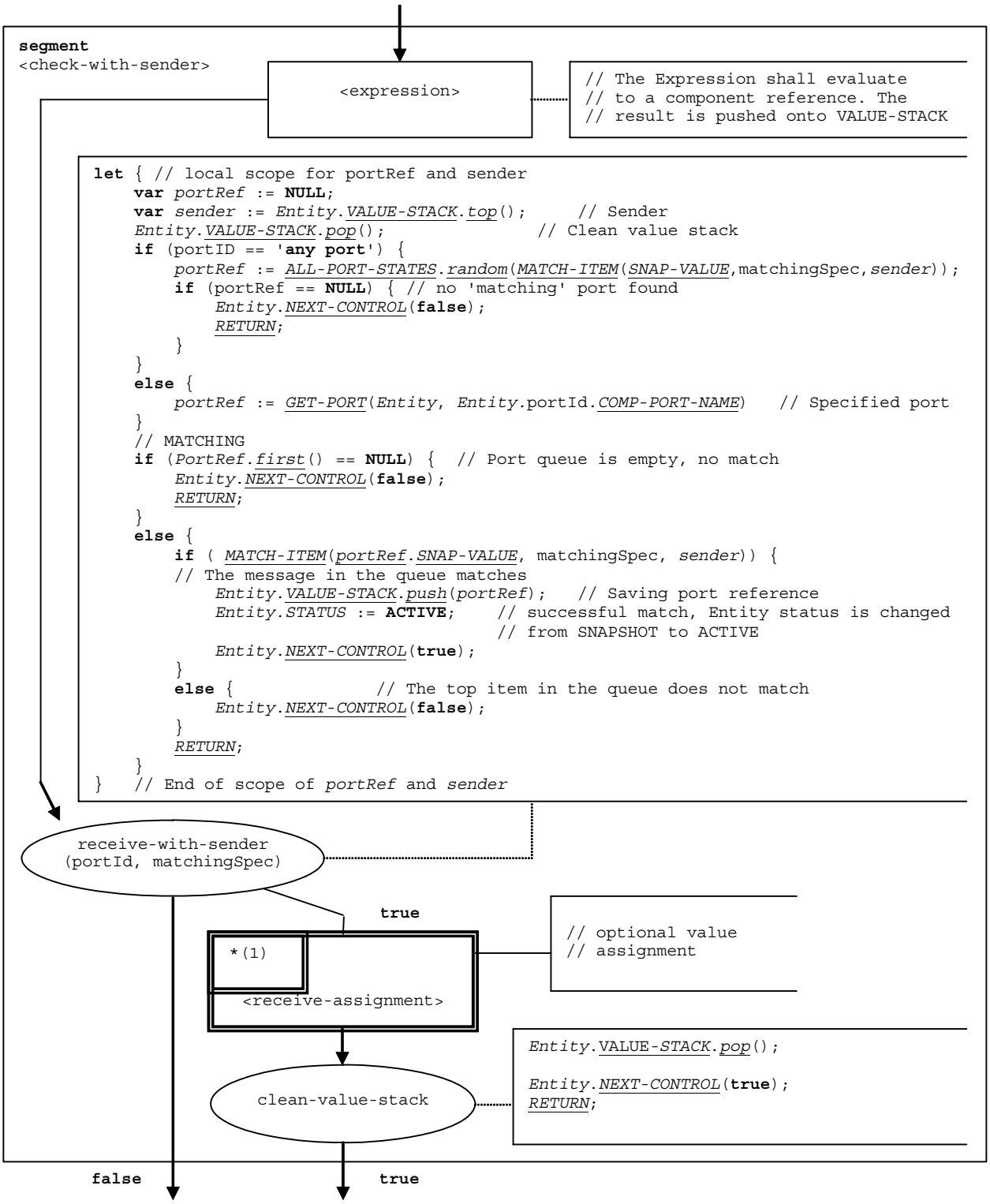


Figure 57 – Flow graph segment <check-with-sender>

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.

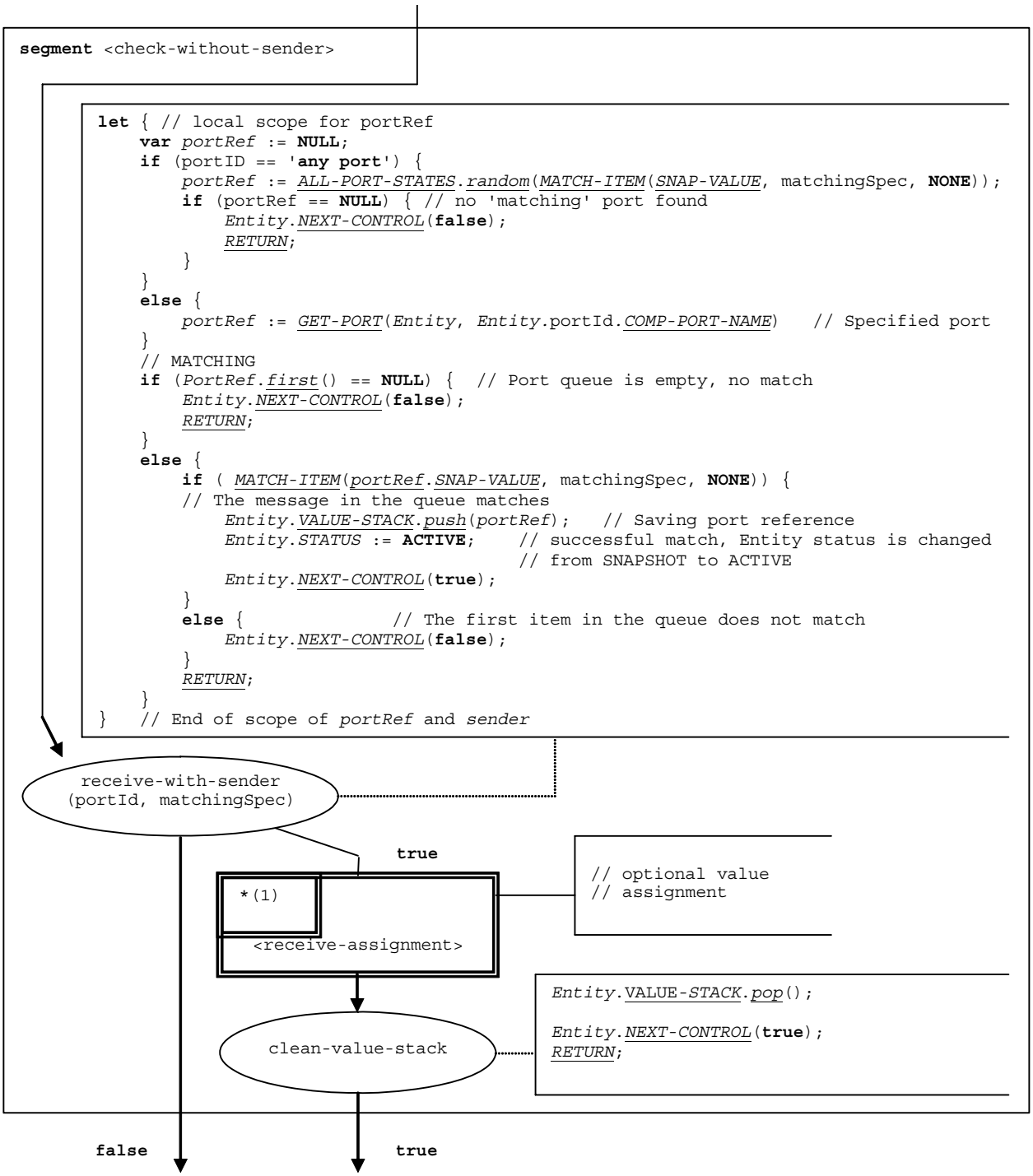


Figure 58 – Flow graph segment <check-without-sender>

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.

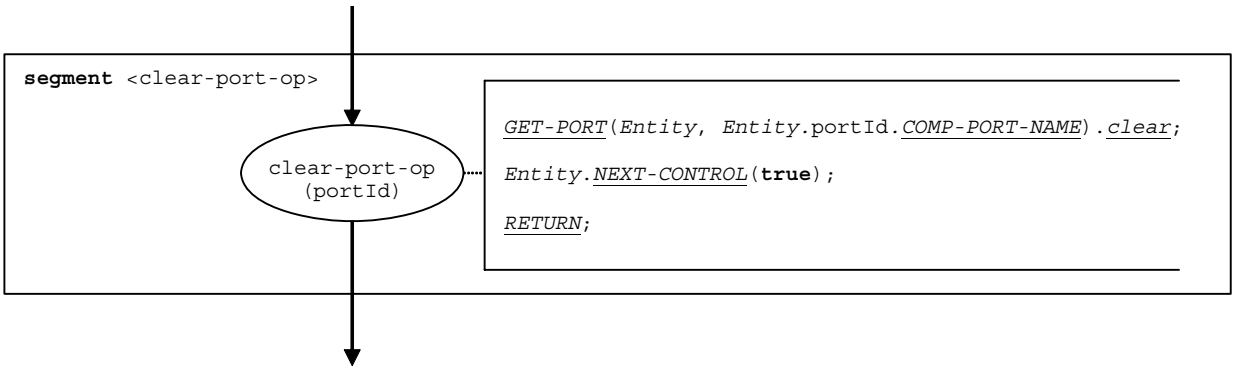


Figure 59 – Flow graph segment <clear-port-op>

9.10 Connect operation

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

```
connect(<component-expression1>:<portId1>, <component-expression2>:<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 are 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.

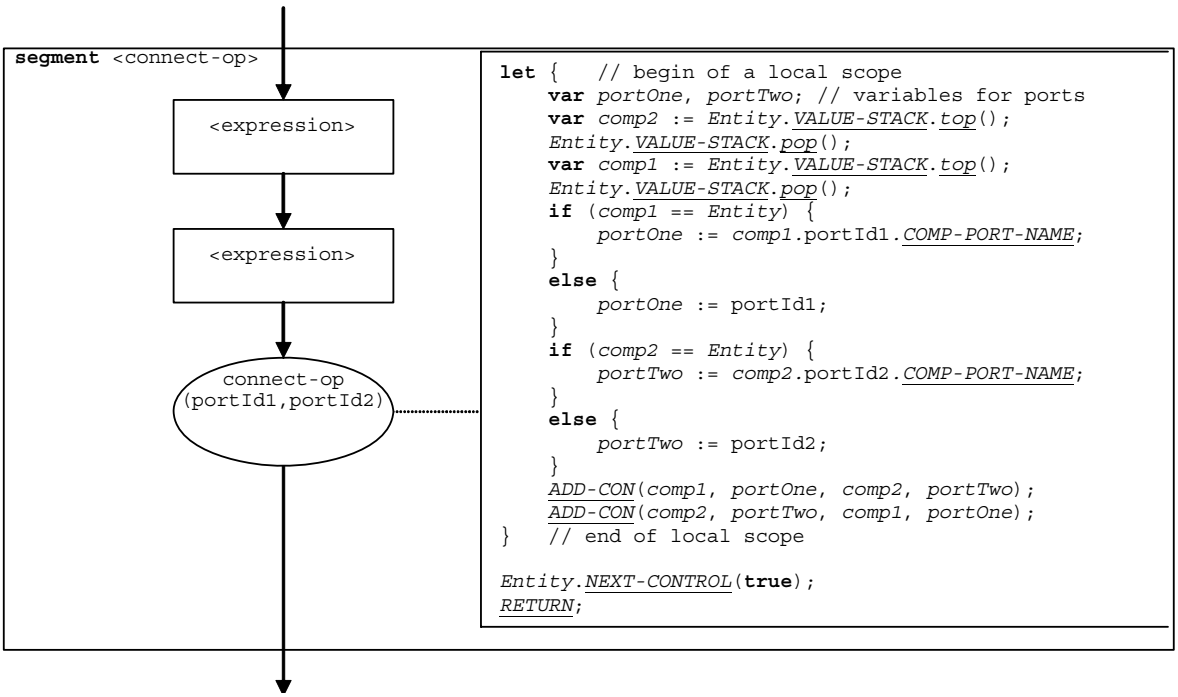


Figure 60 – Flow graph segment <connect-op>

9.11 Constant definition

The syntactical structure of a **constant** definition is:

```
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 shall never occur on the left side of an assignment, shall 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.

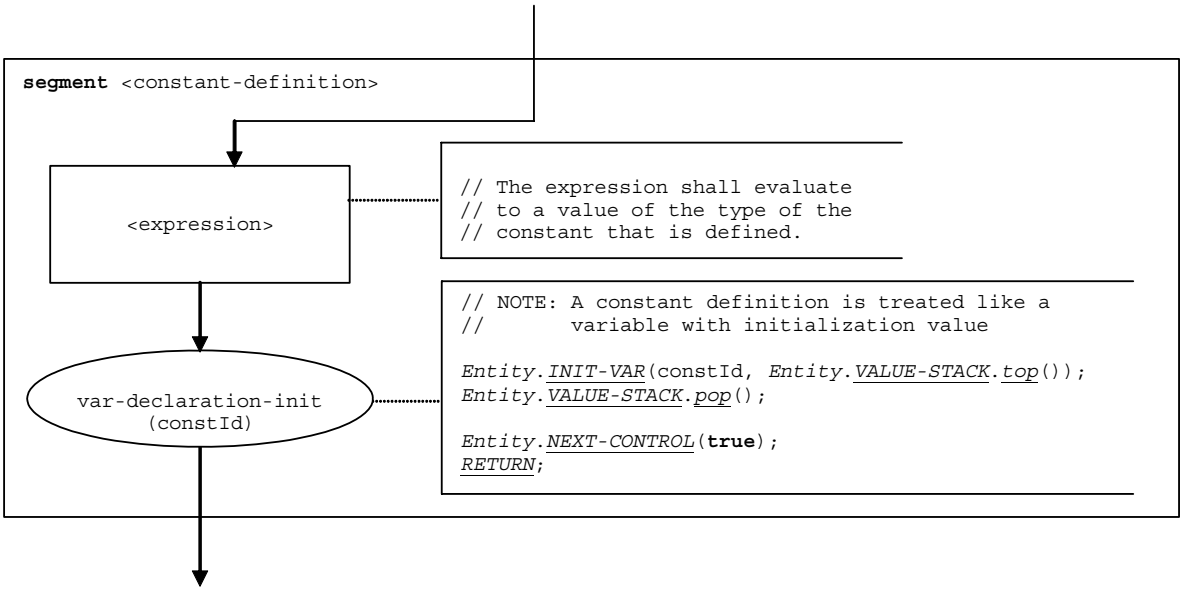


Figure 61 – Flow graph segment <constant-definition>

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.

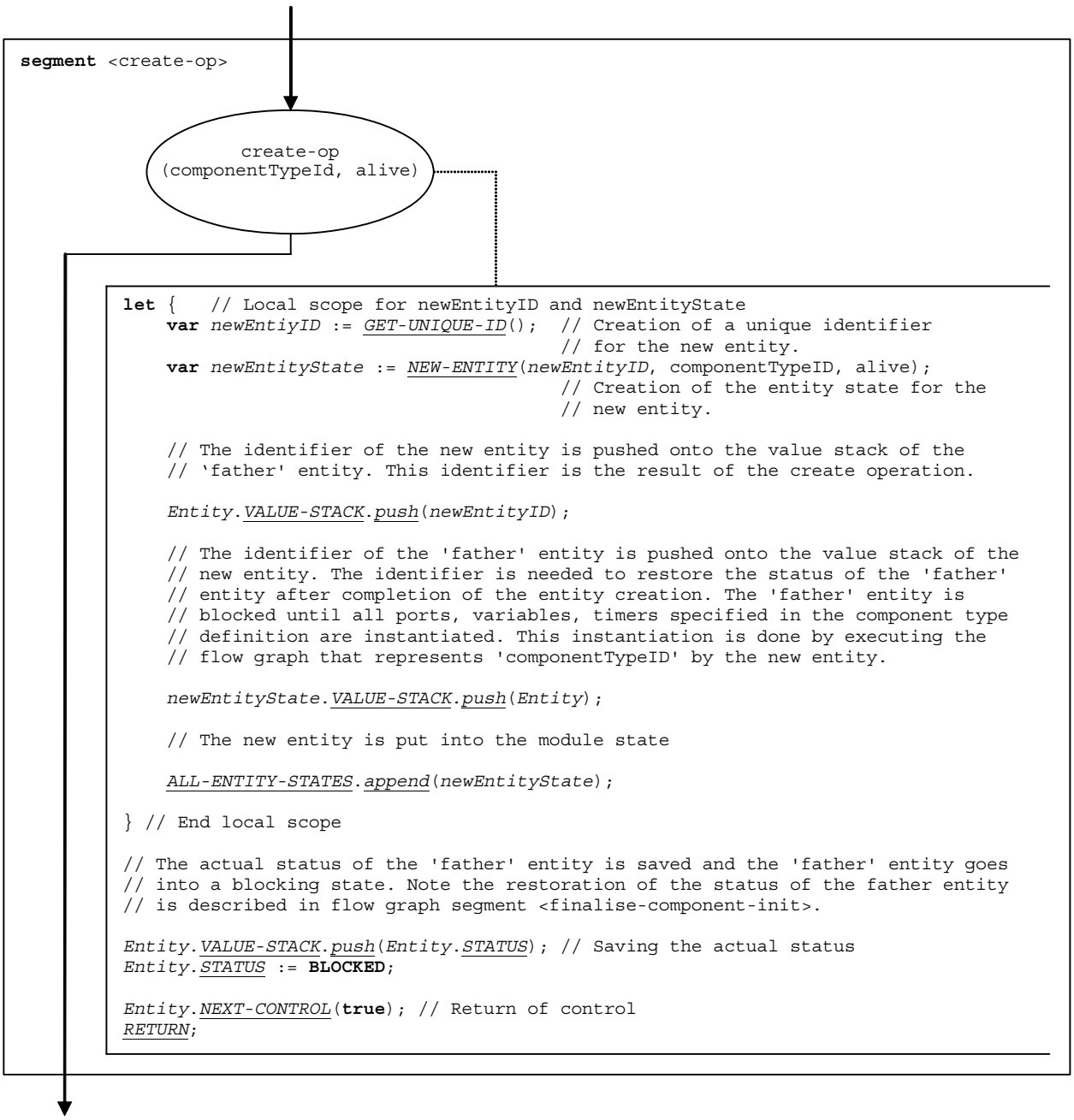


Figure 62 – Flow graph segment <create-op>

9.13 Deactivate statement

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

```
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 63-a.

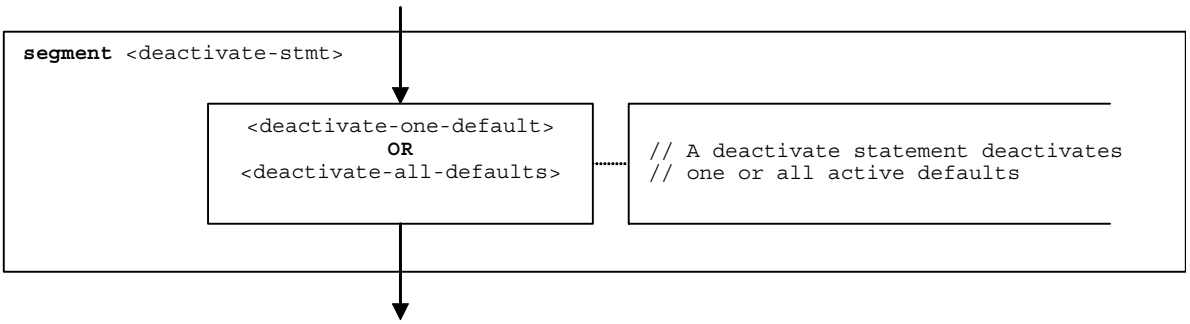


Figure 63-a – Flow graph segment <deactivate-stmt>

9.13.1 Flow graph segment <deactivate-one-default>

The flow graph segment `<deactivate-one-default>` in Figure 63-b 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.

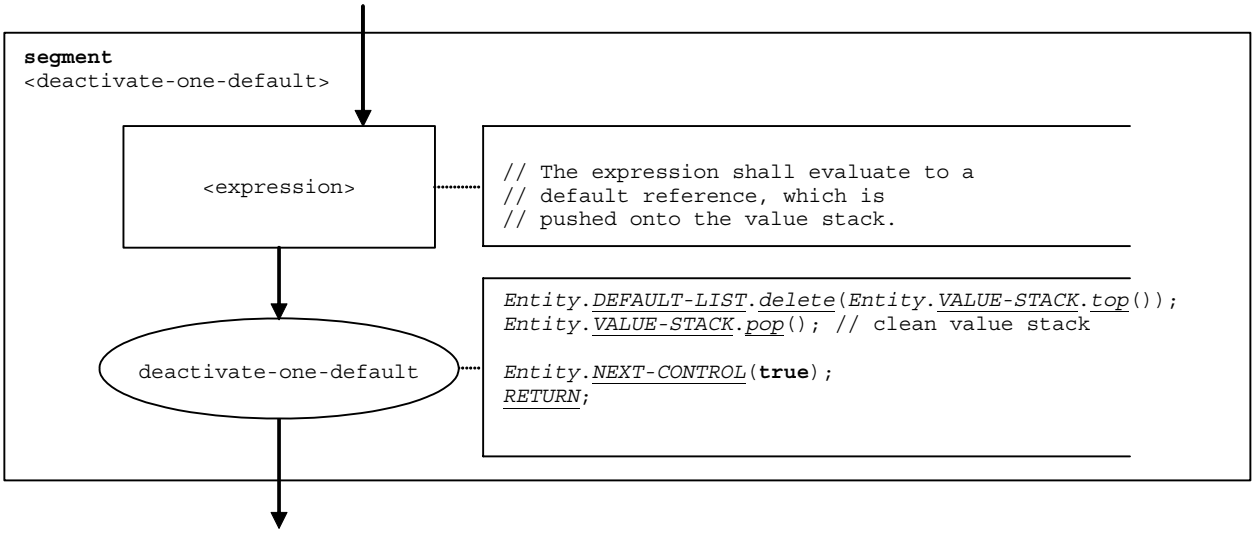


Figure 63-b – Flow graph segment <deactivate-one-default>

9.13.2 Flow graph segment <deactivate-all-defaults>

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

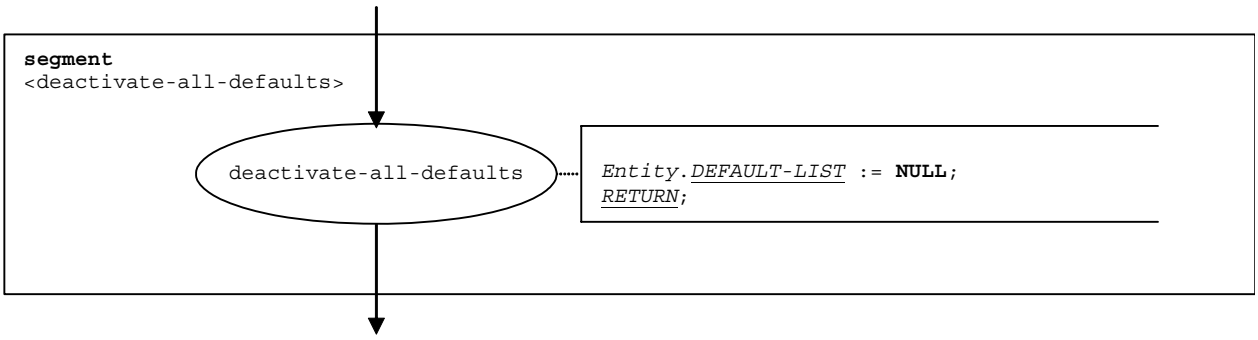


Figure 63-c – Flow graph segment <deactivate-all-defaults>

9.14 Disconnect operation

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

```
disconnect(<component-expression1>:<portId1>,
    <component-expression2>:<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-expression1>` and `<component-expression2>`. 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 execution of the **disconnect** operation is defined by the flow graph segment `<disconnect-op>` shown in Figure 64. In the flow graph segment the first expression to be evaluated refers to `<component-expression1>` and the second expression to `<component-expression2>`, i.e., the `<component-expression2>` is on top of the value stack when the `disconnect-op` node is executed.

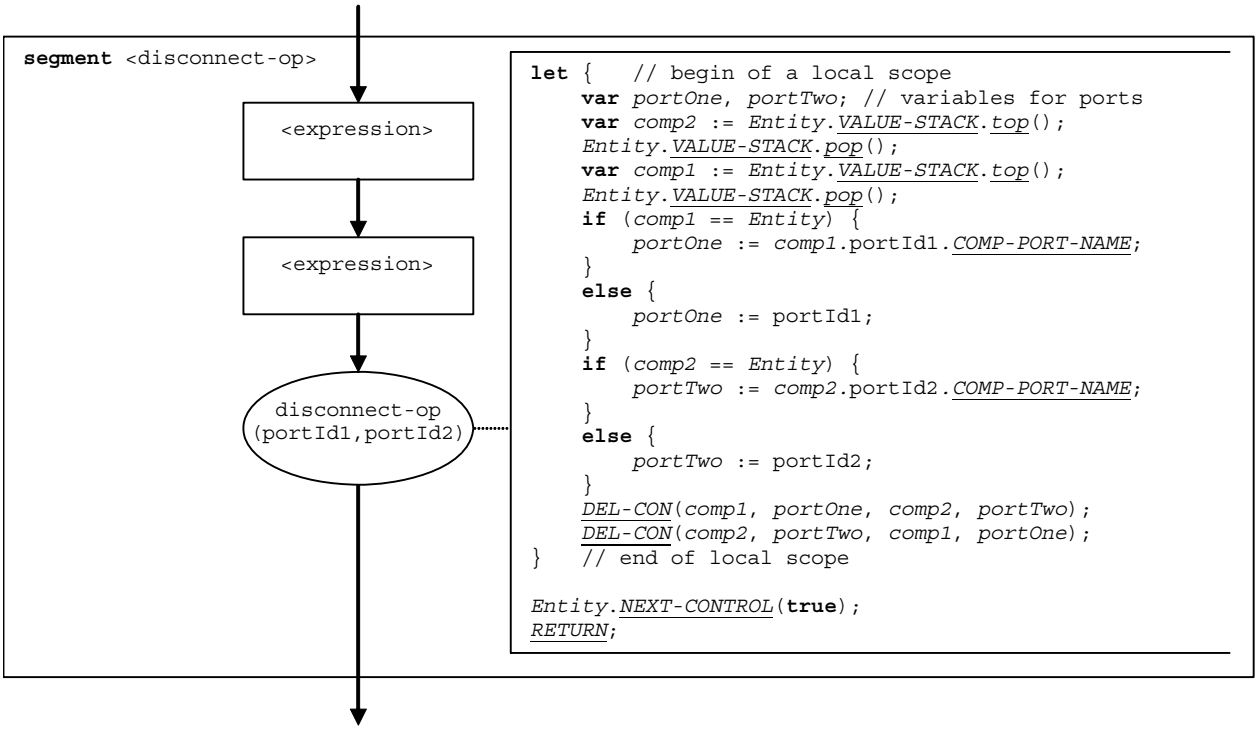


Figure 64 – Flow graph segment <disconnect-op>

9.15 Do-while statement

The syntactical structure of the **do-while** statement is:

```
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.

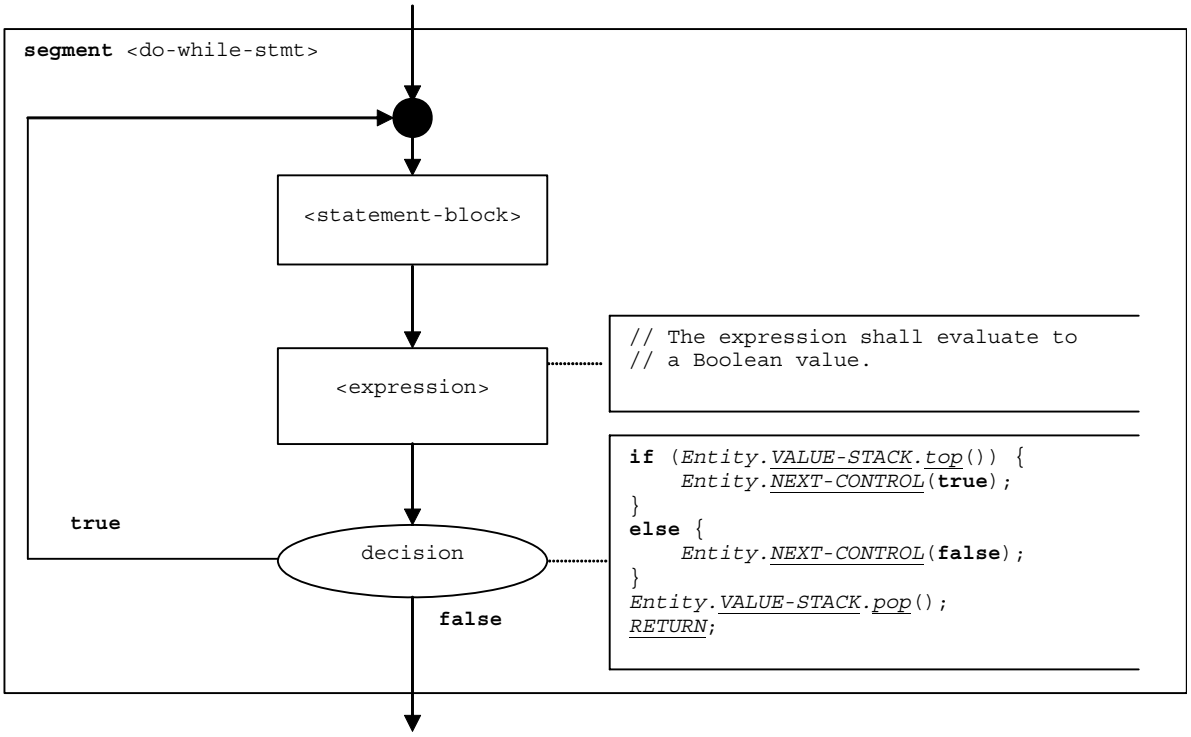


Figure 65 – Flow graph segment <do-while-stmt>

9.16 Done component operation

The syntactical structure of the **done** component operation is:

```
<component-expression>.done
```

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 flow graph segment <done-op> in Figure 66 defines the execution of the **done** component operation.

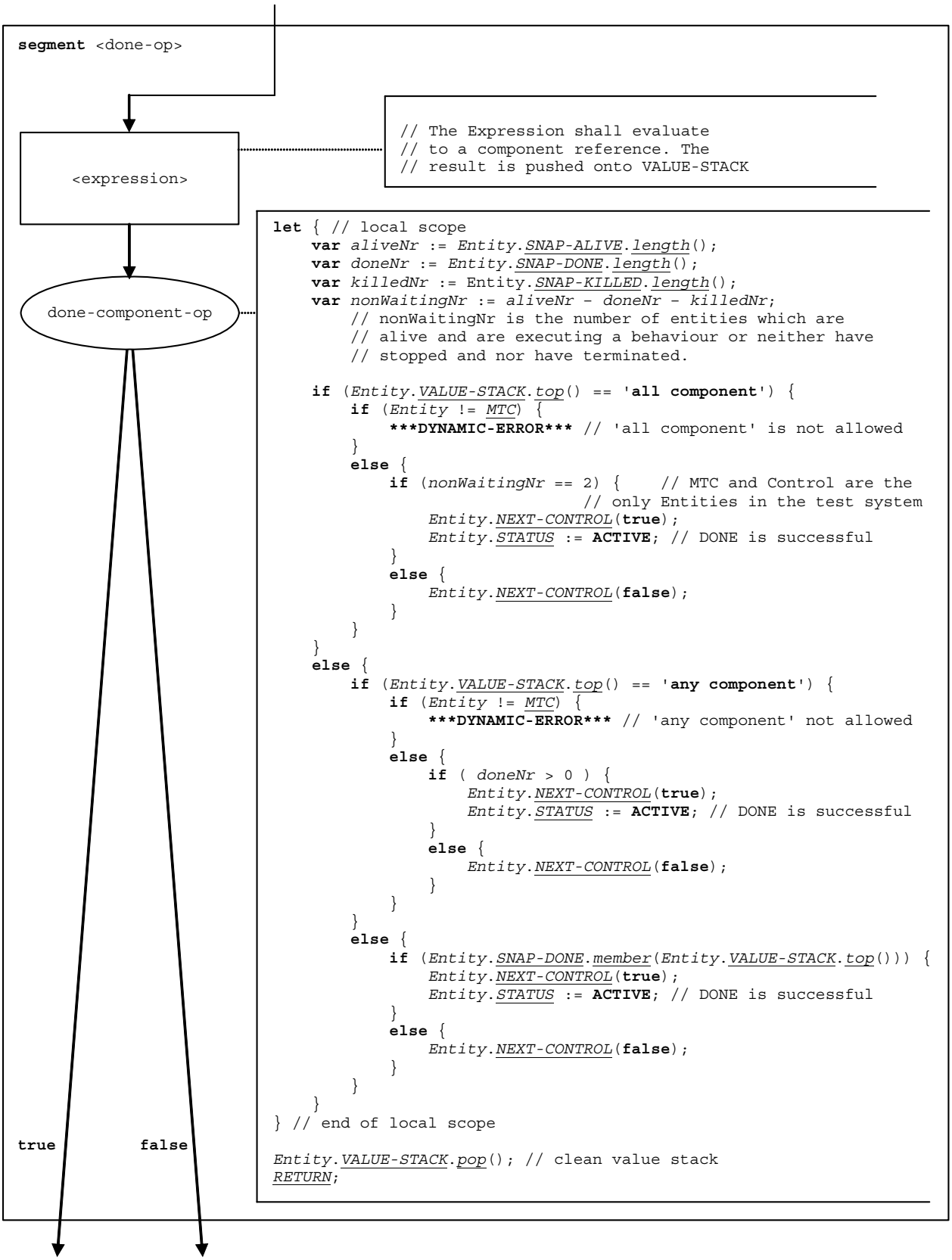


Figure 66 – Flow graph segment <done-op>

9.17 Execute statement

The syntactical structure of the **execute** statement is:

```
execute(<testCaseId>([<act-par1>, … , <act-parn>)]) [, <float-expression>])
```

The **execute** statement describes the execution of a test case `<testCaseId>` with the (optional) actual parameters `<act-par1>, … , <act-parn>`. 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.

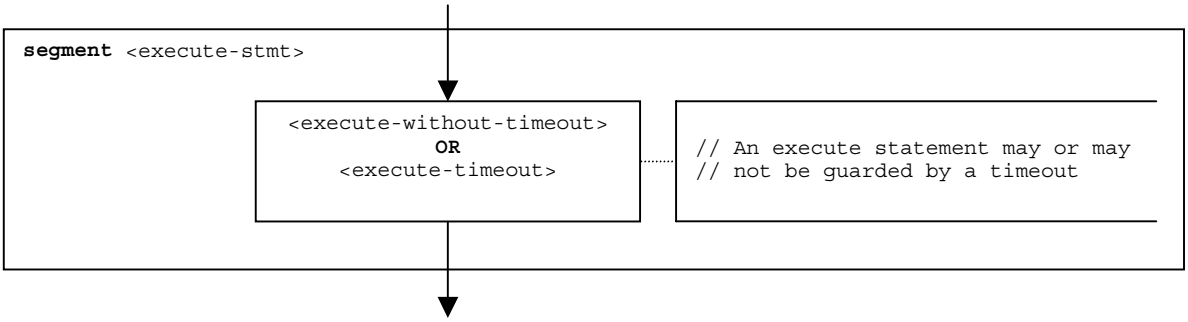


Figure 67 – Flow graph segment <execute-stmt>

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:

```
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**.

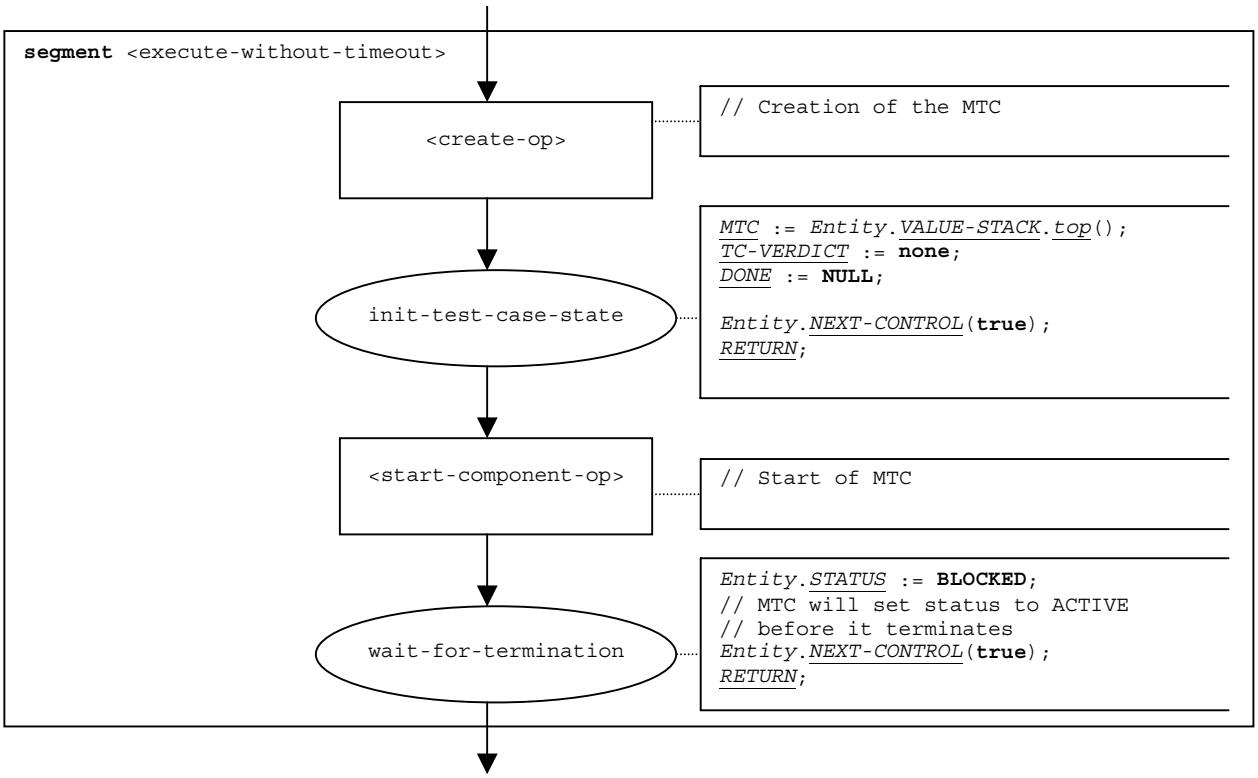


Figure 68 – Flow graph segment <execute-without-timeout>

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.

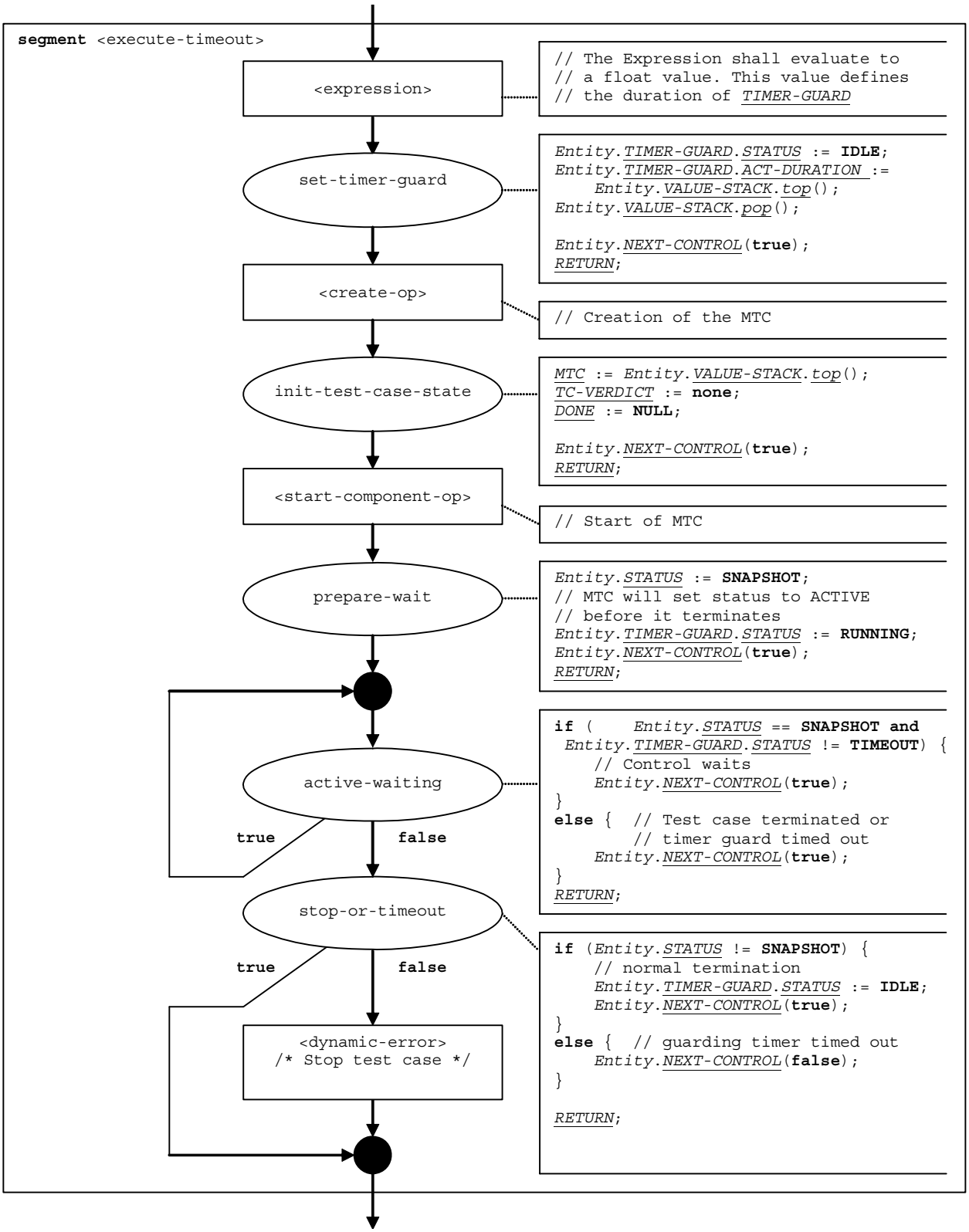


Figure 69 – Flow graph segment <execute-timeout>

9.18 Expression

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.

The syntactical structure of an expression is:

```
<lit-val> | <var-val> | <func-op-call> | <operator-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.

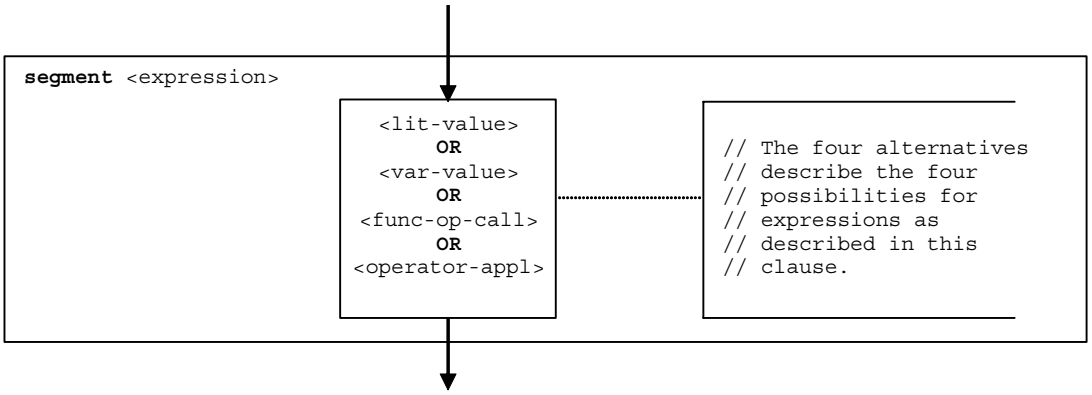


Figure 70 – Flow graph segment <expression>

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.

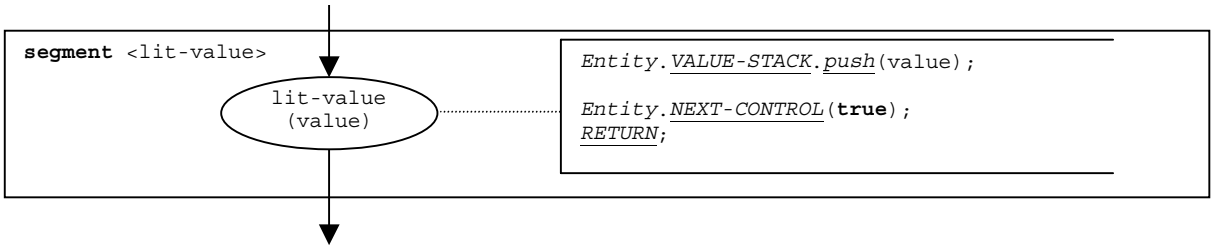


Figure 71 – Flow graph segment <lit-value>

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.

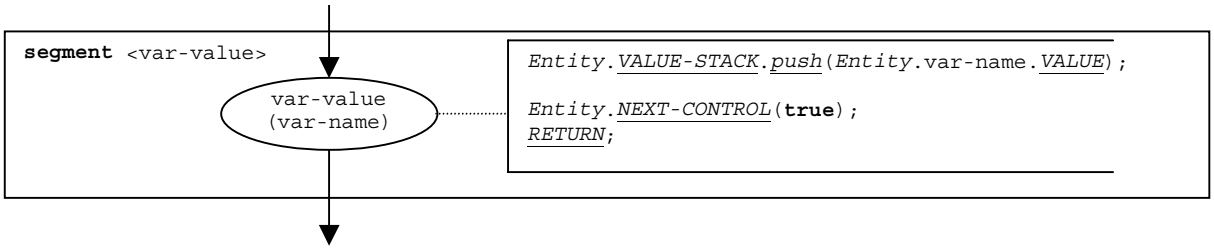


Figure 72 – Flow graph segment <var-value>

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.

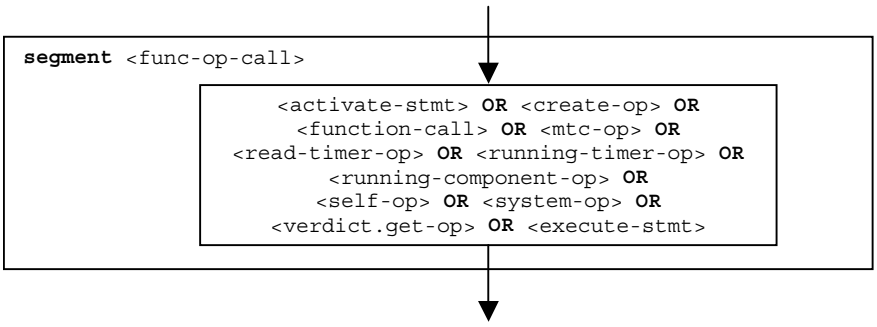


Figure 73 – Flow graph segment <func-op-call>

9.18.4 Flow graph segment <operator-appl>

The flow graph representation in Figure 74-a 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 of the result are considered to be part of the operator application (`Entity.APPLY-OPERATOR(operator)` statement in Figure 74-a), i.e., are not modelled by the operational semantics.

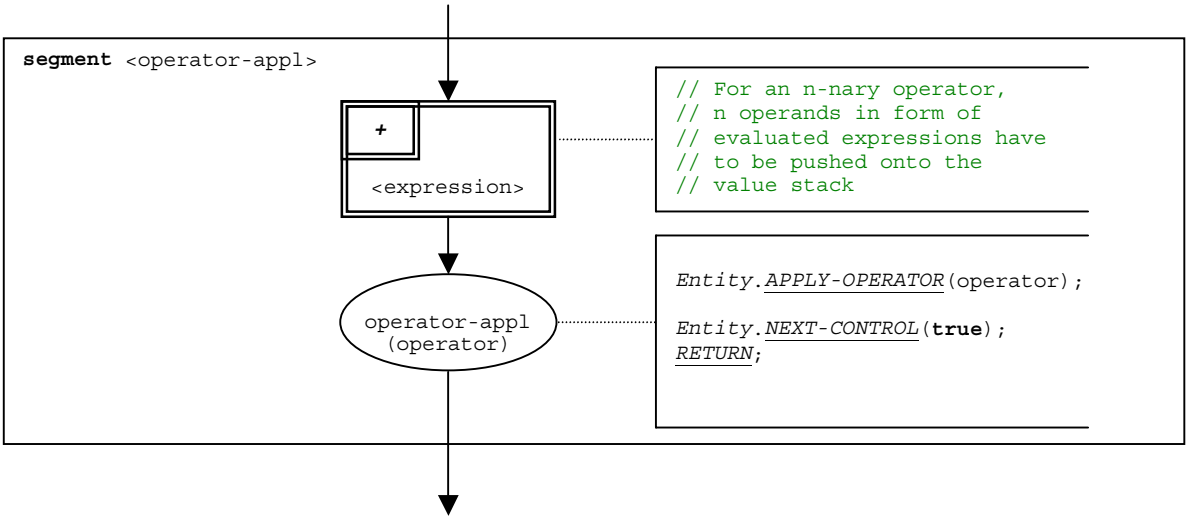


Figure 74-a – Flow graph segment <operator-appl>

9.18.5 Flow graph segment <dynamic-error>

In case of a dynamic error, the flow graph segment `<dynamic-error>` is invoked by the test system. 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).

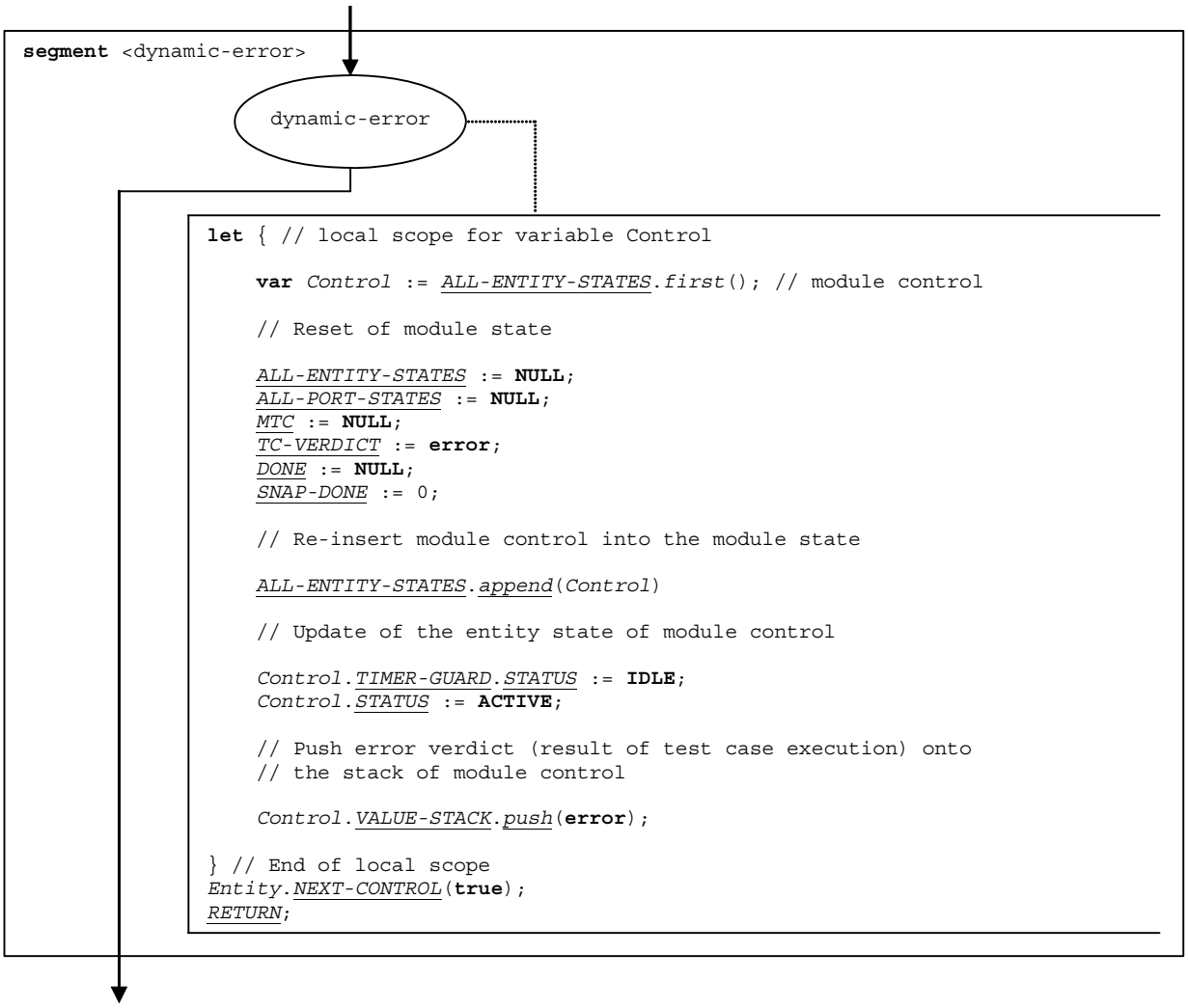


Figure 74-b – Flow graph segment <dynamic-error>

9.19 Flow graph segment <finalise-component-init>

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

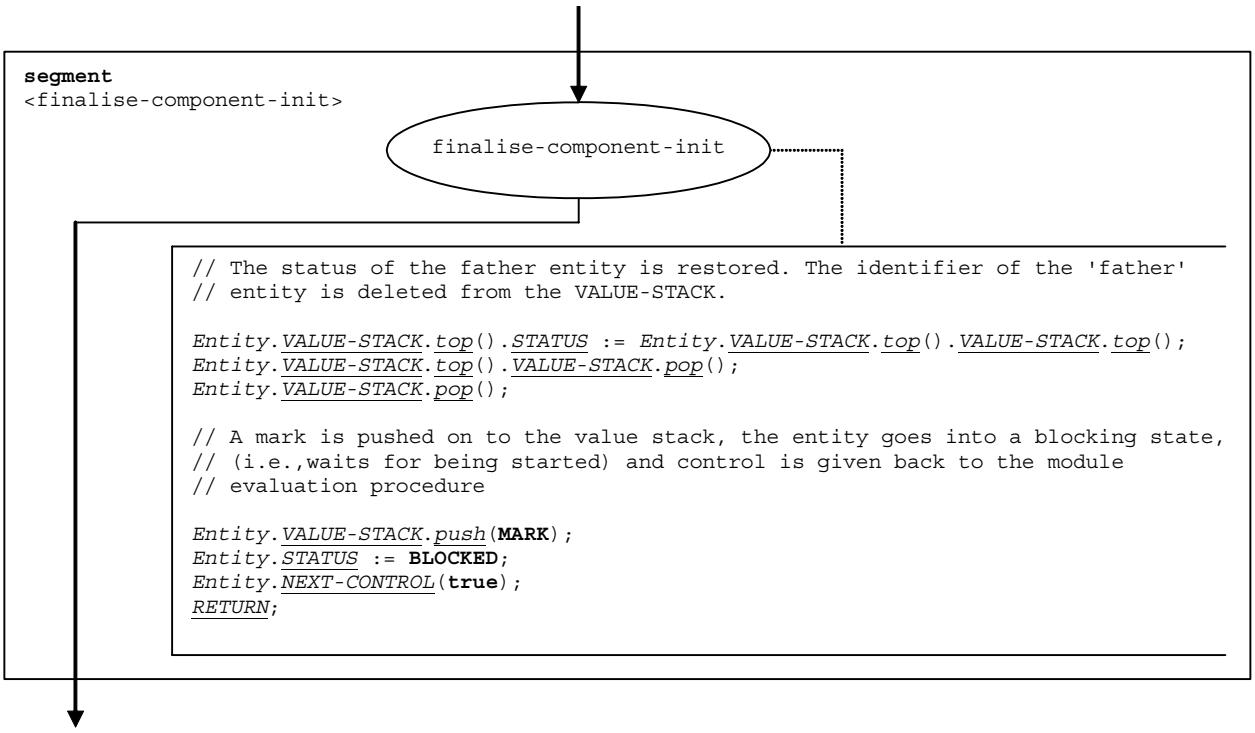


Figure 75 – Flow graph segment <finalise-component-init>

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.

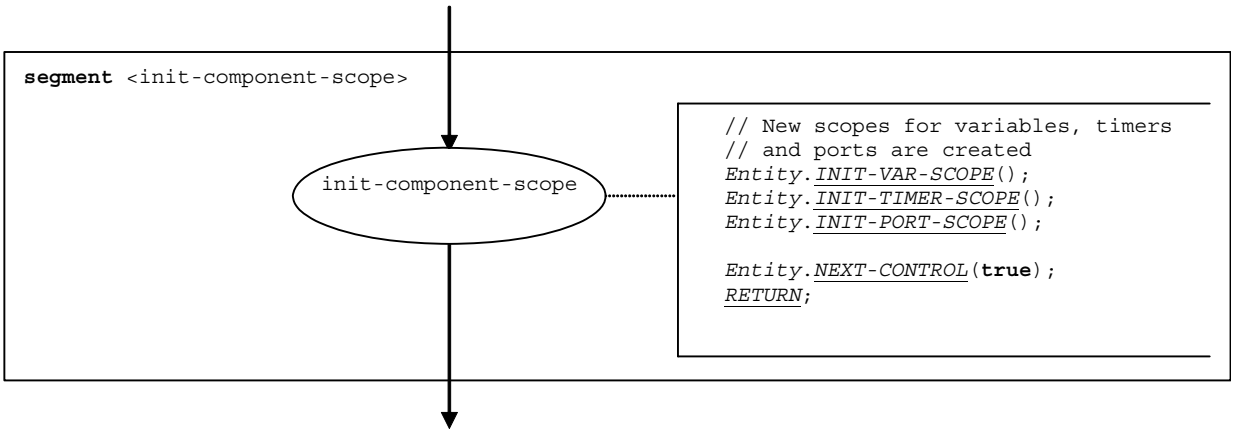


Figure 76 – Flow graph segment <init-component-scope>

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 76-a.

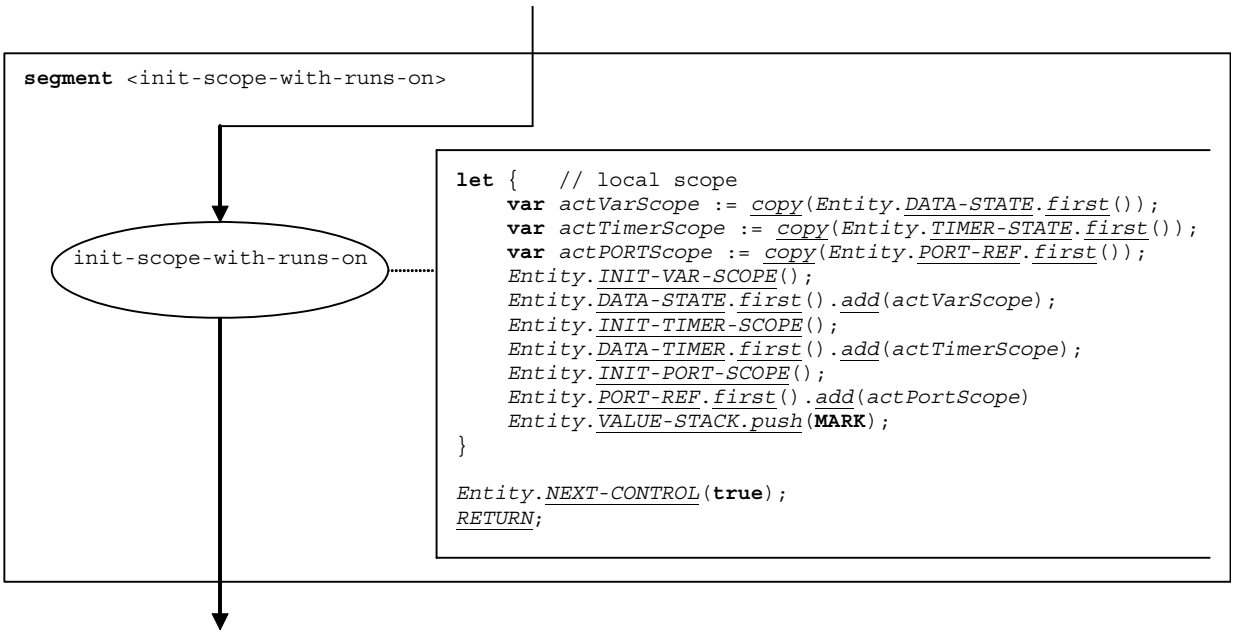


Figure 76-a – Flow graph segment <init-scope-with-runs-on>

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 76-b.

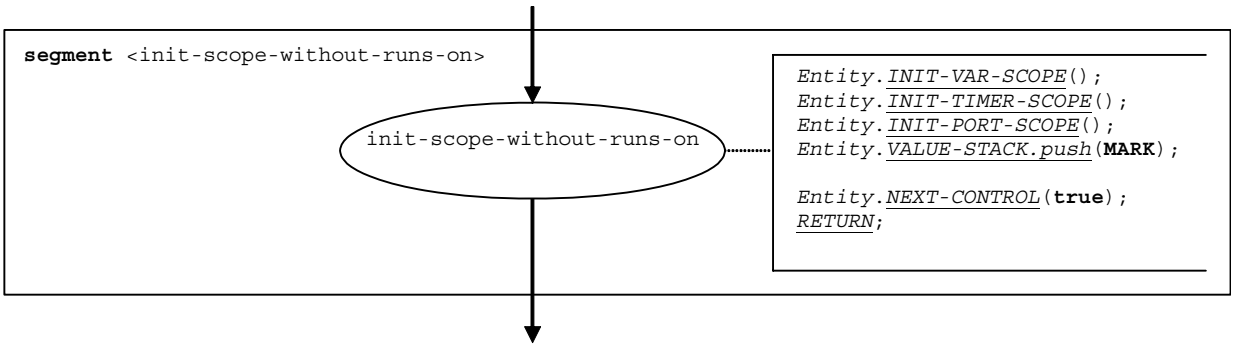


Figure 76-b – Flow graph segment <init-scope-without-runs-on>

9.21 Flow graph segment <parameter-handling>

The flow graph segment `<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 `<parameter-handling>` assumes that the call record of the called test case, altstep or function is the top of the value stack.

The execution of flow graph segment `<parameter-handling>` is shown in Figure 77.

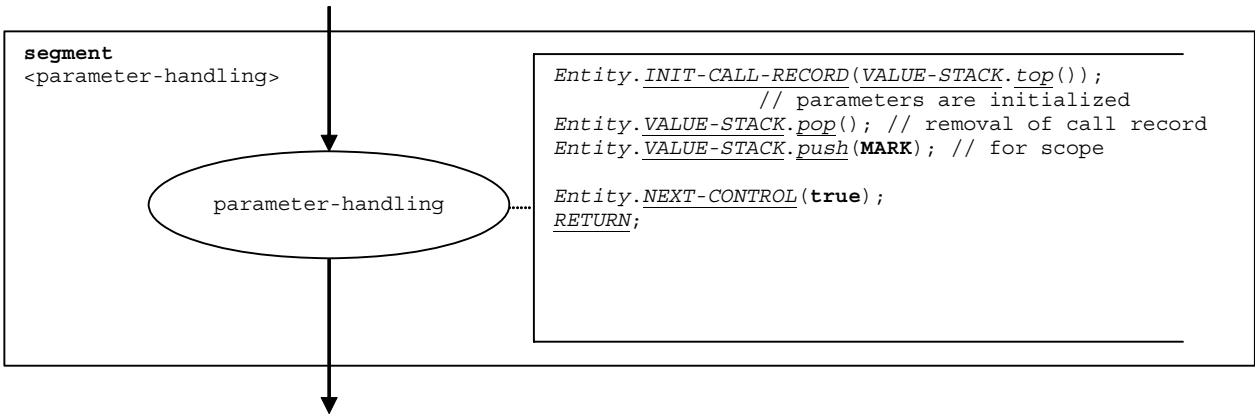


Figure 77 – Flow graph segment <parameter-handling>

9.22 Flow graph segment <statement-block>

The syntactical structure of a statement block is:

```
{ <statement1>; … ; <statementn> }
```

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.

NOTE 1 – The statement block is not an 'official' TTCN-3 concept. Statement blocks only occur as body of functions, altsteps, test cases and module control, and within compound statements, e.g., **`alt`**, **`if-else`** or **`do-while`**.

NOTE 2 – Receiving operations and altstep calls cannot appear in statement blocks, they are embedded in **`alt`** statements or **`call`** operations.

NOTE 3 – The operational semantics also handles operations and declarations like statements, i.e., they are allowed in statement blocks.

NOTE 4 – Some TTCN-3 functions, like e.g., **`system`** or **`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 `<statement-block>` in Figure 78 defines the execution of a statement block.

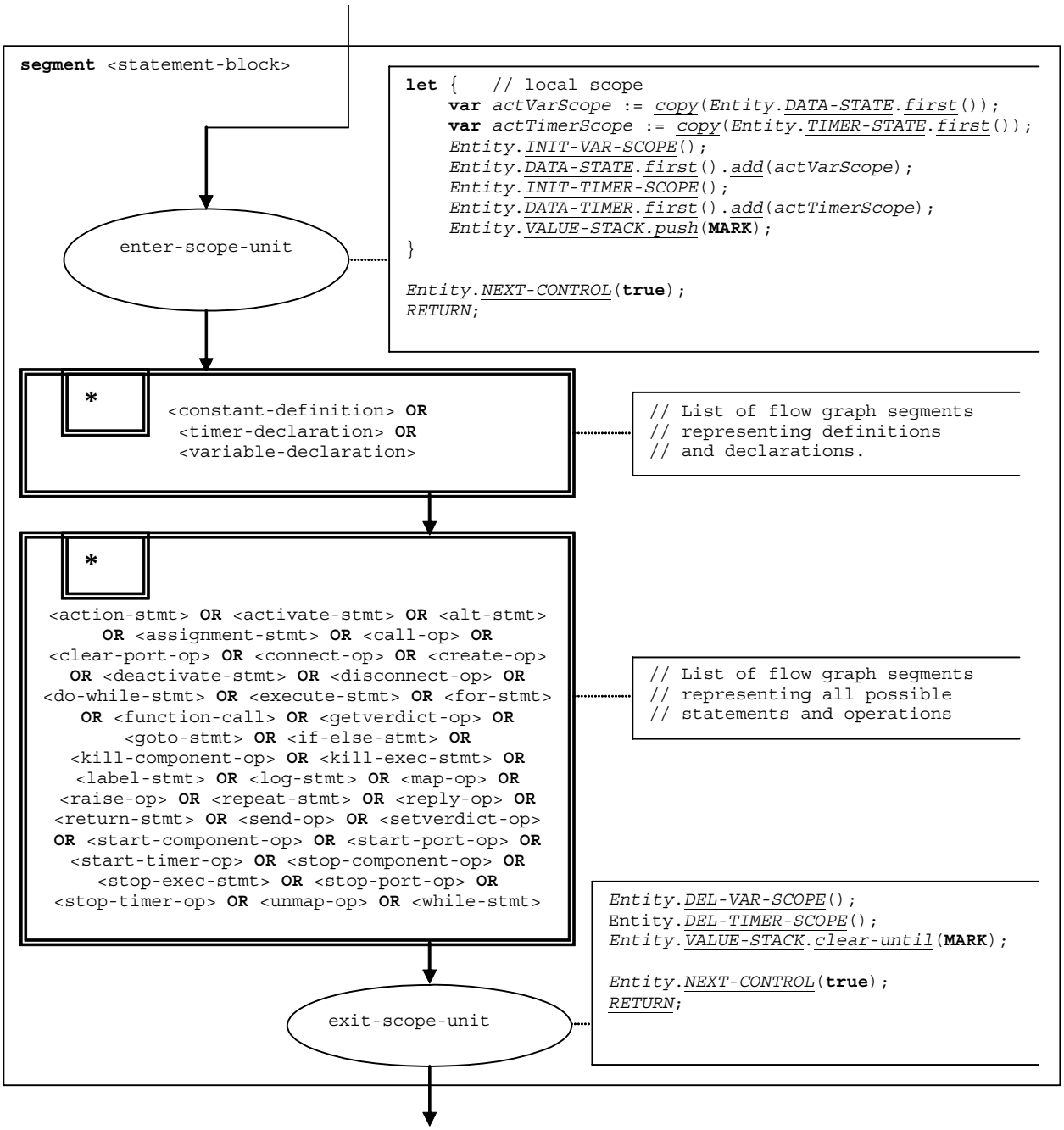


Figure 78 – Flow graph segment <statement-block>

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.

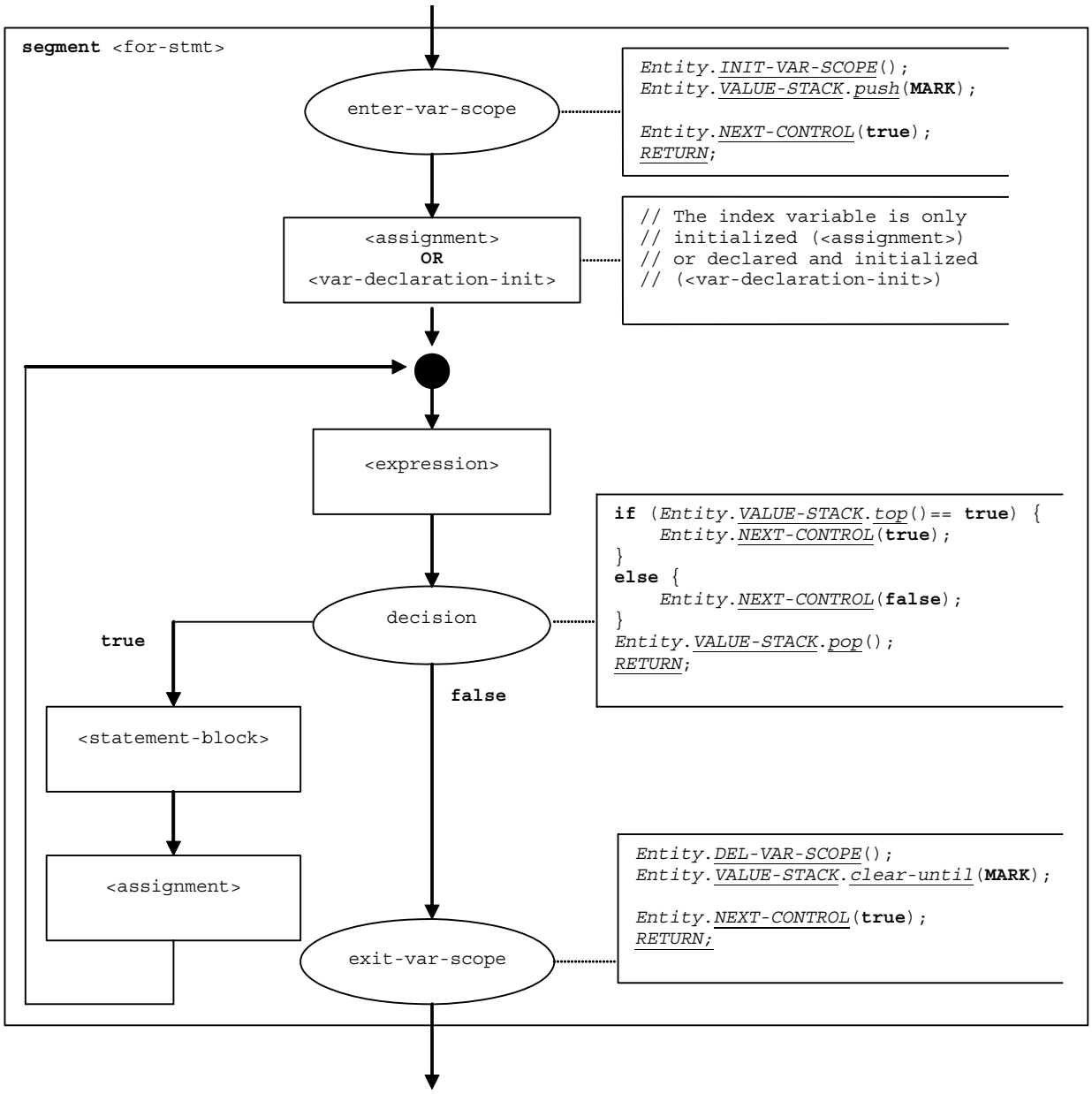


Figure 79 – Flow graph segment <for-stmt>

9.24 Function call

The syntactical structure of a function call is:

<function-name>([<act-par-desc$_1$>, … , <act-par-desc$_n$>])

The <function-name> denotes the name of a function and <act-par-desc$_1$>, … , <act-par-desc$_n$> describe 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$_1$> the corresponding formal parameter identifier <f-par-Id$_1$> is known, i.e., we can extend the syntactical structure above to:

<function-name>((<f-par-Id$_1$>,<act-par-desc$_1$>), … , (<f-par-Id$_n$>,<act-par-desc$_n$>))

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.

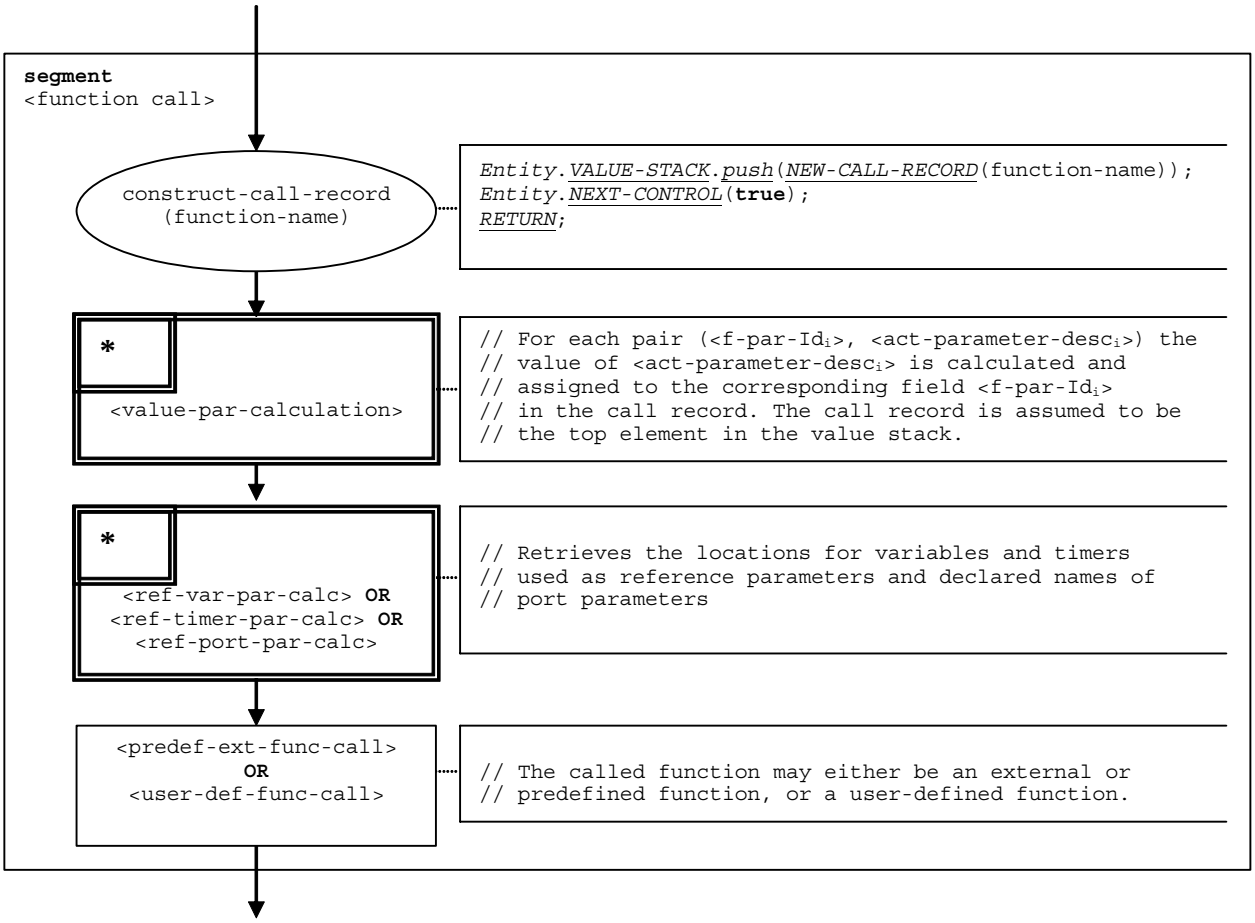


Figure 80 – Flow graph segment <function-call>

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:

(<f-par-Id$_i$>, <act-parameter-desc$_i$>)

has to be handled. <act-parameter-desc$_i$> has to be evaluated 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 `<value-par-calculation>` is shown in Figure 81.

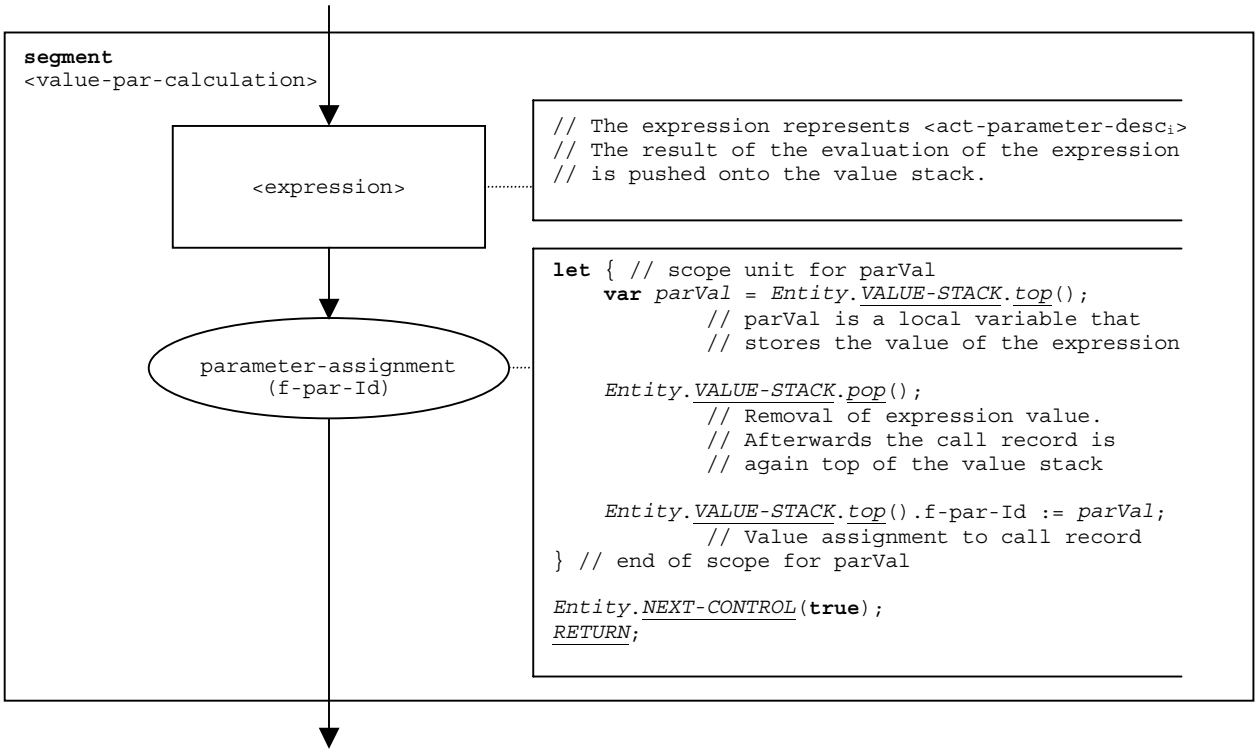


Figure 81 – Flow graph segment <value-par-calculation>

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:

(<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-var-calc>` is shown in Figure 82.

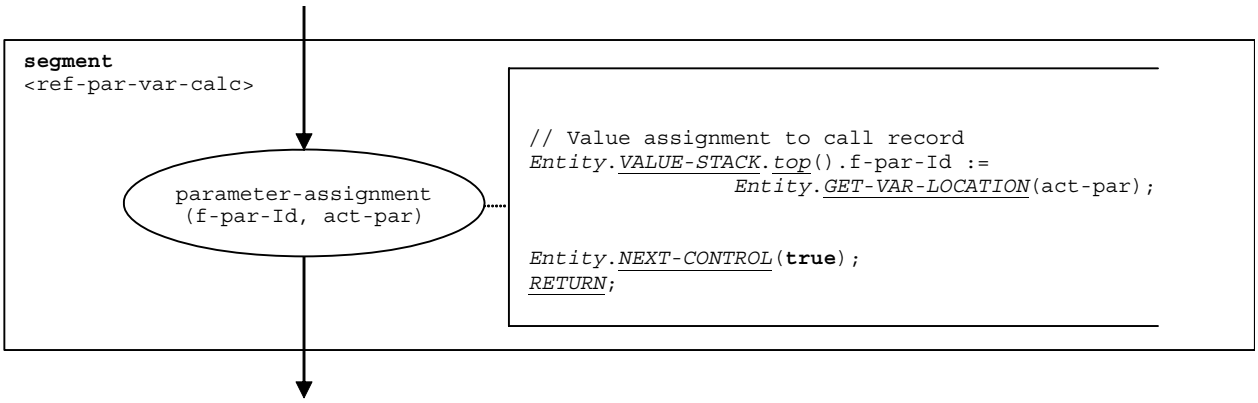


Figure 82 – Flow graph segment <ref-par-var-calc>

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.

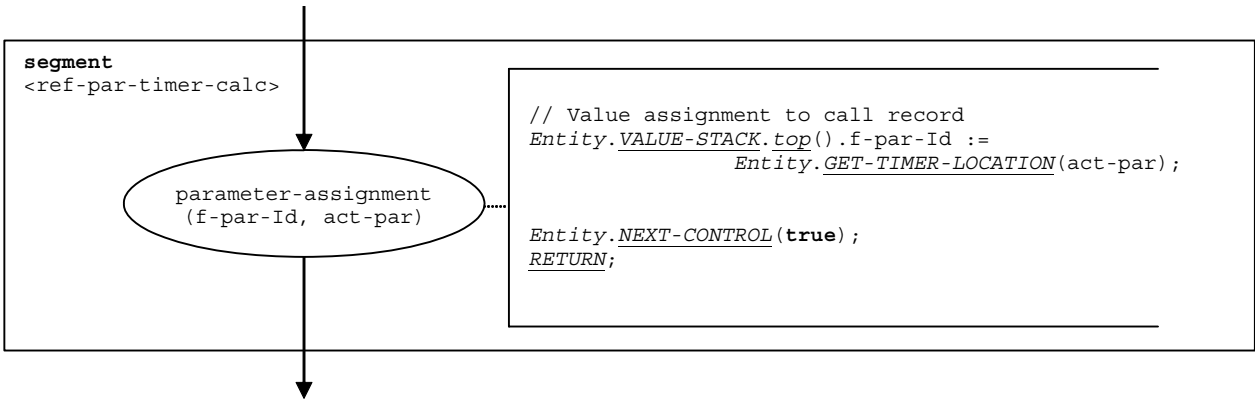


Figure 83 – Flow graph segment <ref-par-timer-calc>

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-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-port-calc>` is shown in Figure 83-a.

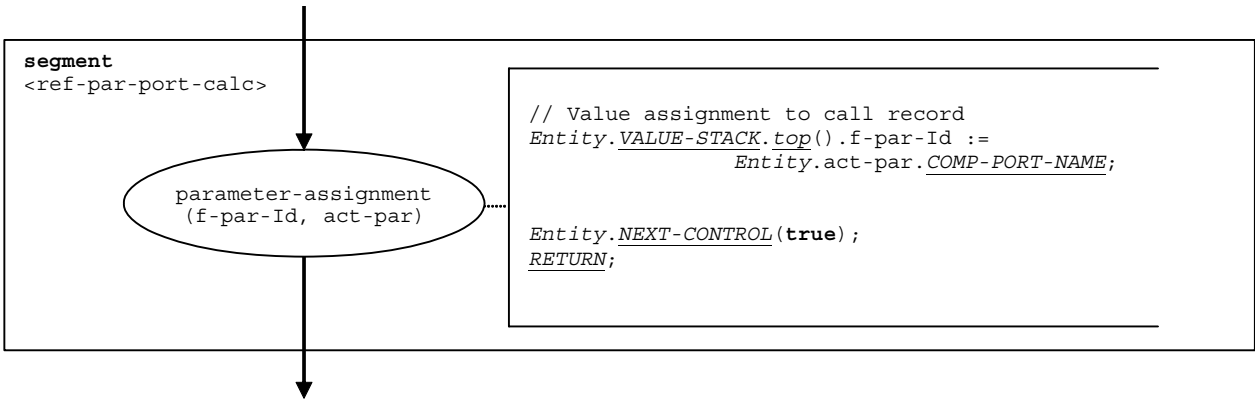


Figure 83-a – Flow graph segment <ref-par-port-calc>

9.24.4 Flow graph segment <user-def-func-call>

The flow graph segment `<user-def-func-call>` (Figure 84) describes the transfer of control to a called user-defined function.

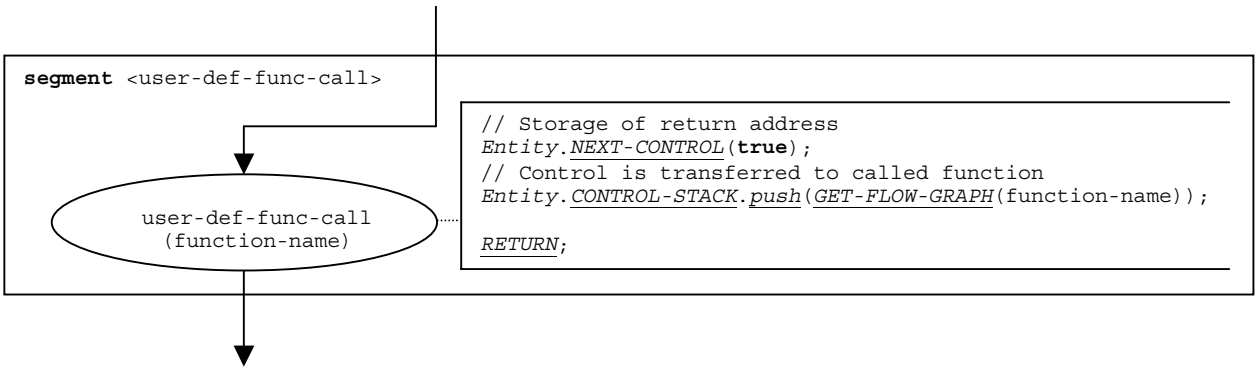


Figure 84 – Flow graph segment <user-def-func-call>

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.

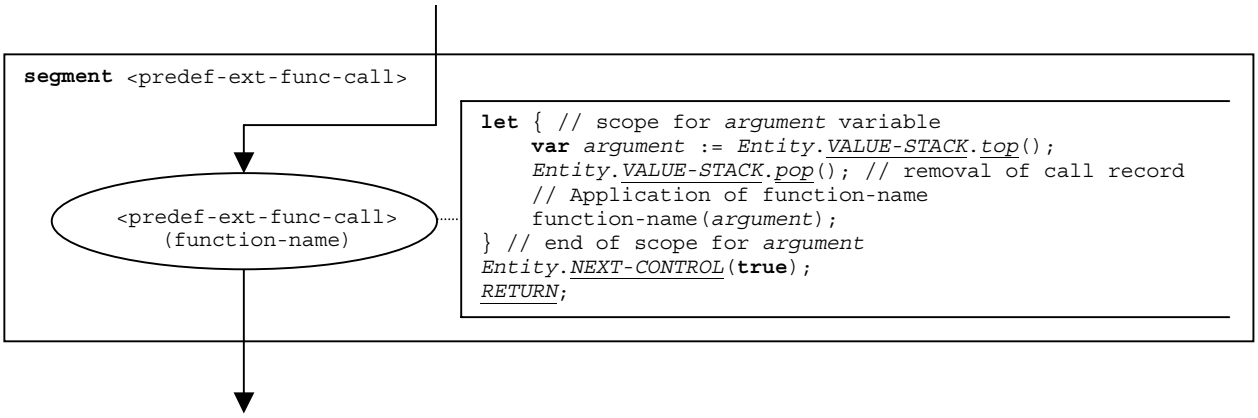


Figure 85 – Flow graph segment <predef-ext-func-call>

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).

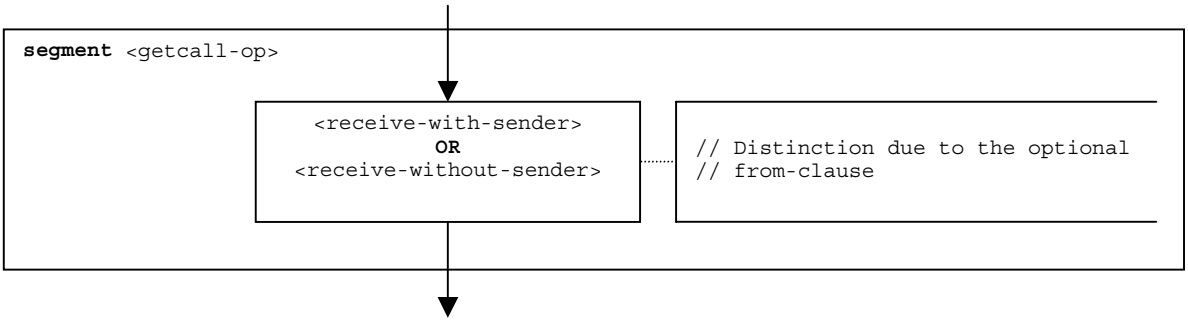


Figure 86 – Flow graph segment <getcall-op>

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).

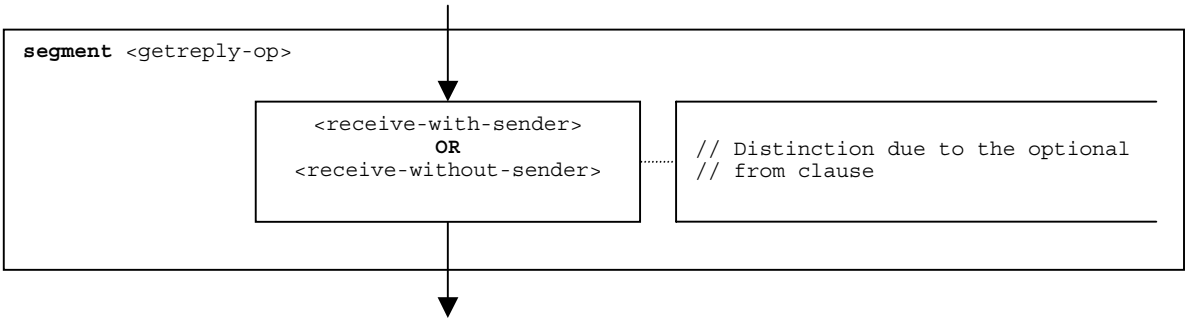


Figure 87 – Flow graph segment <getreply-op>

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.

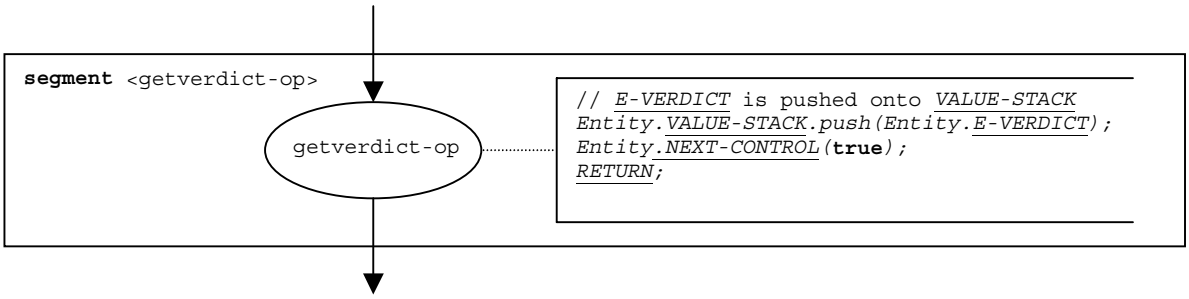


Figure 88 – Flow graph segment <getverdict-op>

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.

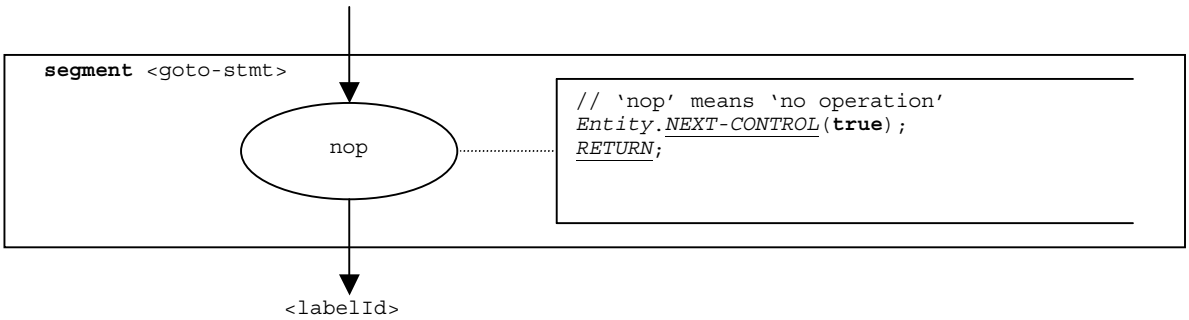


Figure 89 – Flow graph segment <goto-stmt>

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).

9.29 If-else statement

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

```
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.

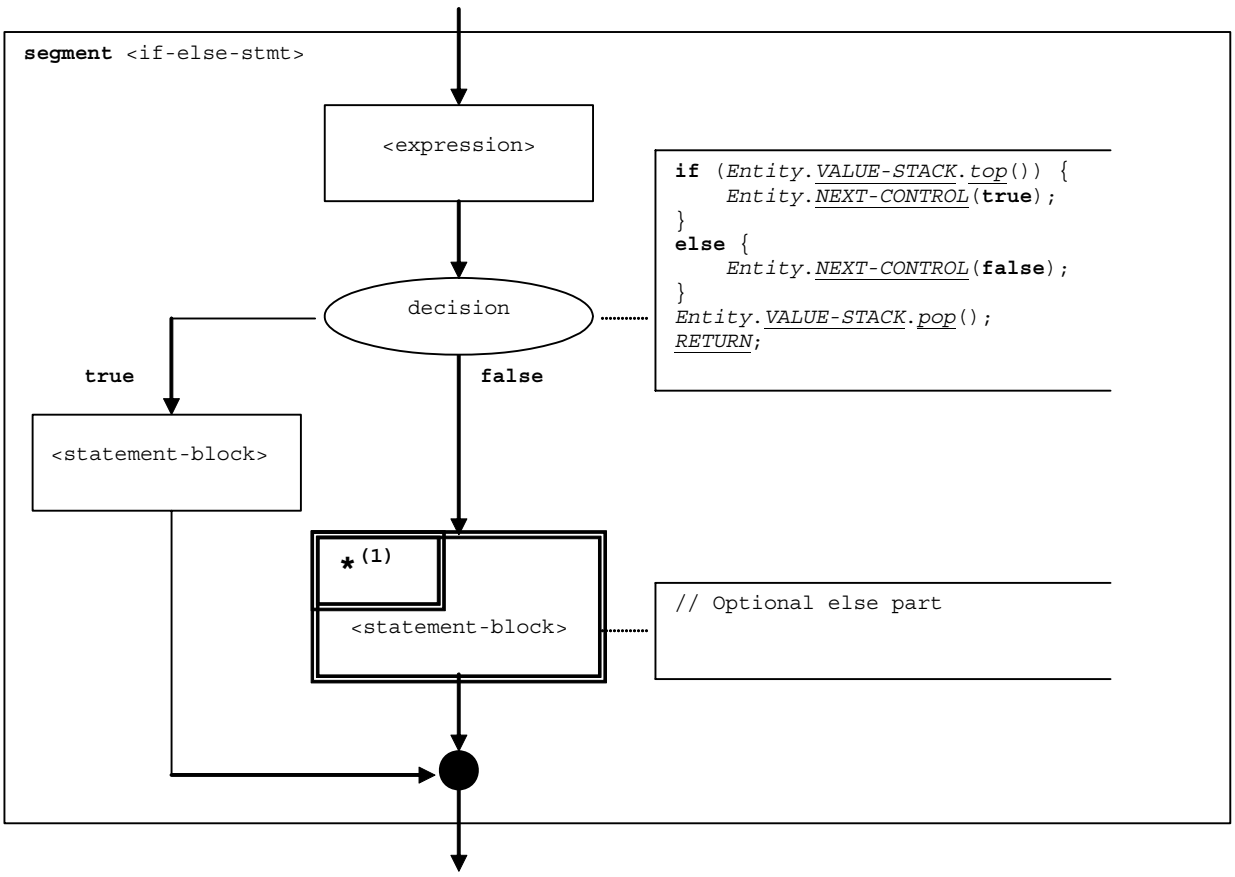


Figure 90 – Flow graph segment <if-else-stmt>

9.29a Kill component operation

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 90-a defines the execution of the **`kill`** component operation.

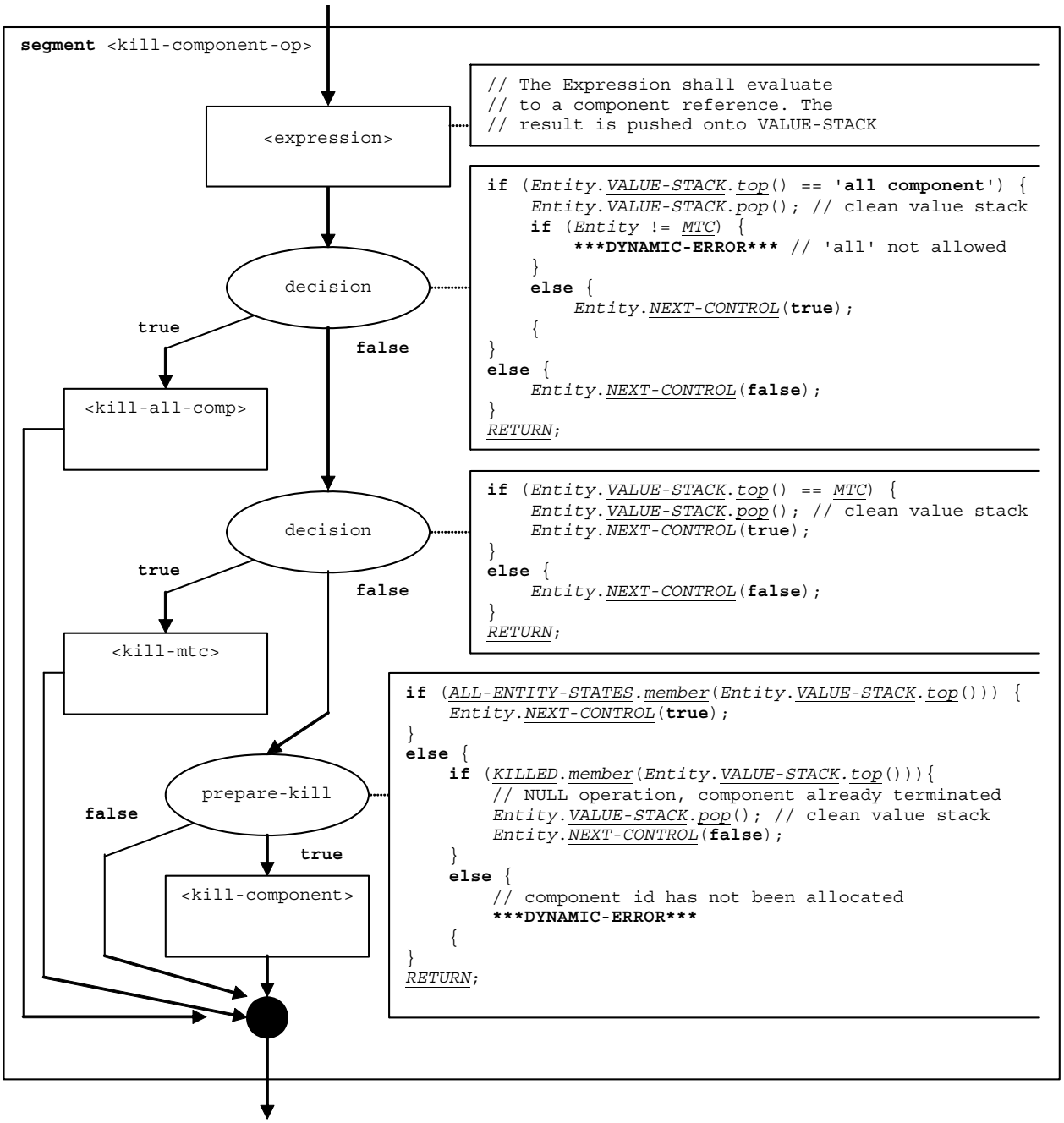


Figure 90-a – Flow graph segment <kill-component-op>

9.29a.1 Flow graph segment <kill-mtc>

The `<kill-mtc>` flow graph segment in Figure 90-b 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.

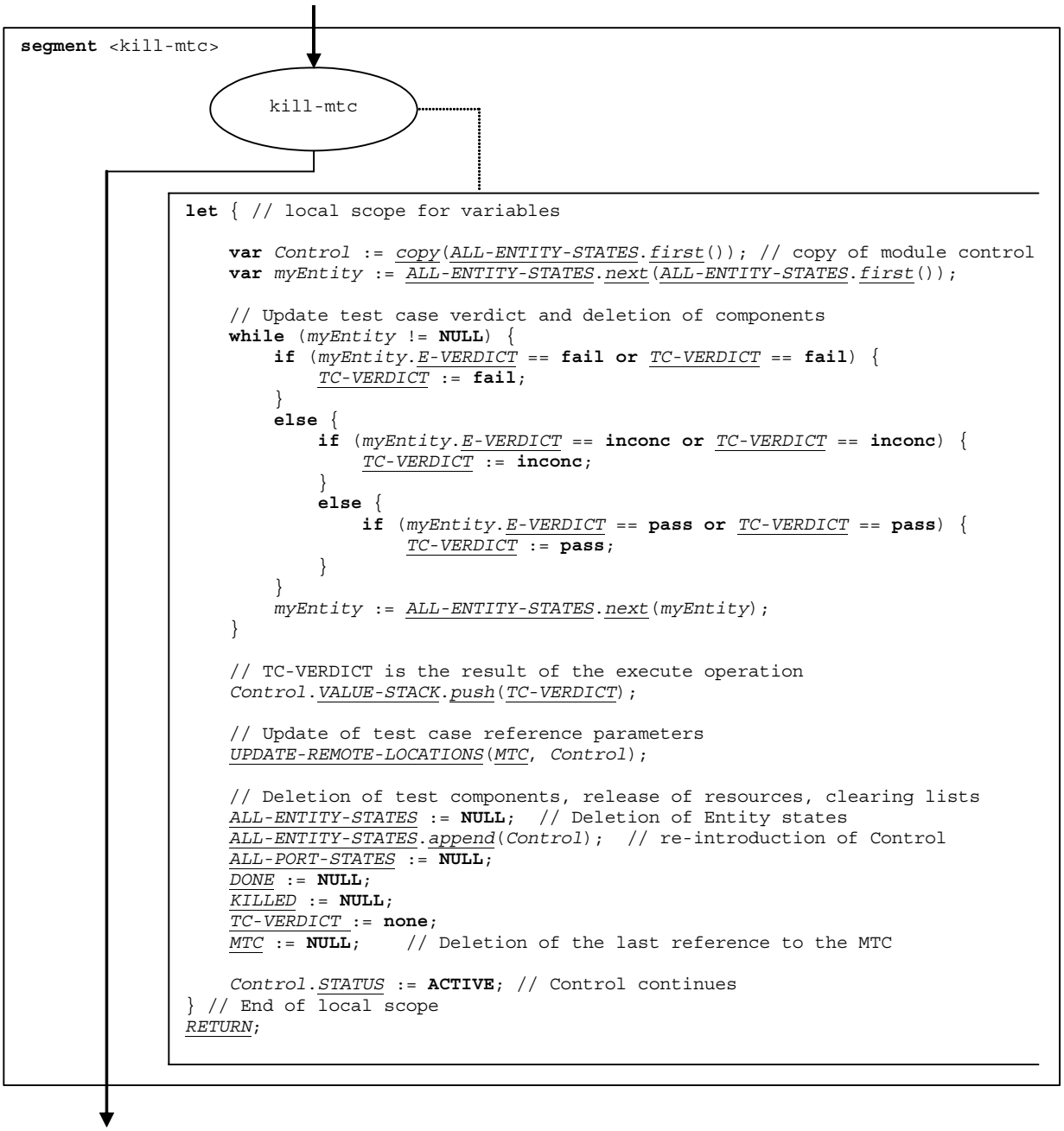


Figure 90-b – Flow graph segment <kill-mtc>

9.29a.2 Flow graph segment <kill-component>

The `<kill-component>` flow graph segment in Figure 90-c 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.

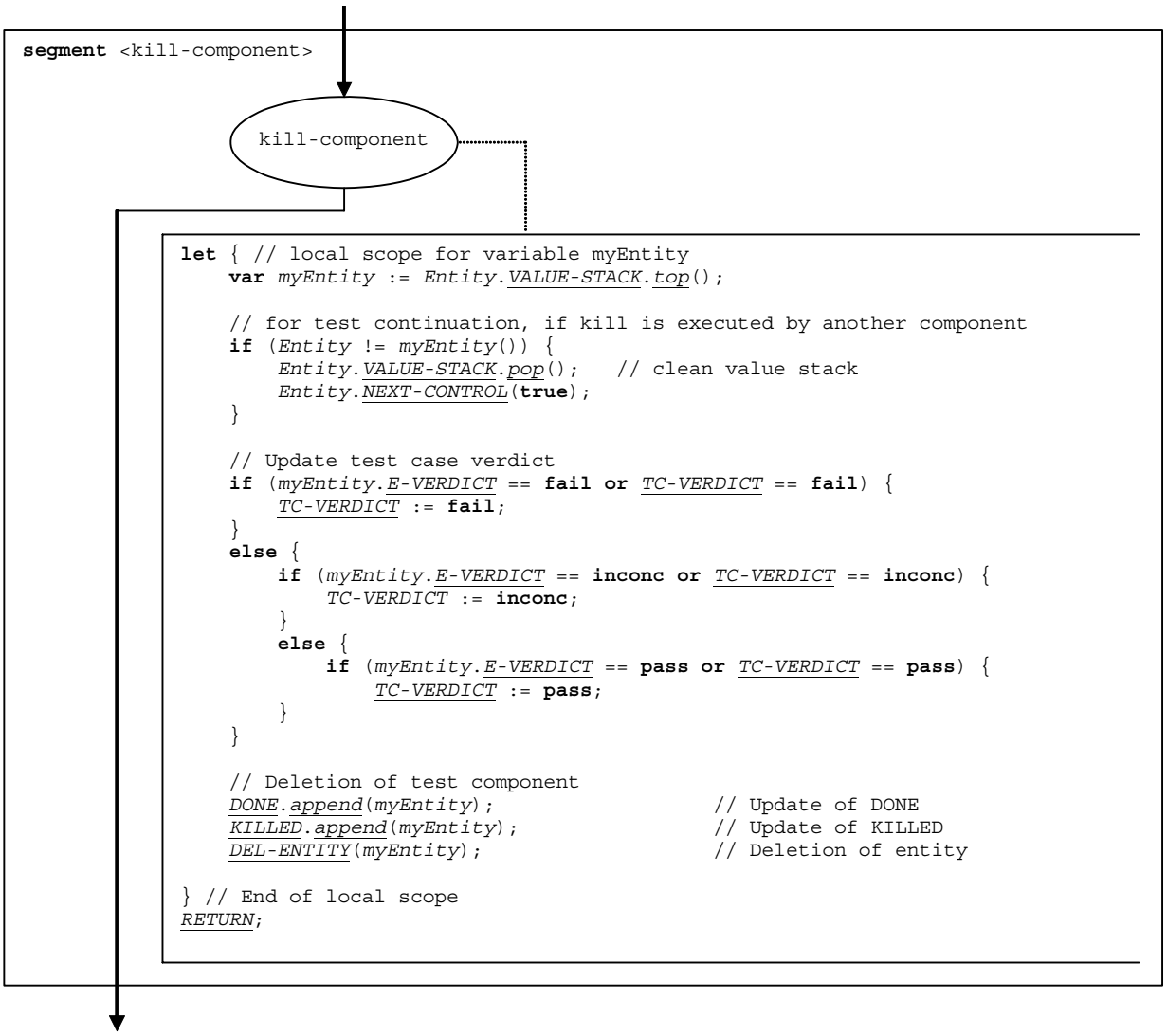


Figure 90-c – Flow graph segment <kill-component>

9.29a.3 Flow graph segment <kill-all-comp>

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

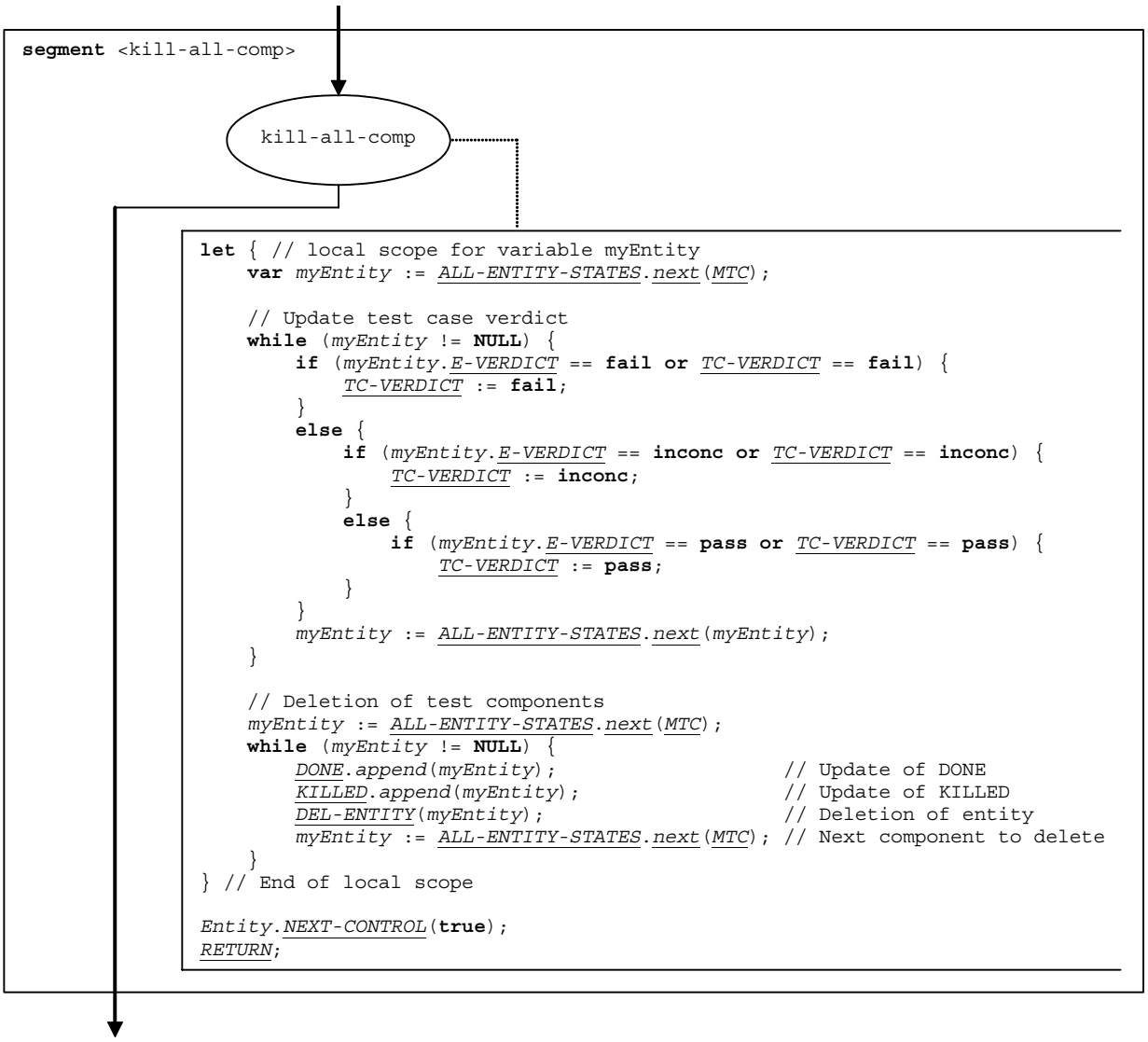


Figure 90-d – Flow graph segment <kill-all-comp>

9.29b Kill execution statement

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 90-e describes the execution of the kill statement.

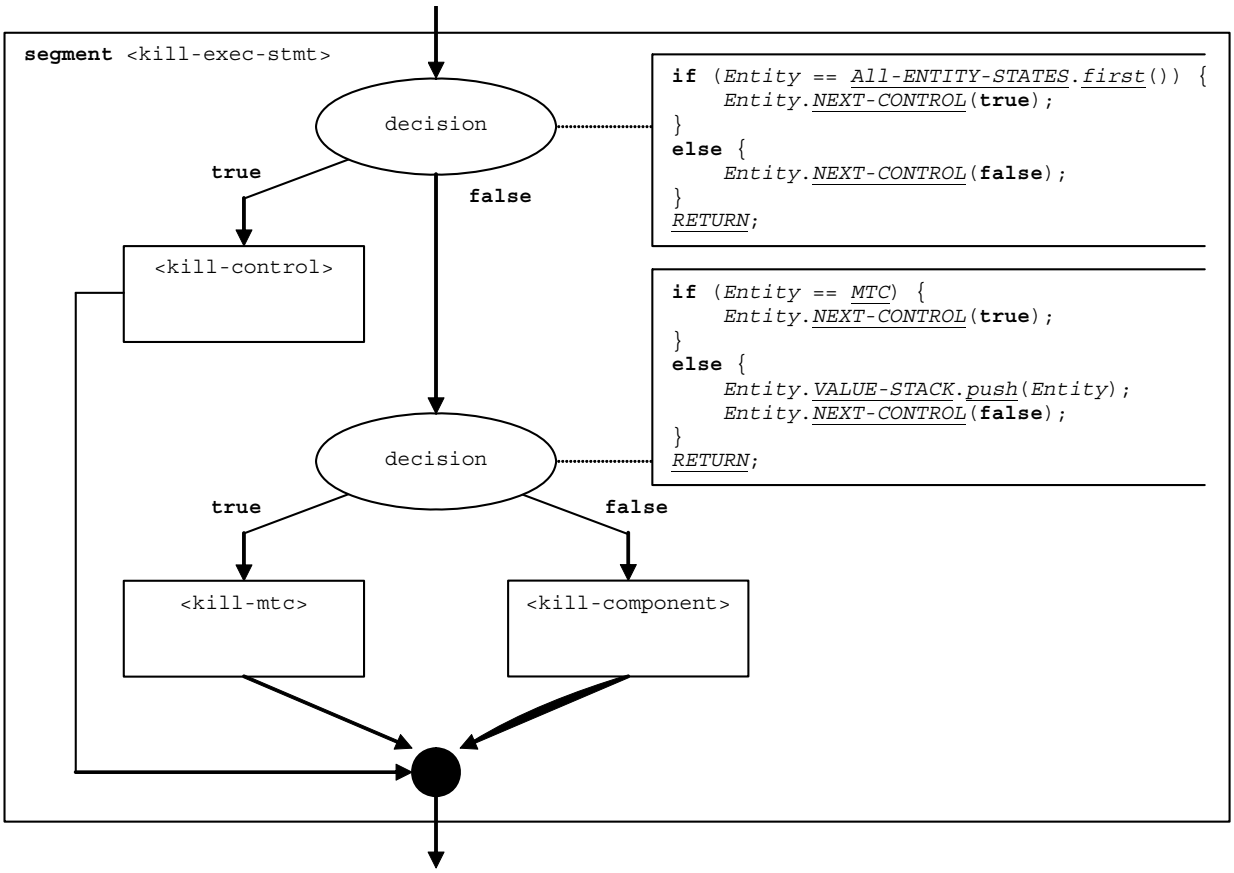


Figure 90-e – Flow graph segment <kill-exec-stmt>

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

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

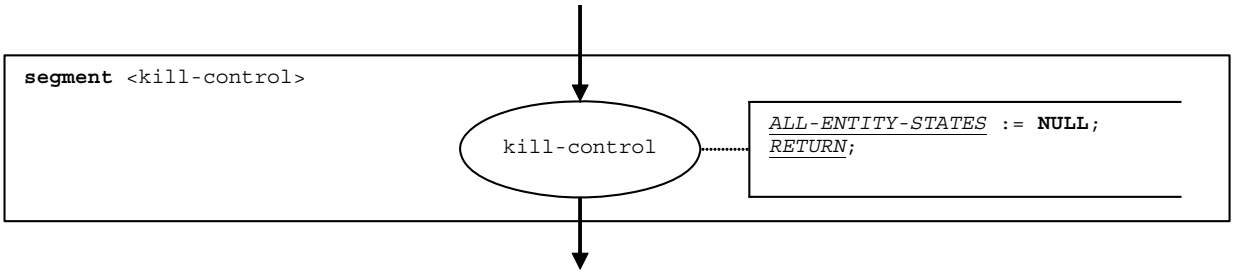


Figure 90-f – Flow graph segment <kill-control>

9.29c Killed component operation

The syntactical structure of the **killed** component operation is:

```
<component-expression>.killed
```

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 flow graph segment `<killed-op>` in Figure 90-g defines the execution of the **`killed`** component operation.

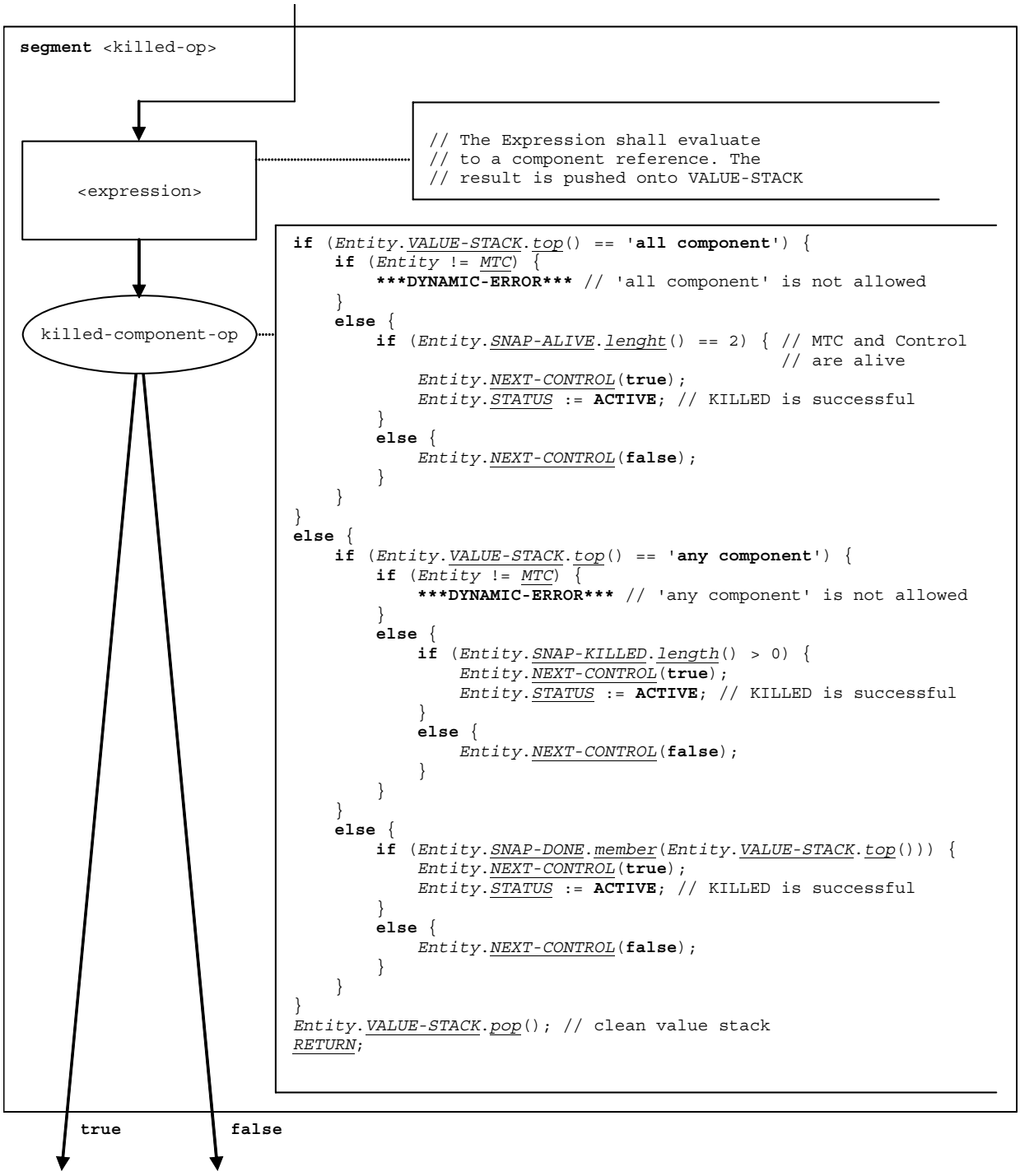


Figure 90-g – Flow graph segment <killed-op>

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.

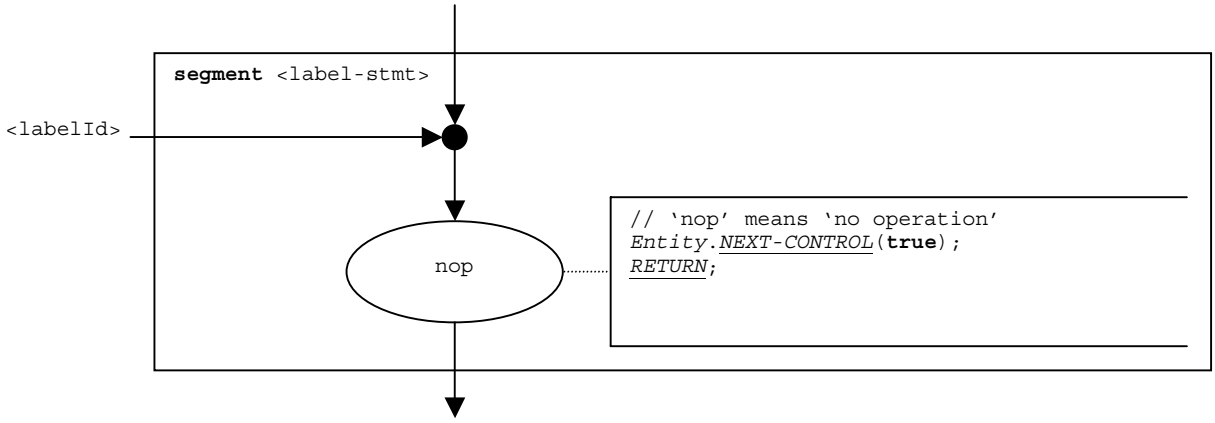


Figure 91 – Flow graph segment <label-stmt>

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).

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.

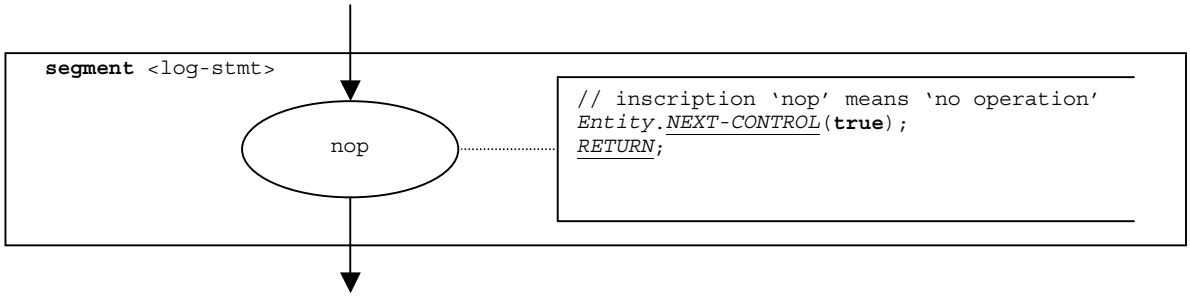


Figure 92 – Flow graph segment <log-stmt>

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.

9.32 Map operation

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

```
map(<component-expression>:<portId1>, system:<portId2>)
```

The identifiers <portId1> and <portId2> are considered to be port identifiers of the corresponding test component and test system interface. The components 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.

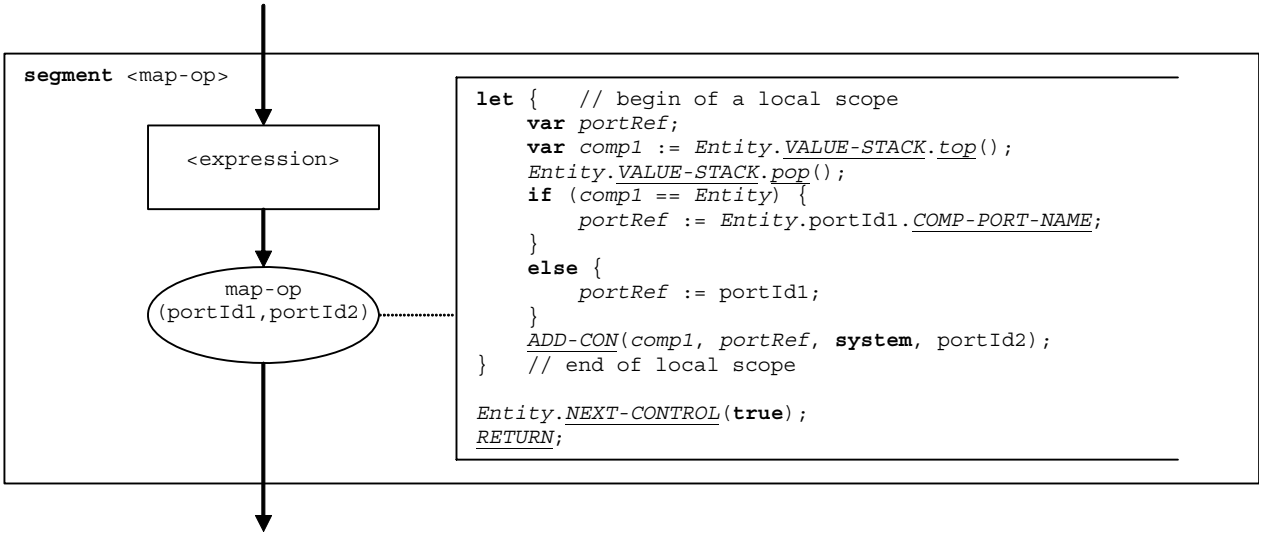


Figure 93 – Flow graph segment <map-op>

9.33 Mtc operation

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

```
mtc
```

The flow graph segment <mtc-op> in Figure 94 defines the execution of the **mtc** operation.

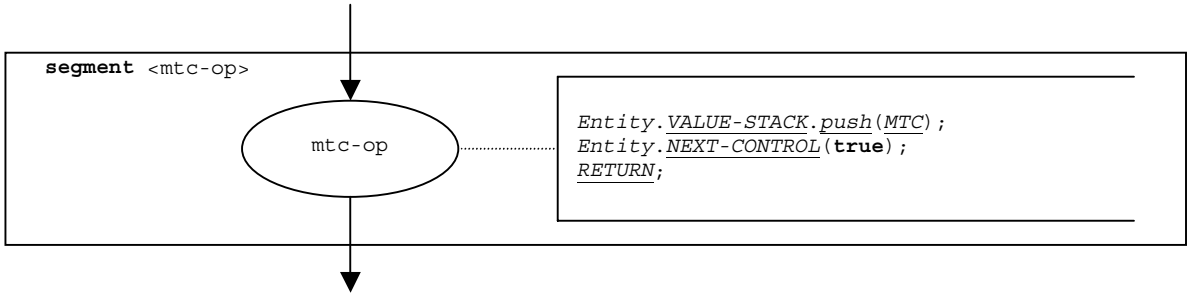


Figure 94 – Flow graph segment <mtc-op>

9.34 Port declaration

The syntactical structure of a **port** declaration is:

```
<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 *PORT-NAME* and *COMP-PORT-NAME* are equal. The flow graph segment <port-declaration> in Figure 95 defines the execution of a port declaration.

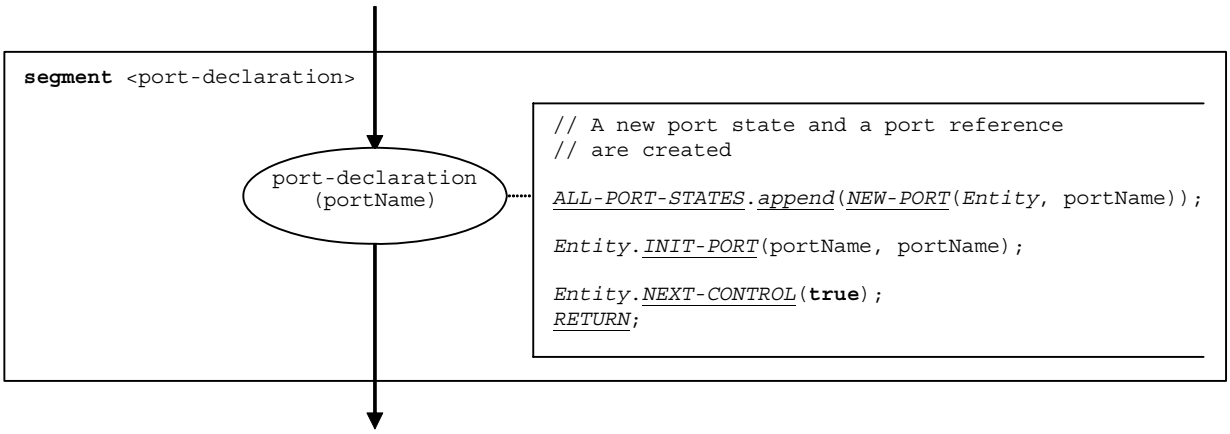


Figure 95 – Flow graph segment <port-declaration>

9.35 Raise operation

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

```
<portId>.raise (<exceptSpec>) [to <receiver-spec>]
```

The optional `<receiver-spec>` in the **to** clause refers to the receivers of the exception. 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 flow graph segment `<raise-op>` in Figure 96 defines the execution of a **raise** operation.

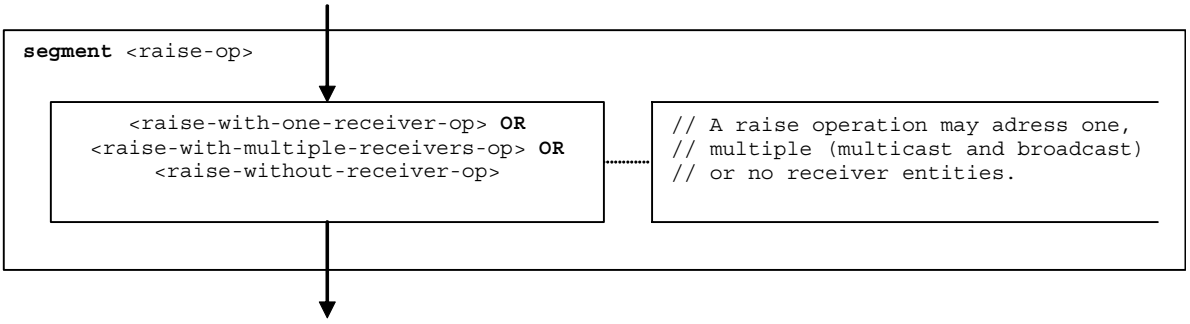


Figure 96 – Flow graph segment <raise-op>

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.

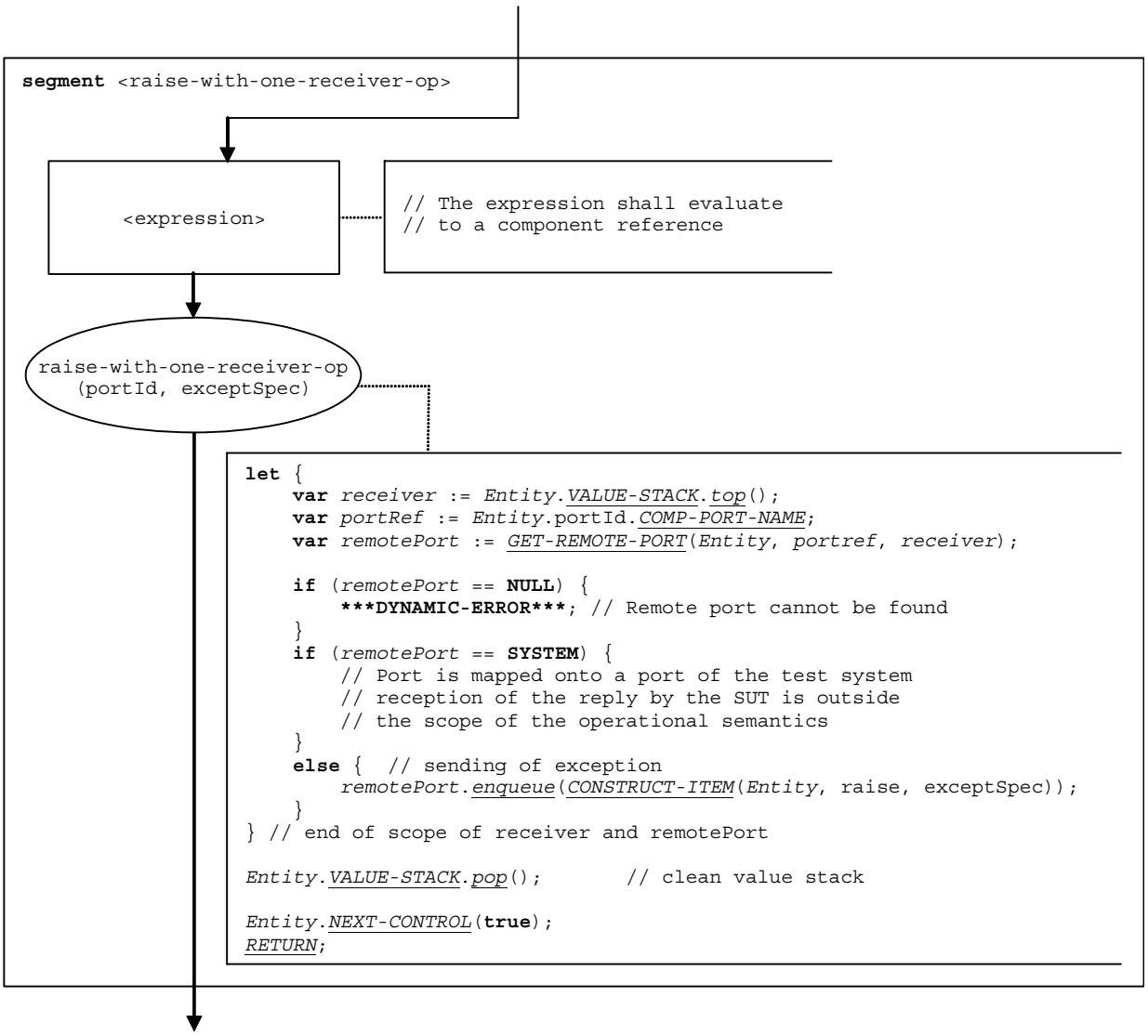


Figure 97 – Flow graph segment <raise-with-one-receiver-op>

9.35.1a Flow graph segment <raise-with-multiple-receivers-op>

The flow graph segment `<raise-with-multiple-receivers-op>` in Figure 97-a 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.

The component references 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 97-a. The `number` parameter is 1 in case of broadcast communication, i.e., the keyword **`all component`** is top element in the value stack.

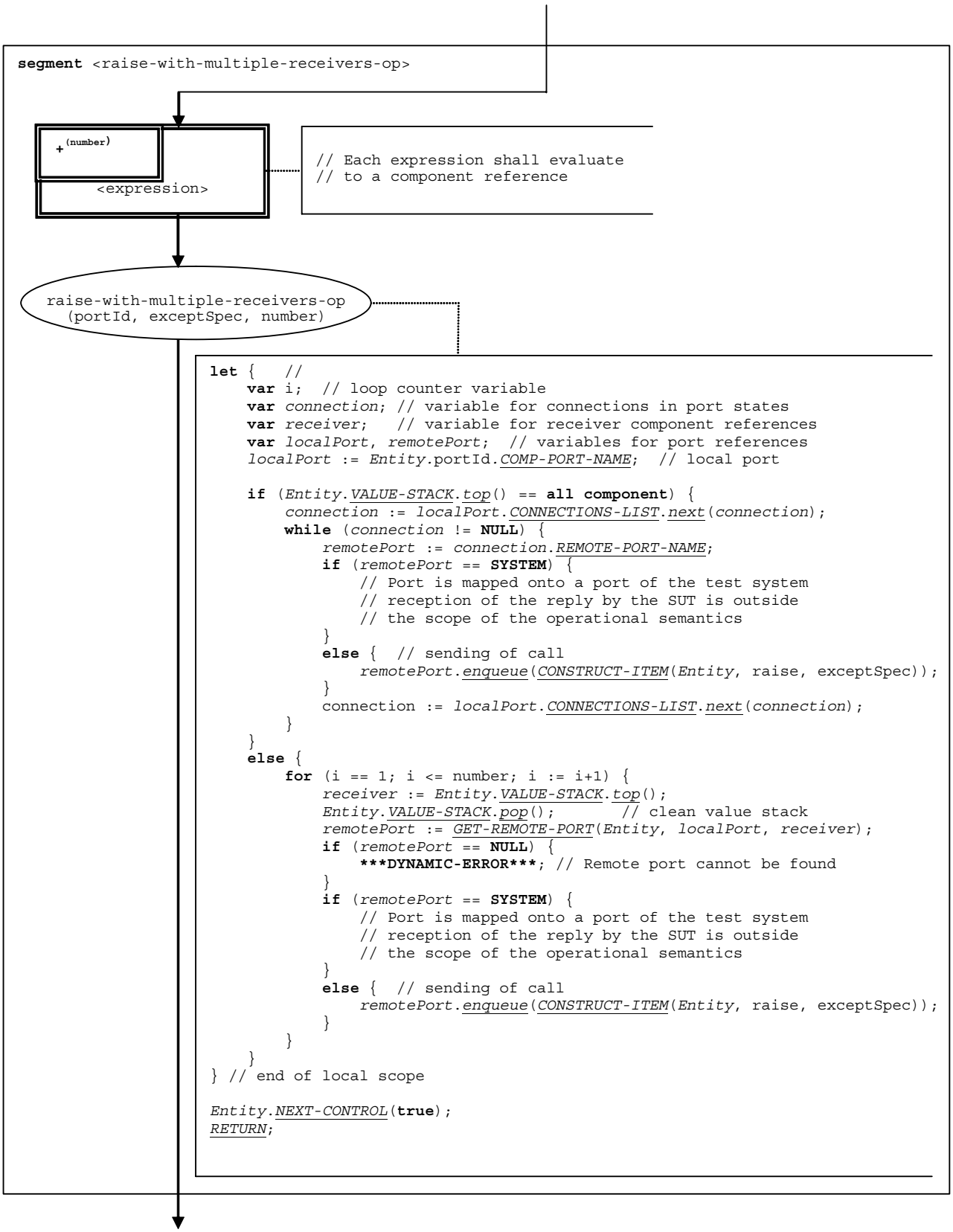


Figure 97-a – Flow graph segment <raise-with-multiple-receivers-op>

9.35.2 Flow graph segment <raise-without-receiver-op>

The flow graph segment `<raise-without-receiver-op>` in Figure 98 defines the execution of a **`raise`** operation without **`to`**-clause.

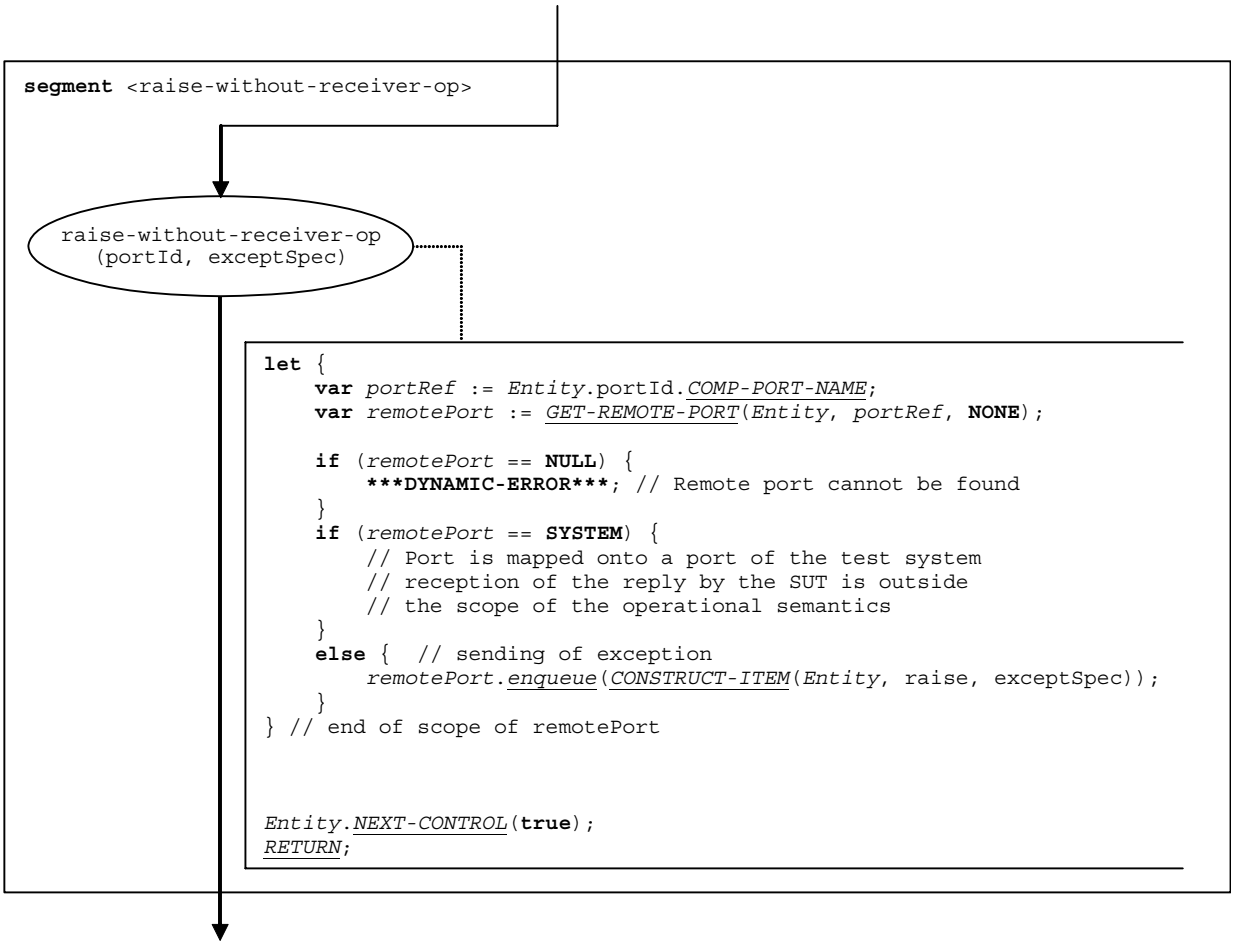


Figure 98 – Flow graph segment <raise-without-receiver-op>

9.36 Read timer operation

The syntactical structure of the **`read`** timer operation is:

```
<timerId>.read
```

The flow graph segment `<read-timer-op>` in Figure 99 defines the execution of the **`read`** timer operation.

The **`read`** 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 the **`read`** timer operation is based on the actual snapshot, i.e., the *SNAP-STATUS* and *SNAP-VALUE* entries of the timer binding, in all other cases, the *STATUS*, *ACT-DURATION* and *TIME-LEFT* entries of the timer binding determine the result of the operation.

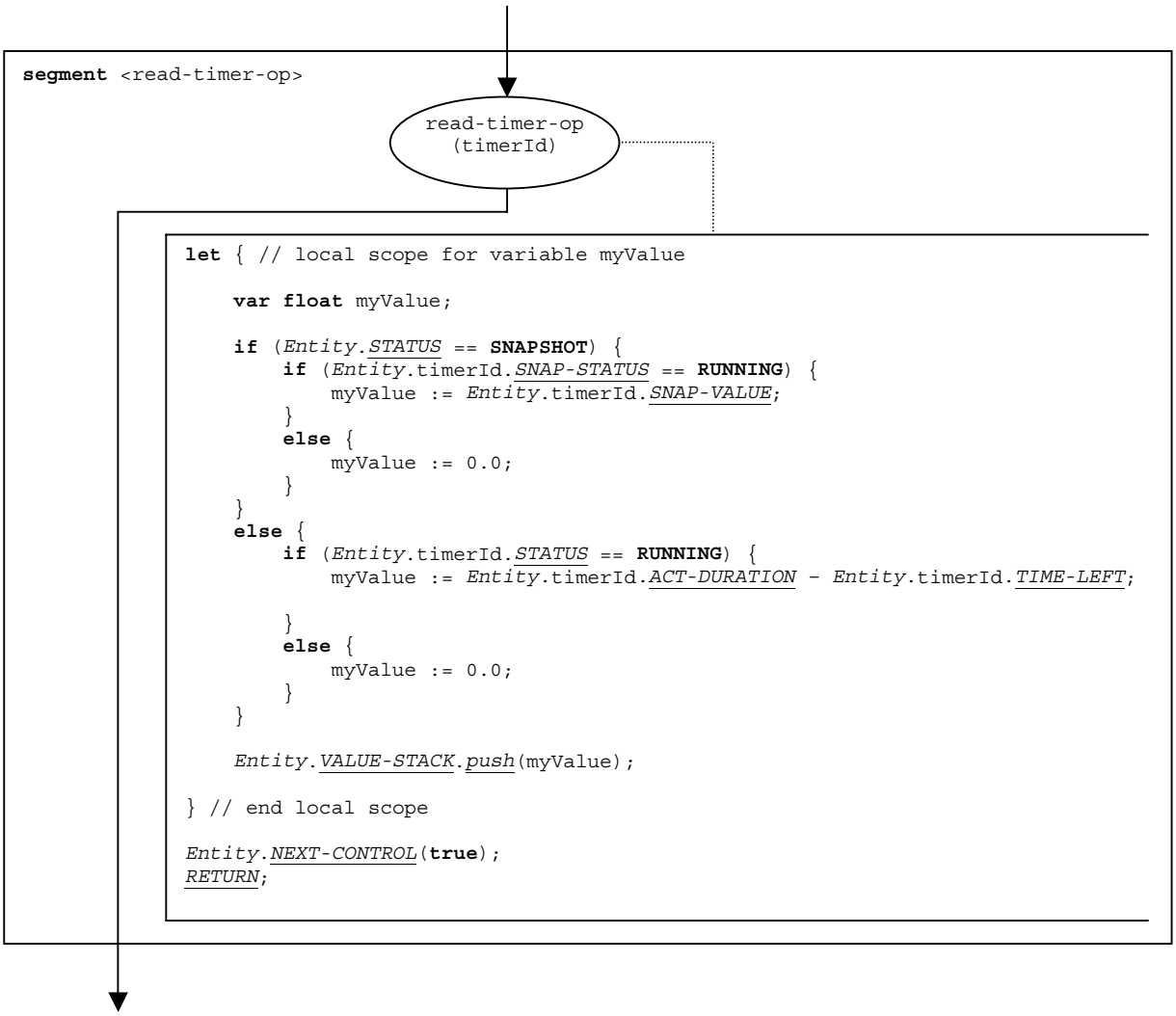


Figure 99 – Flow graph segment <read-timer-op>

9.37 Receive operation

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.

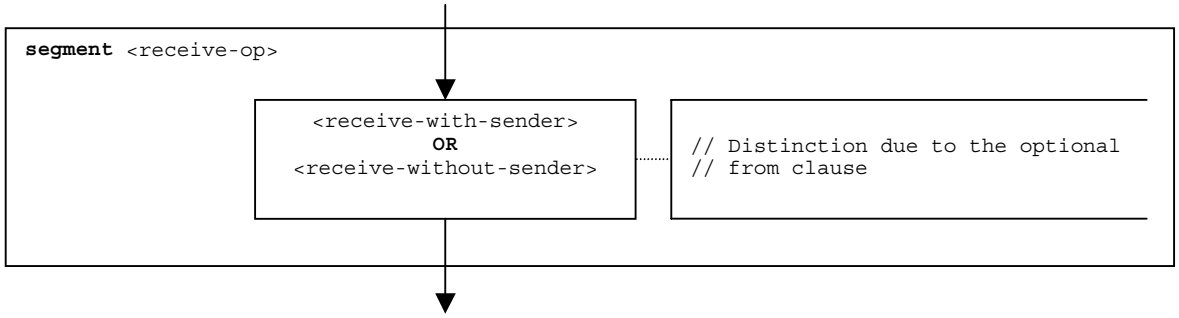


Figure 100 – Flow graph segment <receive-op>

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.

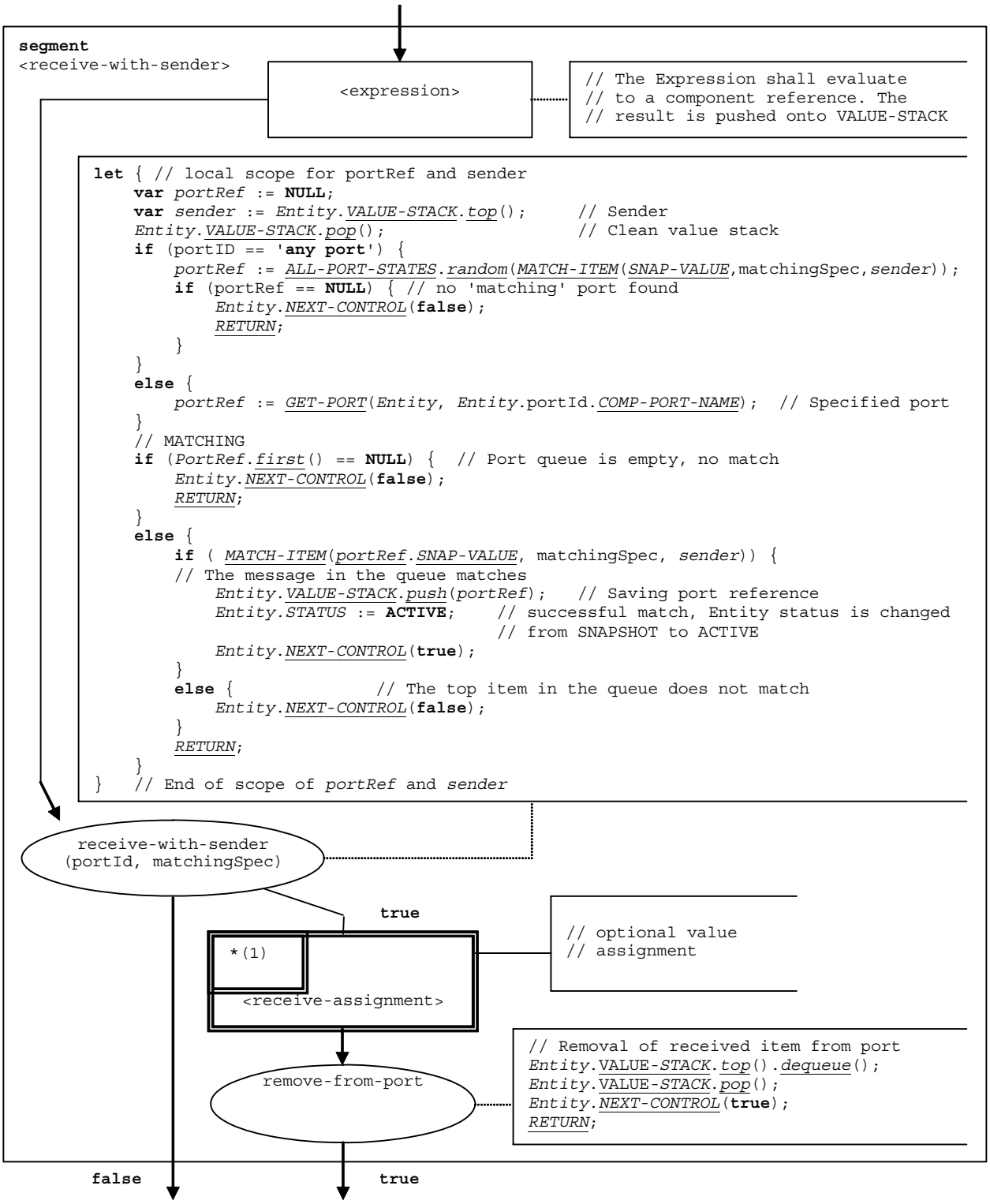


Figure 101 – Flow graph segment <receive-with-sender>

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.

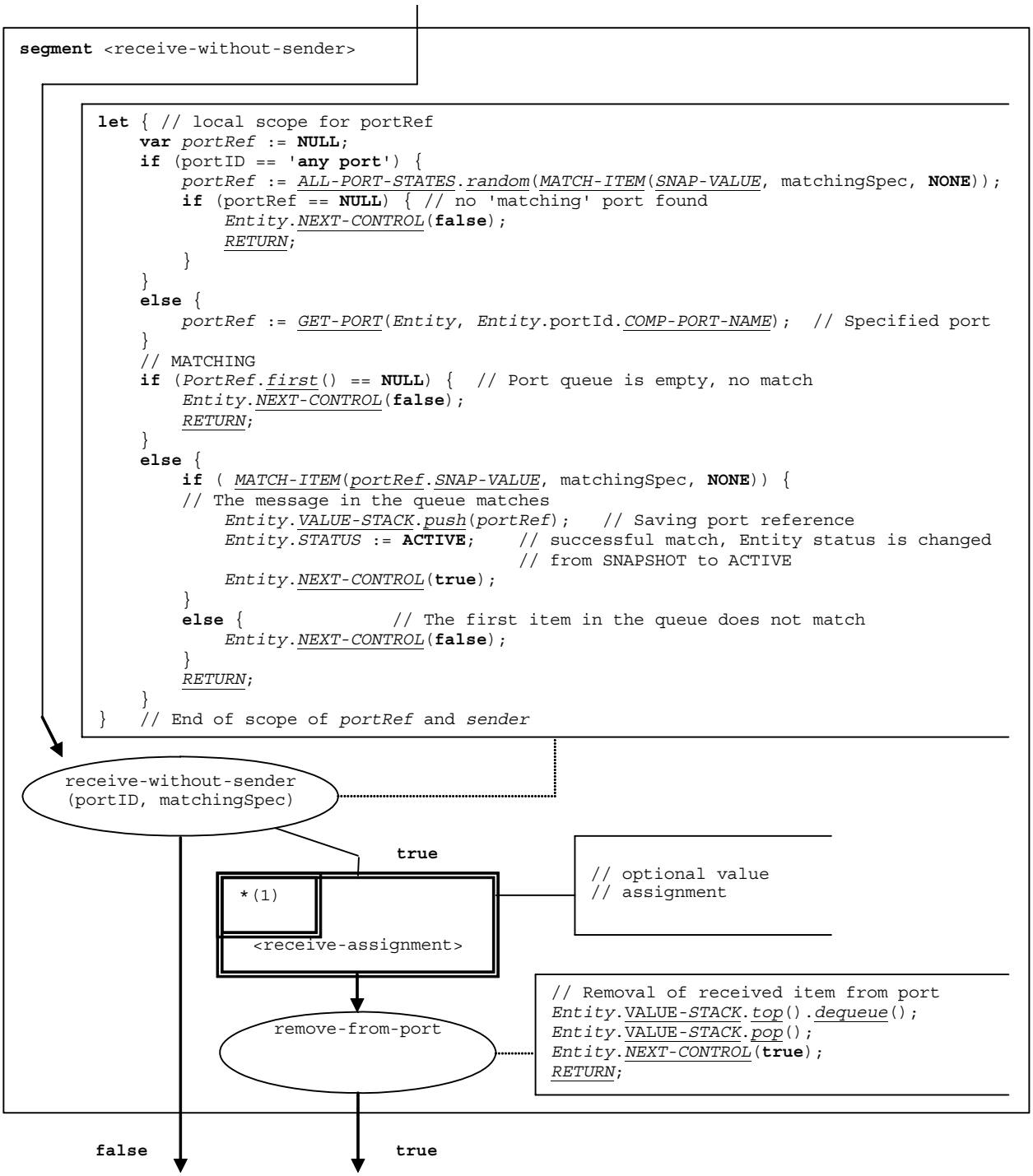


Figure 102 – Flow graph segment <receive-without-sender>

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.

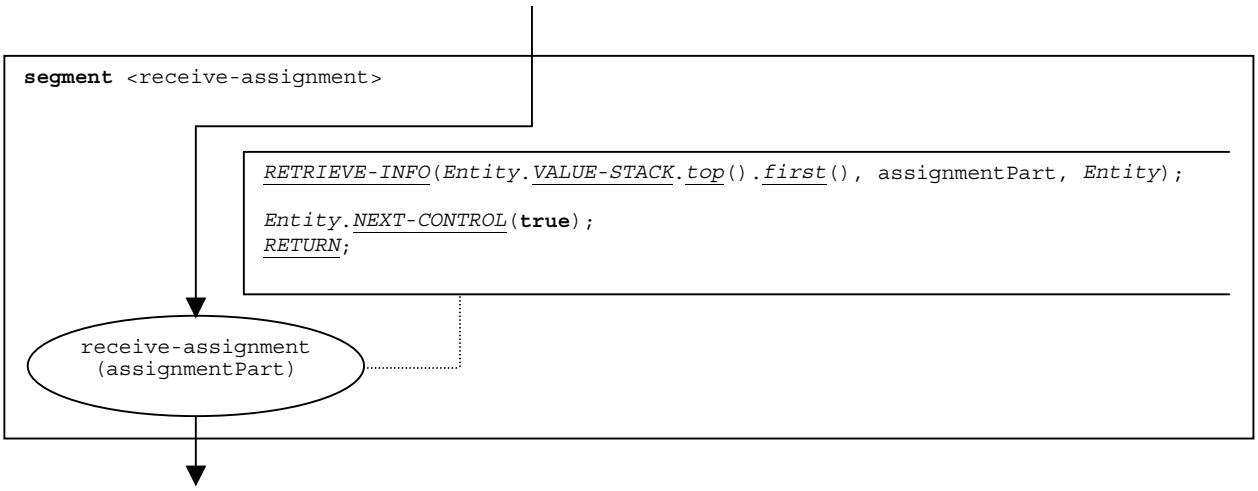


Figure 103 – Flow graph segment <receive-assignment>

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.

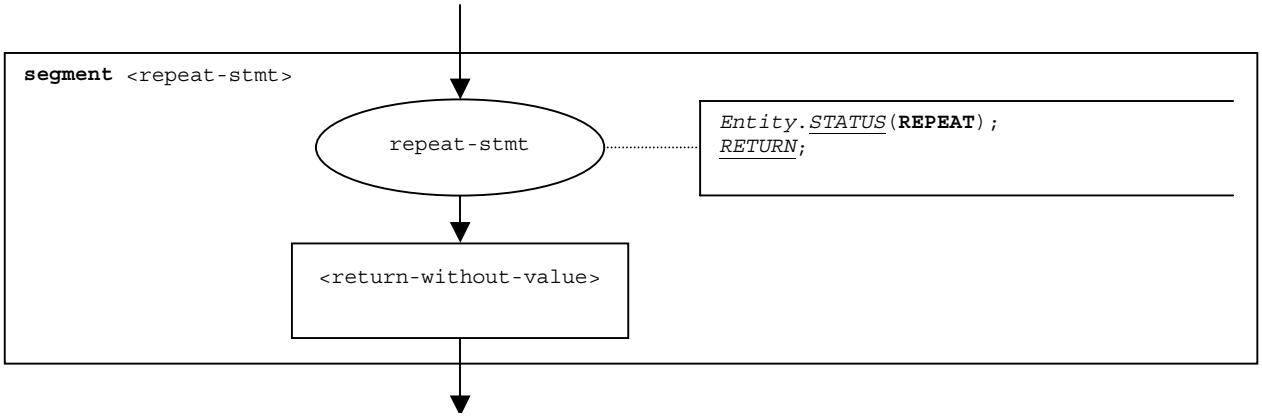


Figure 104 – Flow graph segment <repeat-stmt>

9.39 Reply operation

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

```
<portId>.reply (<replySpec>) [to <receiver-spec>]
```

The optional <receiver-spec> in the **to** clause refers to the receivers of the reply. 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 flow graph segment <reply-op> in Figure 105 defines the execution of a **reply** operation.

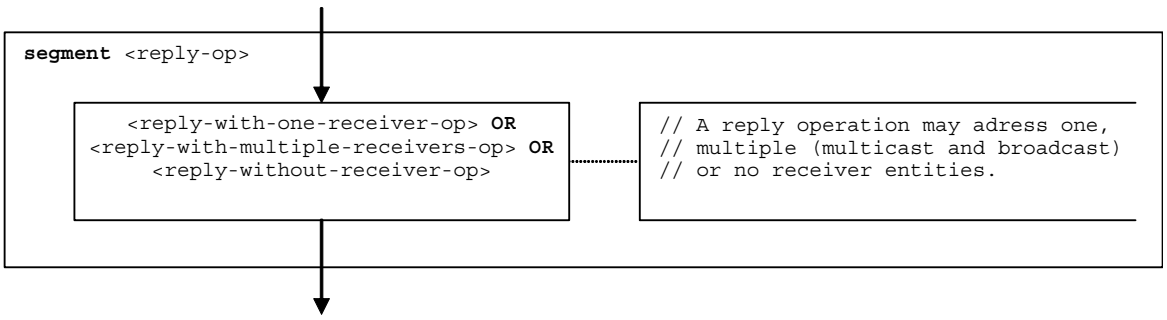


Figure 105 – Flow graph segment <reply-op>

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.

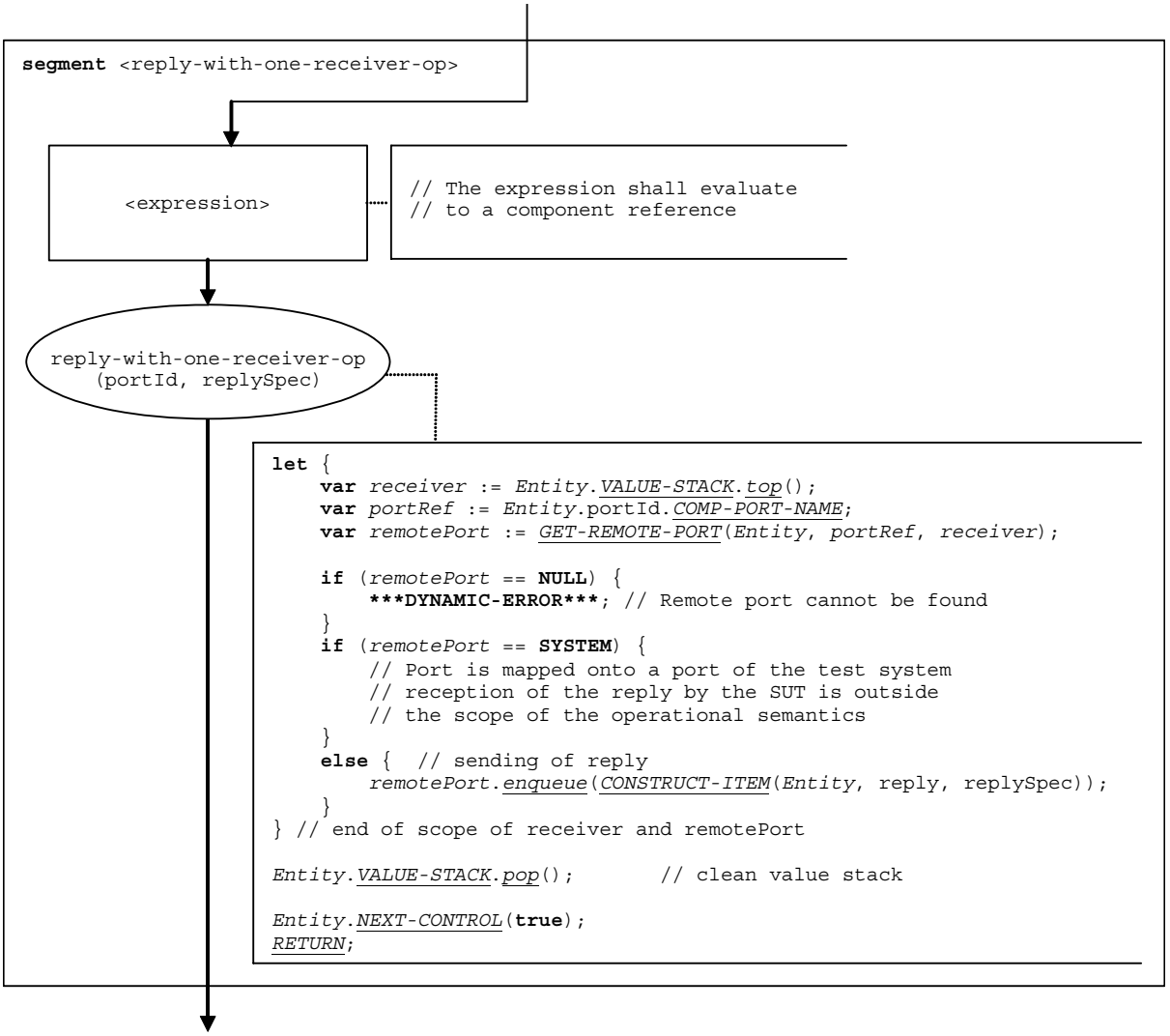


Figure 106 – Flow graph segment <reply-with-one-receiver-op>

9.39.1a Flow graph segment <reply-with-multiple-receivers-op>

The flow graph segment `<reply-with-multiple-receivers-op>` in Figure 106-a 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.

The component references 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 `reply-with-multiple-receivers-op` in Figure 106-a. The `number` parameter is 1 in case of broadcast communication, i.e., the keyword **`all component`** is top element in the value stack.

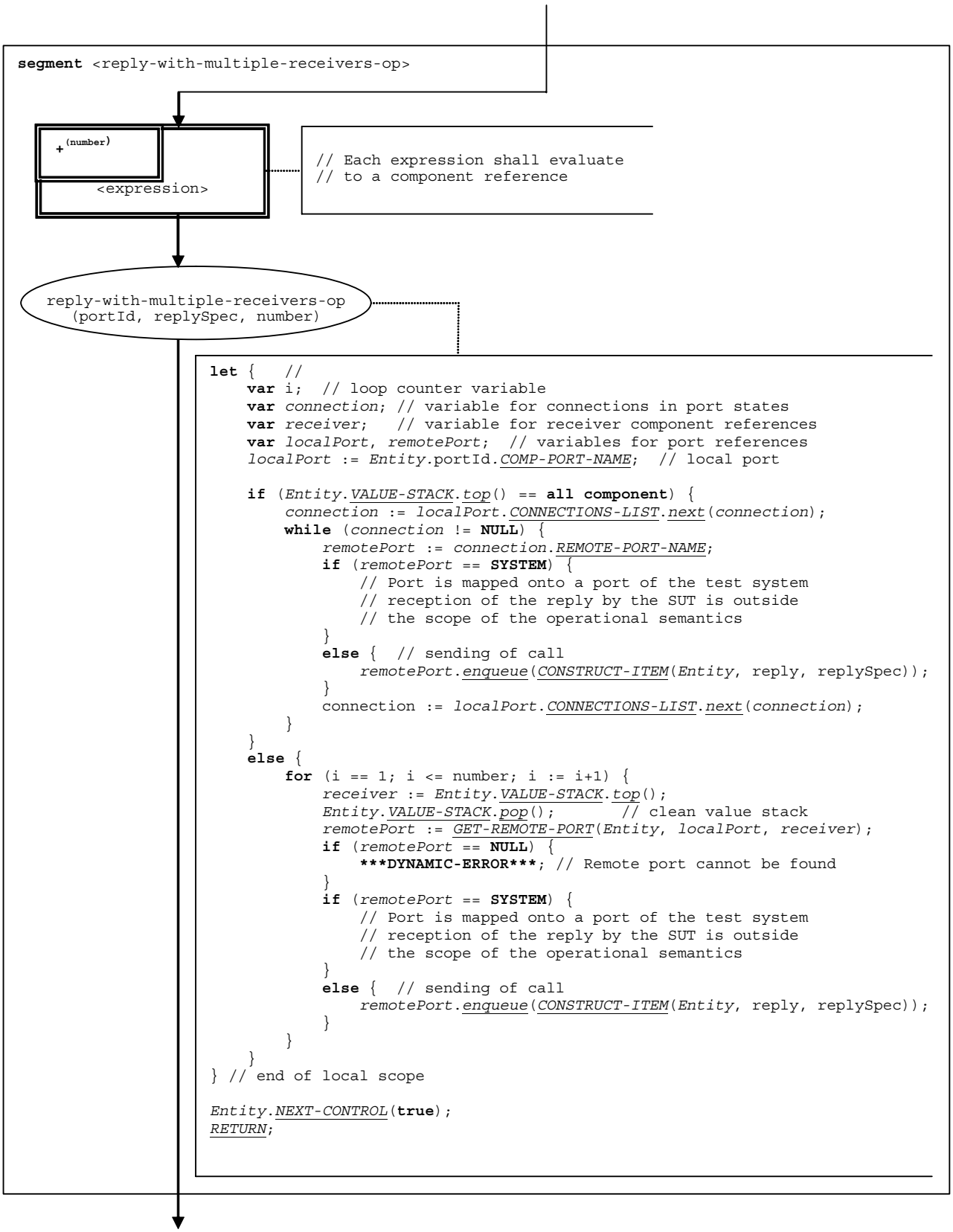


Figure 106-a – Flow graph segment <reply-with-multiple-receivers-op>

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.

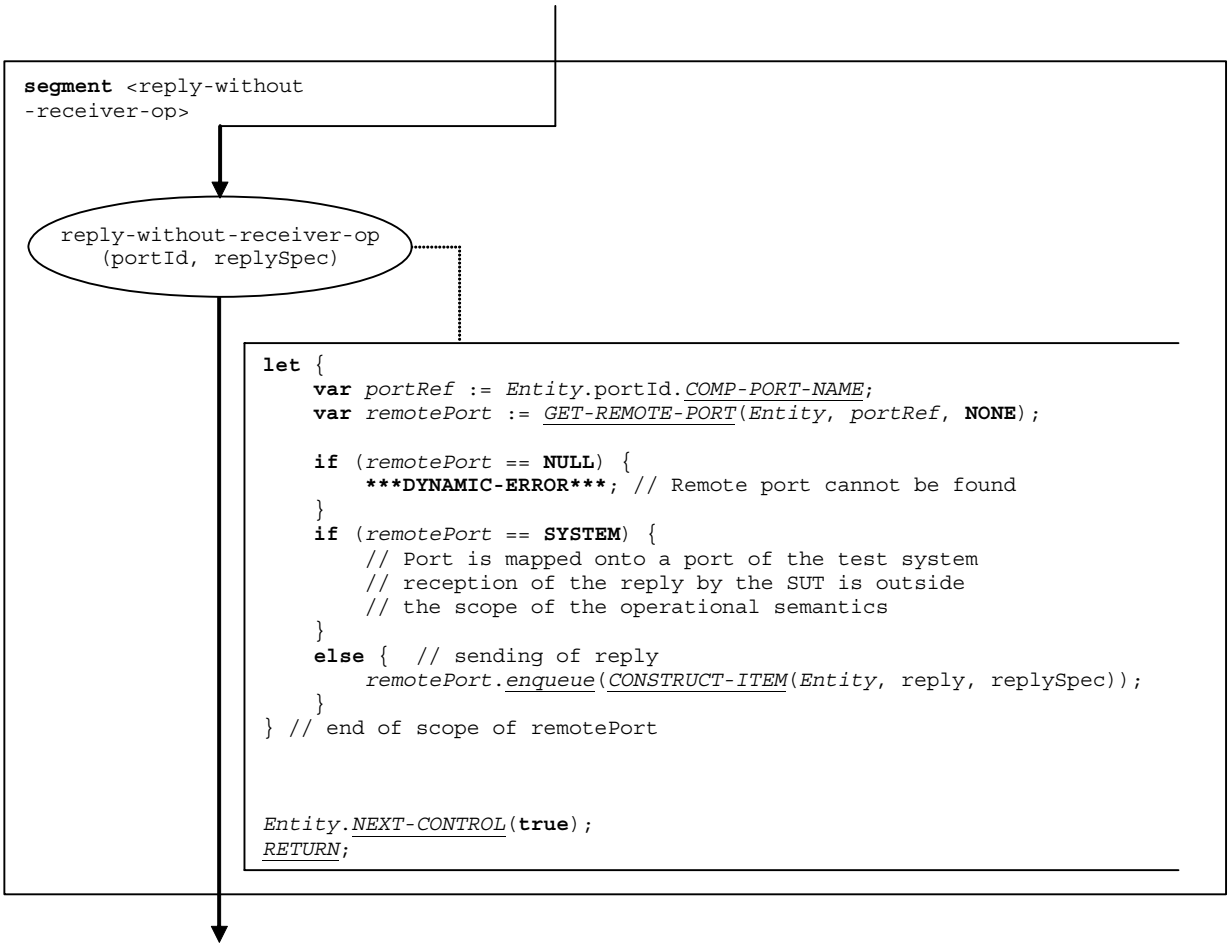


Figure 107 – Flow graph segment <reply-without-receiver-op>

9.40 Return statement

The syntactical structure of the **`return`** statement is:

```
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.

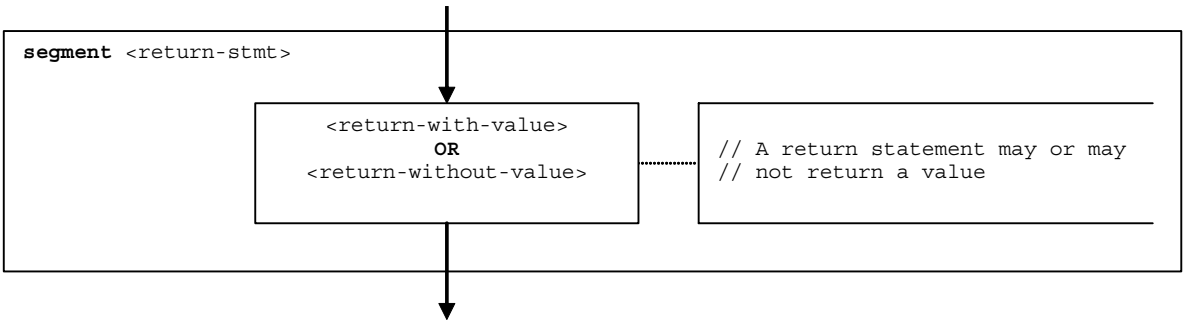


Figure 108 – Flow graph segment <return-stmt>

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`** statement that returns a value specified in form of an expression.

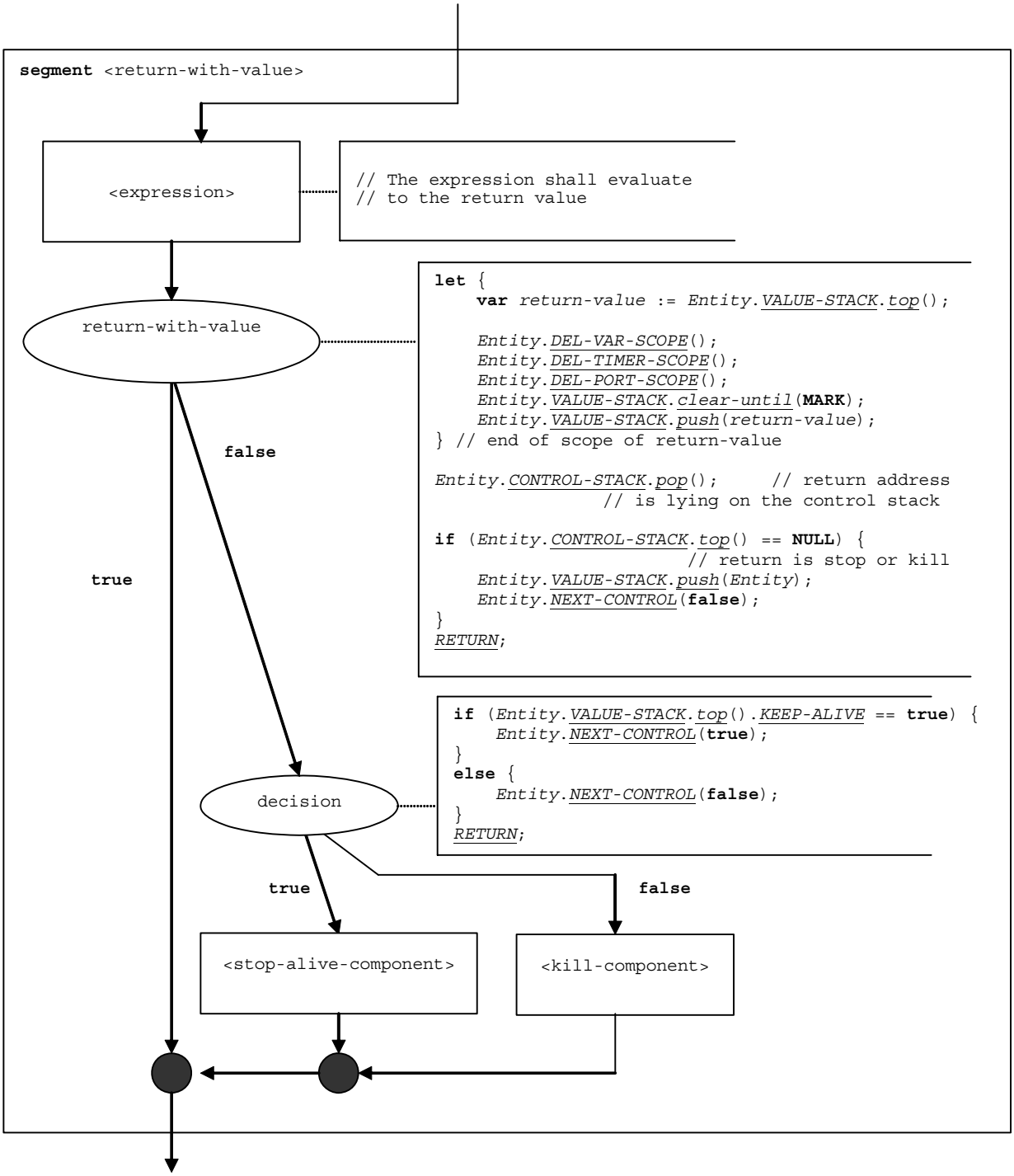


Figure 109 – Flow graph segment <return-with-value>

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.

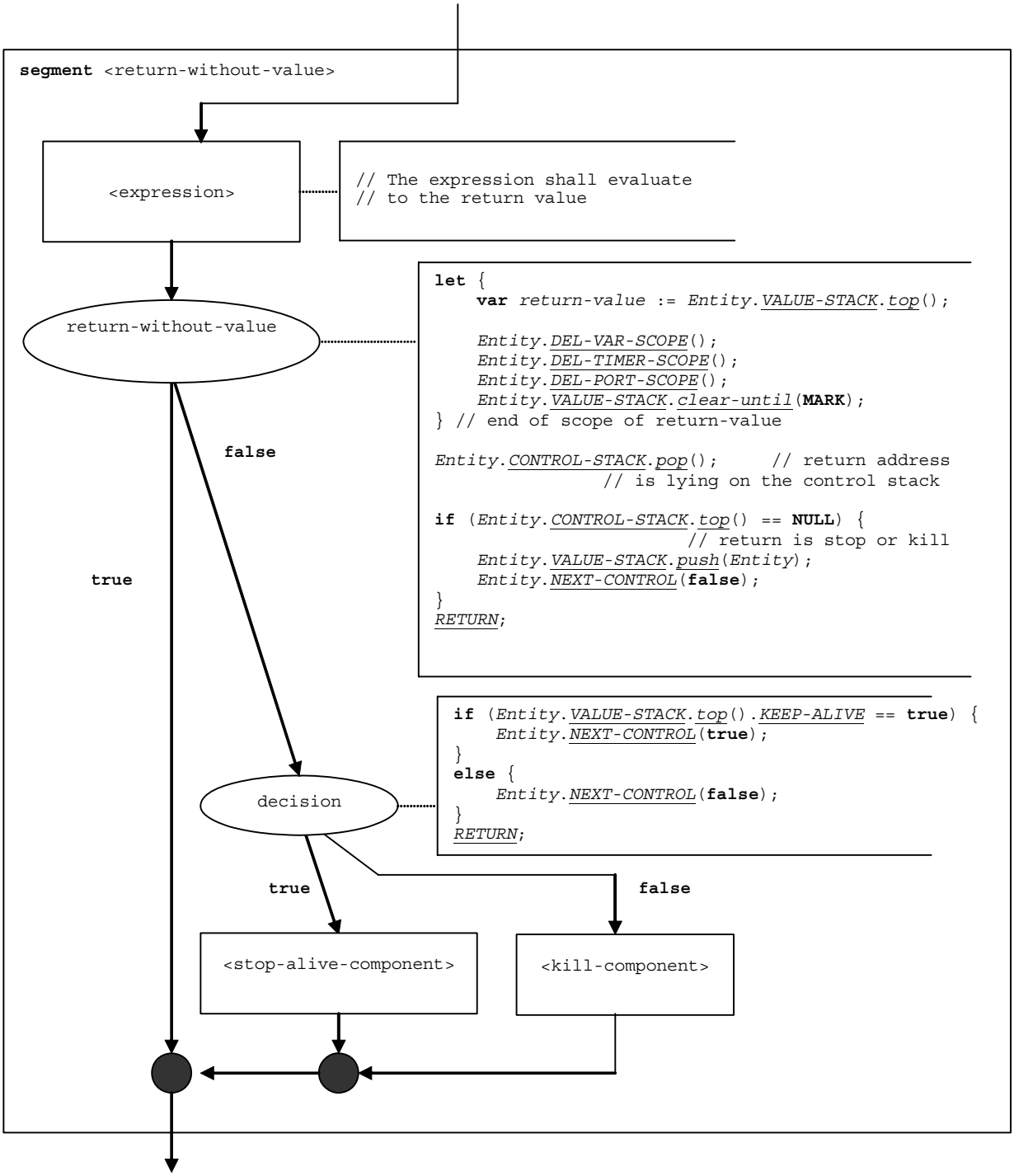


Figure 110 – Flow graph segment <return-without-value>

9.41 Running component operation

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.

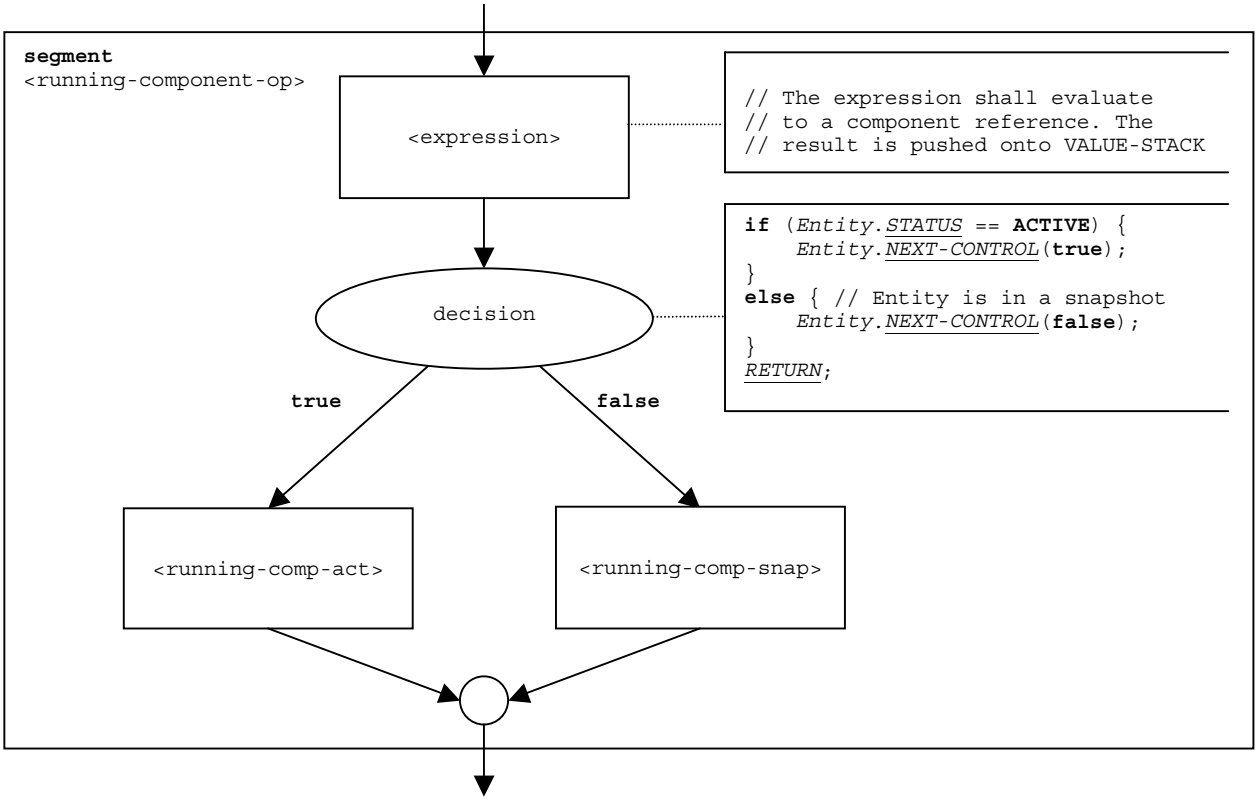


Figure 111 – Flow graph segment <running-component-op>

9.41.1 Flow graph segment <running-comp-act>

The flow graph segment `<running-comp-act>` in Figure 112 describes the execution of the **running** component operation outside a snapshot, i.e., the component is in the status **ACTIVE**.

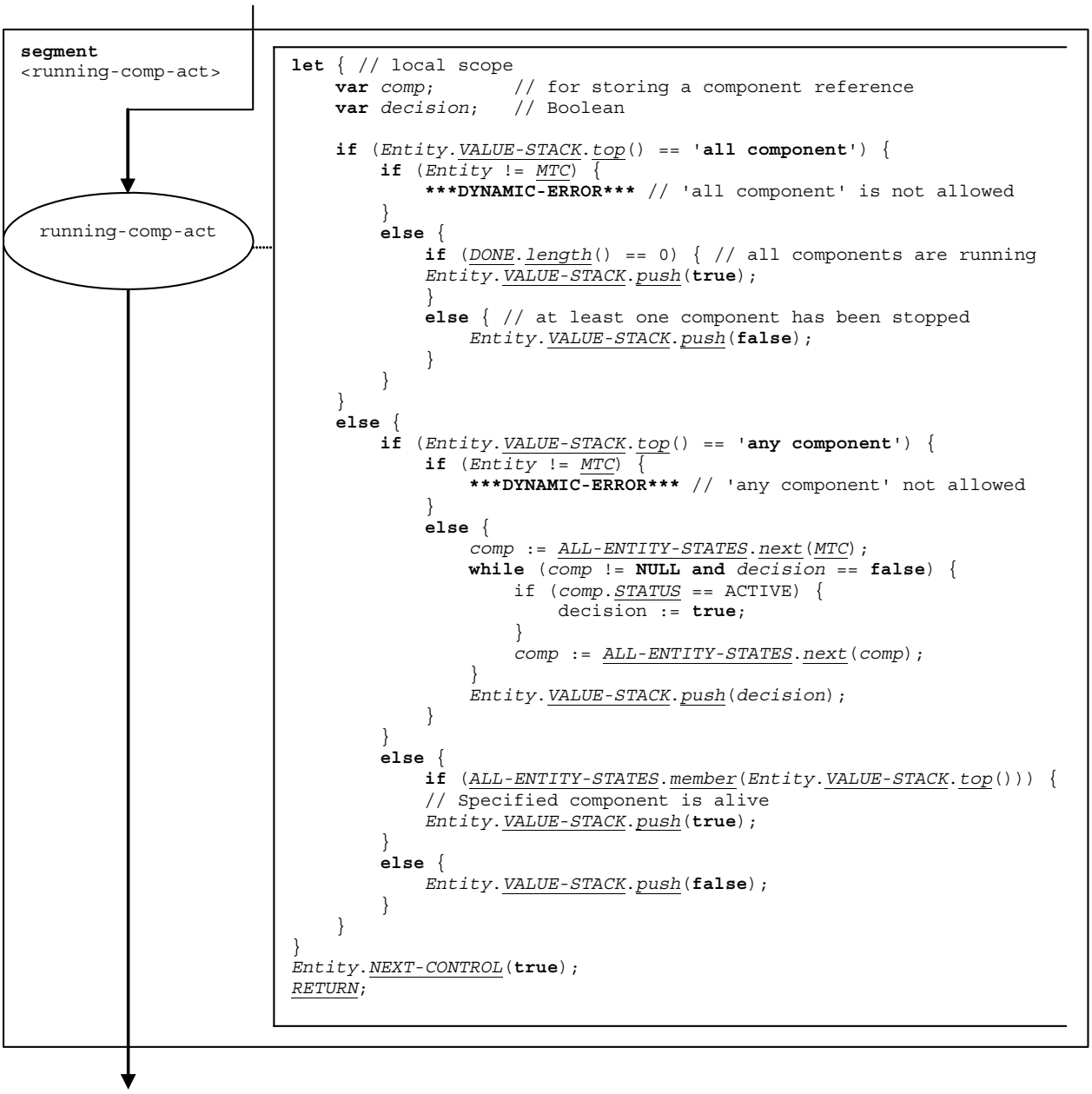


Figure 112 – Flow graph segment <running-comp-act>

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`**.

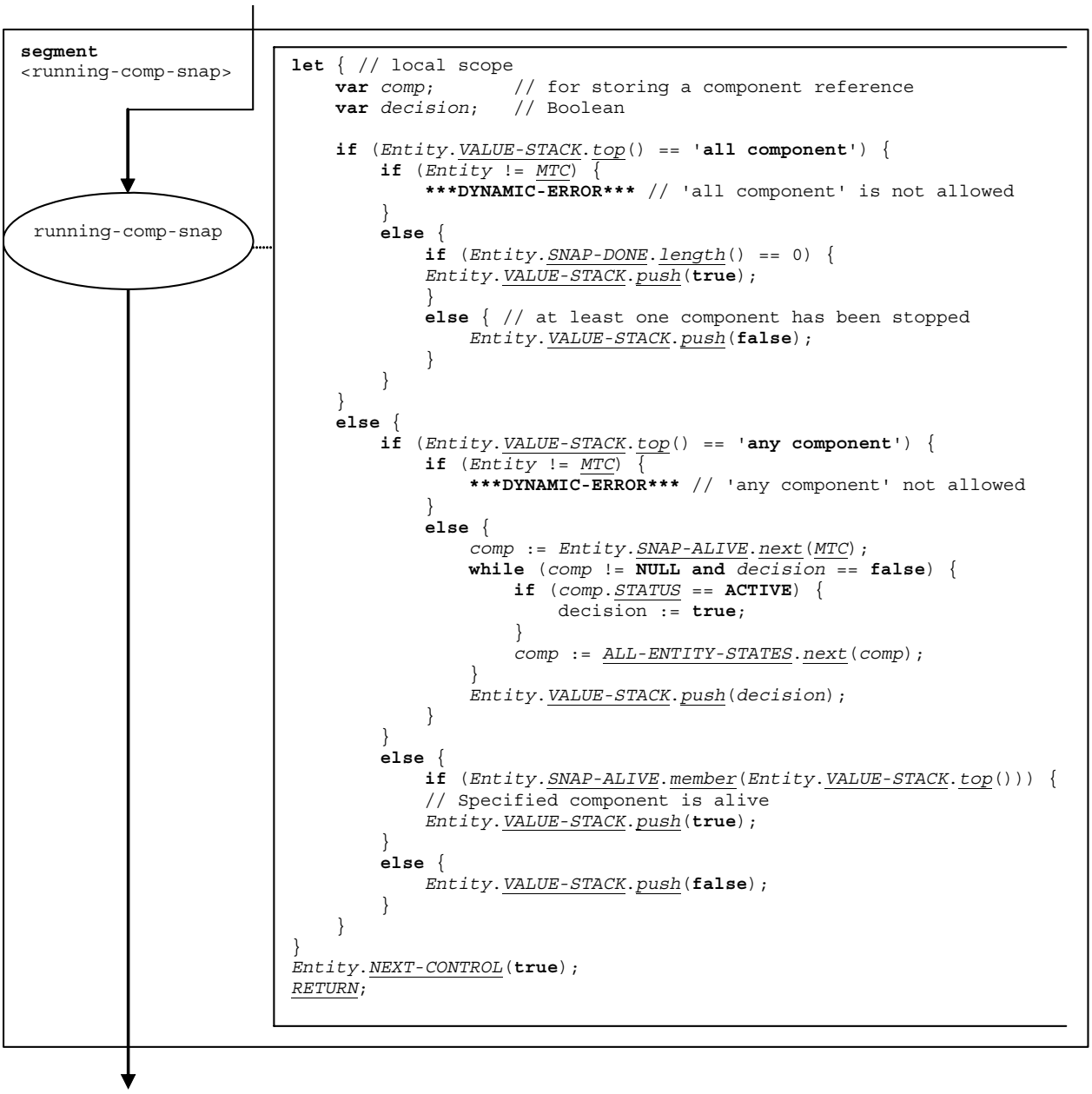


Figure 113 – Flow graph segment <running-comp-snap>

9.42 Running timer operation

The syntactical structure of the **`running`** timer operation is:

```
<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`.

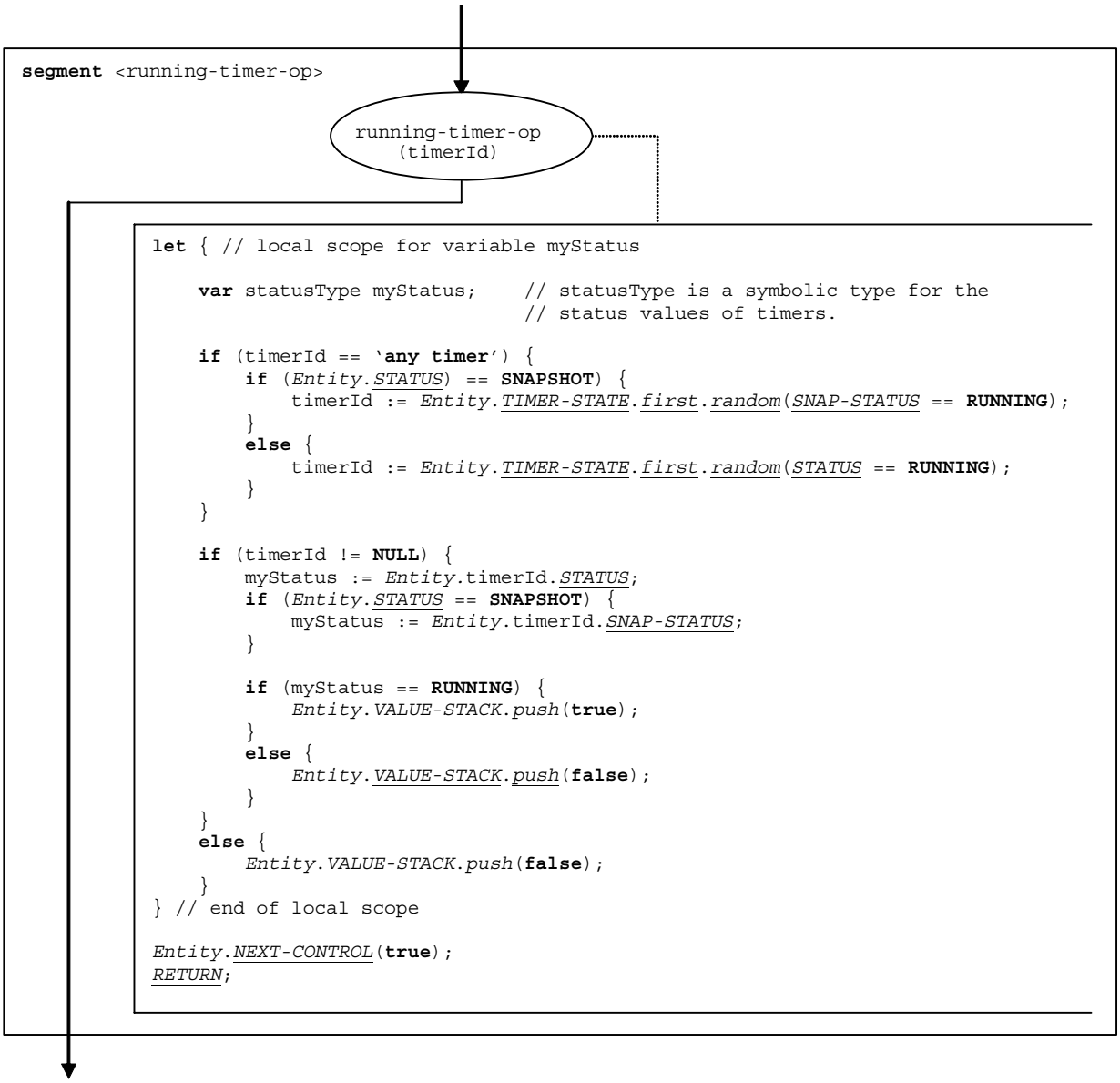


Figure 114 – Flow graph segment <running-timer-op>

9.43 Self operation

The syntactical structure of the **`self`** operation is:

```
self
```

The flow graph segment `<self-op>` in Figure 115 defines the execution of the **`self`** operation.

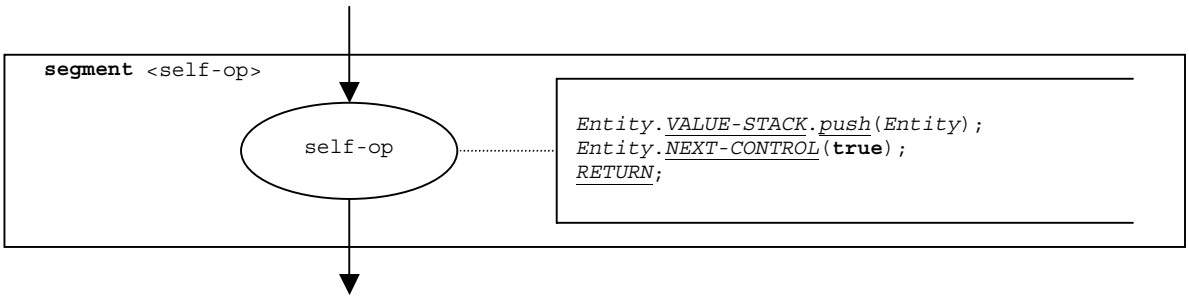


Figure 115 – Flow graph segment <self-op>

9.44 Send operation

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

```
<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 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.

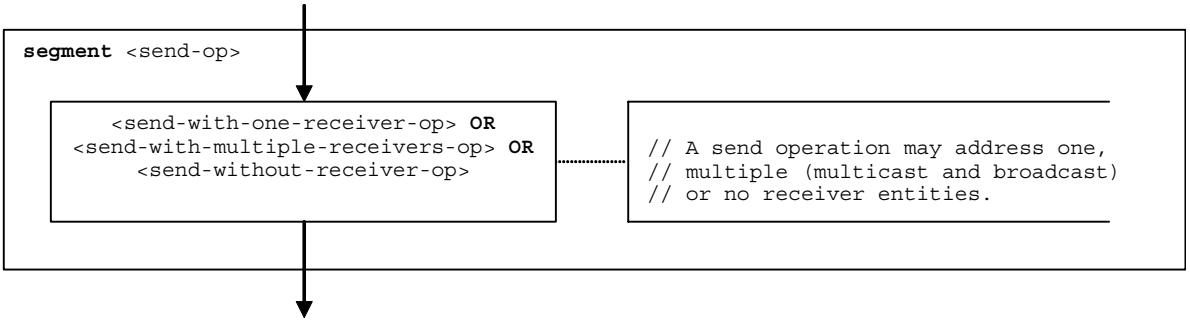


Figure 116 – Flow graph segment <send-op>

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.

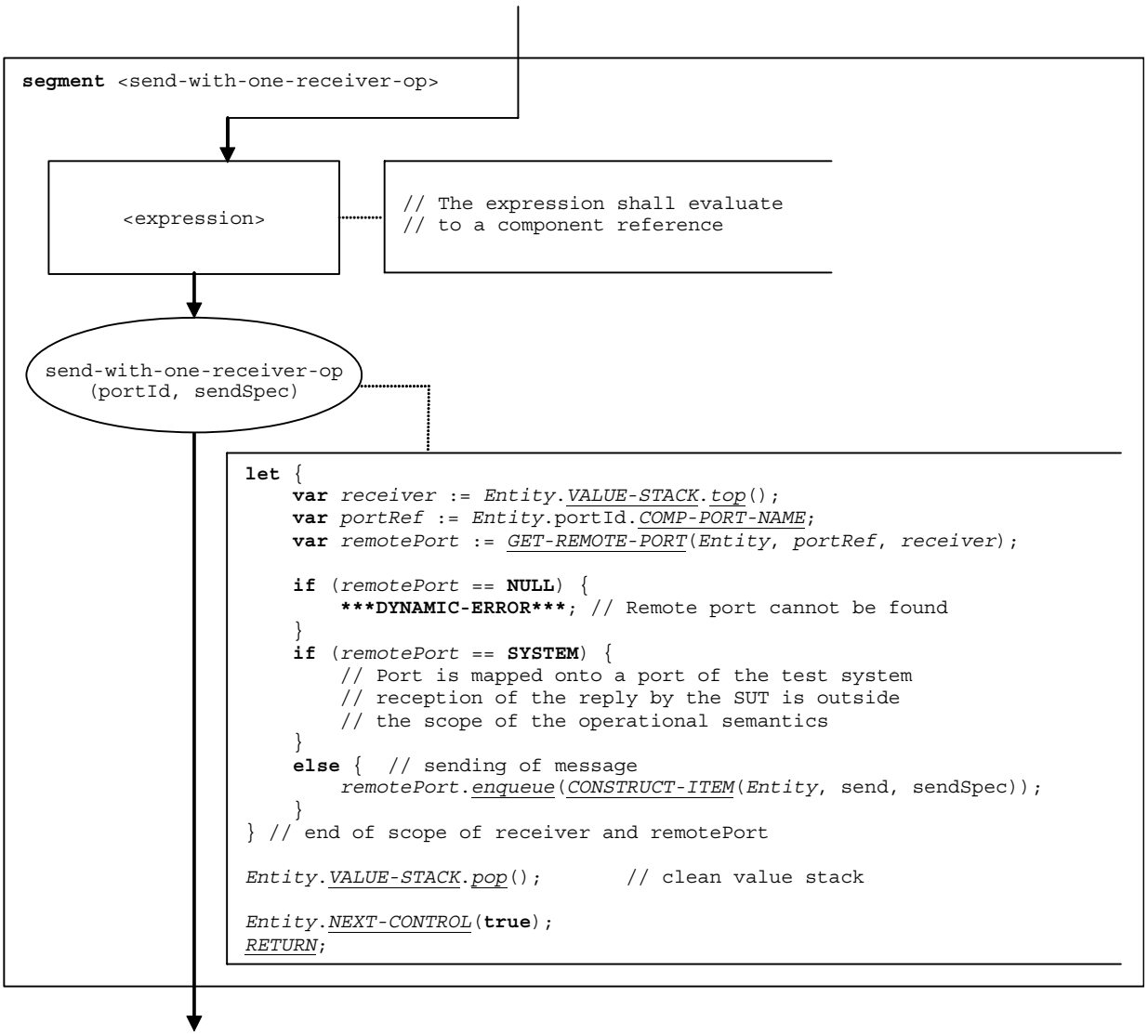


Figure 117 – Flow graph segment <send-with-one-receiver-op>

9.44.1a Flow graph segment <send-with-multiple-receivers-op>

The flow graph segment `<send-with-multiple-receivers-op>` in Figure 117-a 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.

The component references 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 117-a. The `number` parameter is 1 in case of broadcast communication, i.e., the keyword **`all component`** is top element in the value stack.

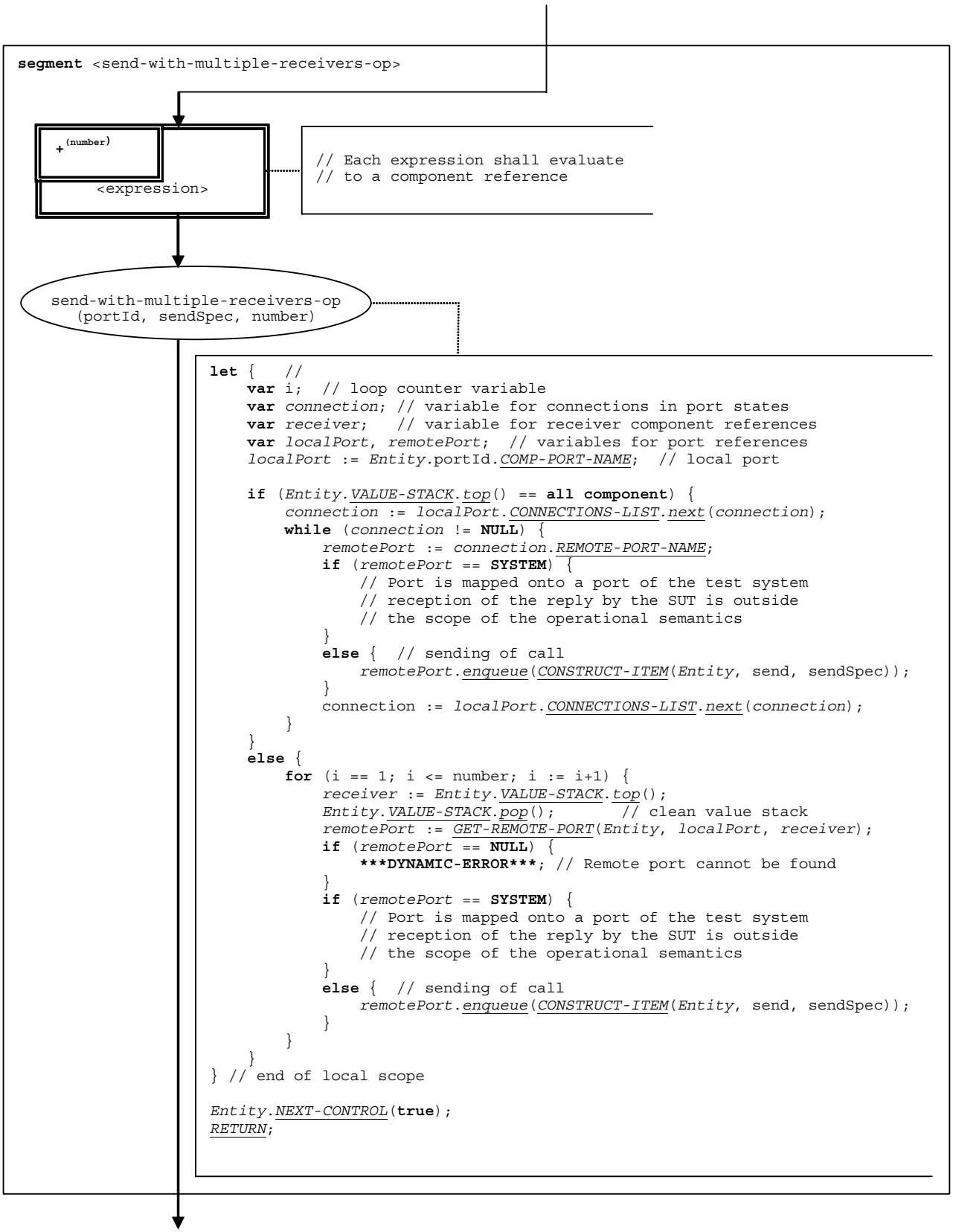


Figure 117-a – Flow graph segment <send-with-multiple-receivers-op>

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.

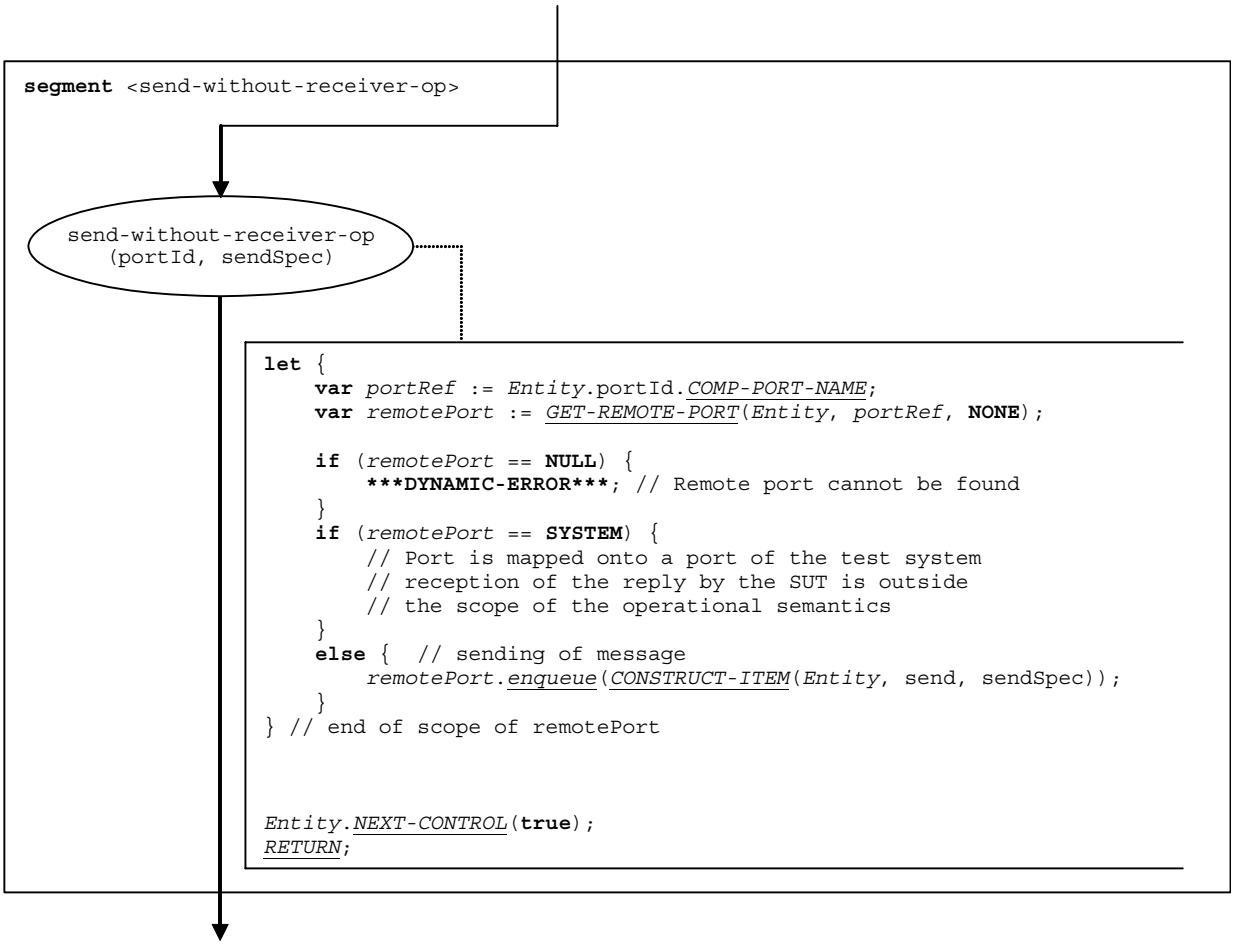


Figure 118 – Flow graph segment <send-without-receiver-op>

9.45 Setverdict operation

The syntactical structure of the **`setverdict`** operation is:

```
setverdict(<verdicttype-expression>)
```

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 flow graph segment `<setverdict-op>` in Figure 119 defines the execution of the **`setverdict`** operation.

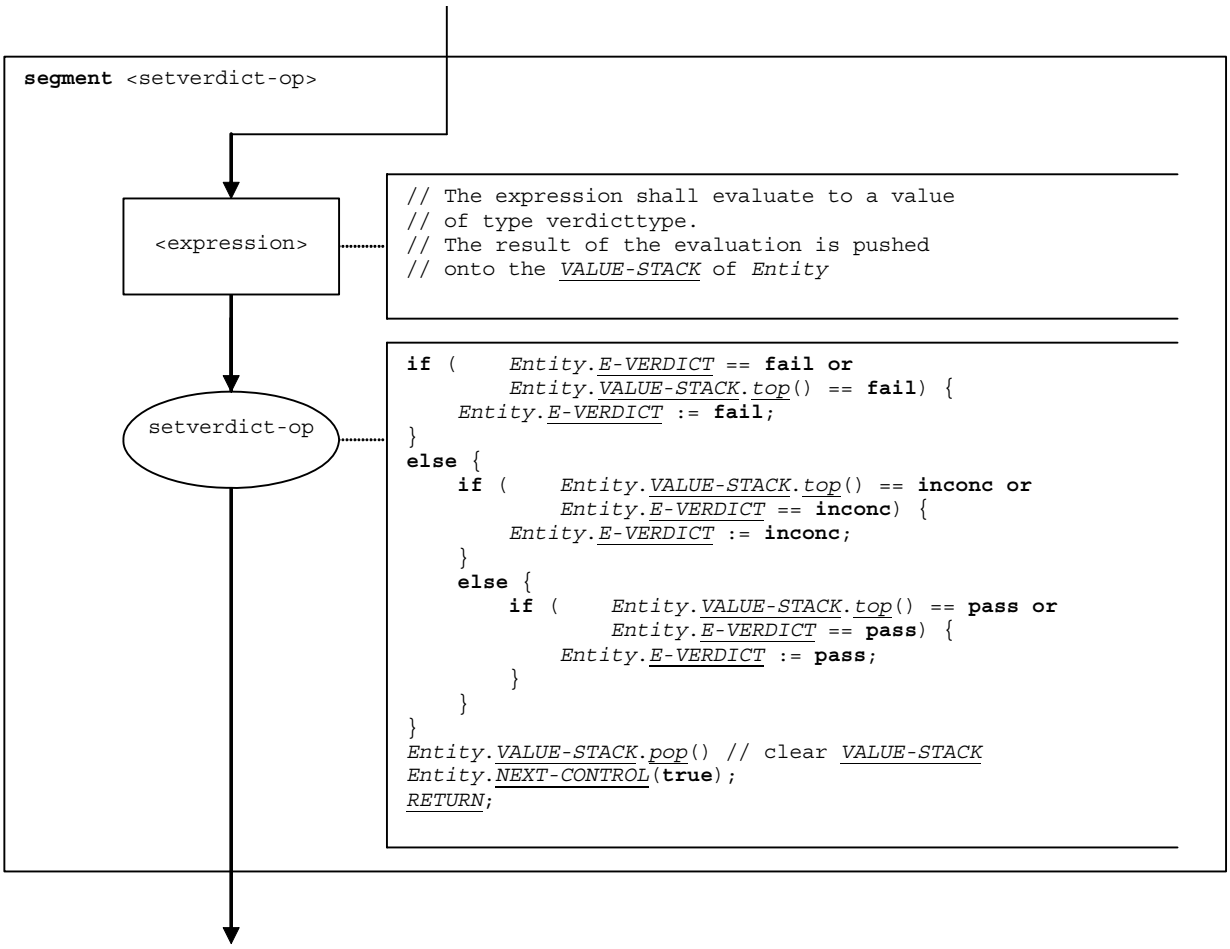


Figure 119 – Flow graph segment <setverdict-op>

9.46 Start component operation

The syntactical structure of the **start** component operation is:

```
<component-expression>.start(<function-name>(<act-par-desc1>,…, <act-par-descn>))
```

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 the name of the function that defines the behaviour of the new component and `<act-par-desc`$_1$`>`, …, `<act-par-desc`$_n$`>` provide the description of the actual parameter values of `<function-name>`. In functions referenced in **start** component operations only value parameters are allowed. 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.

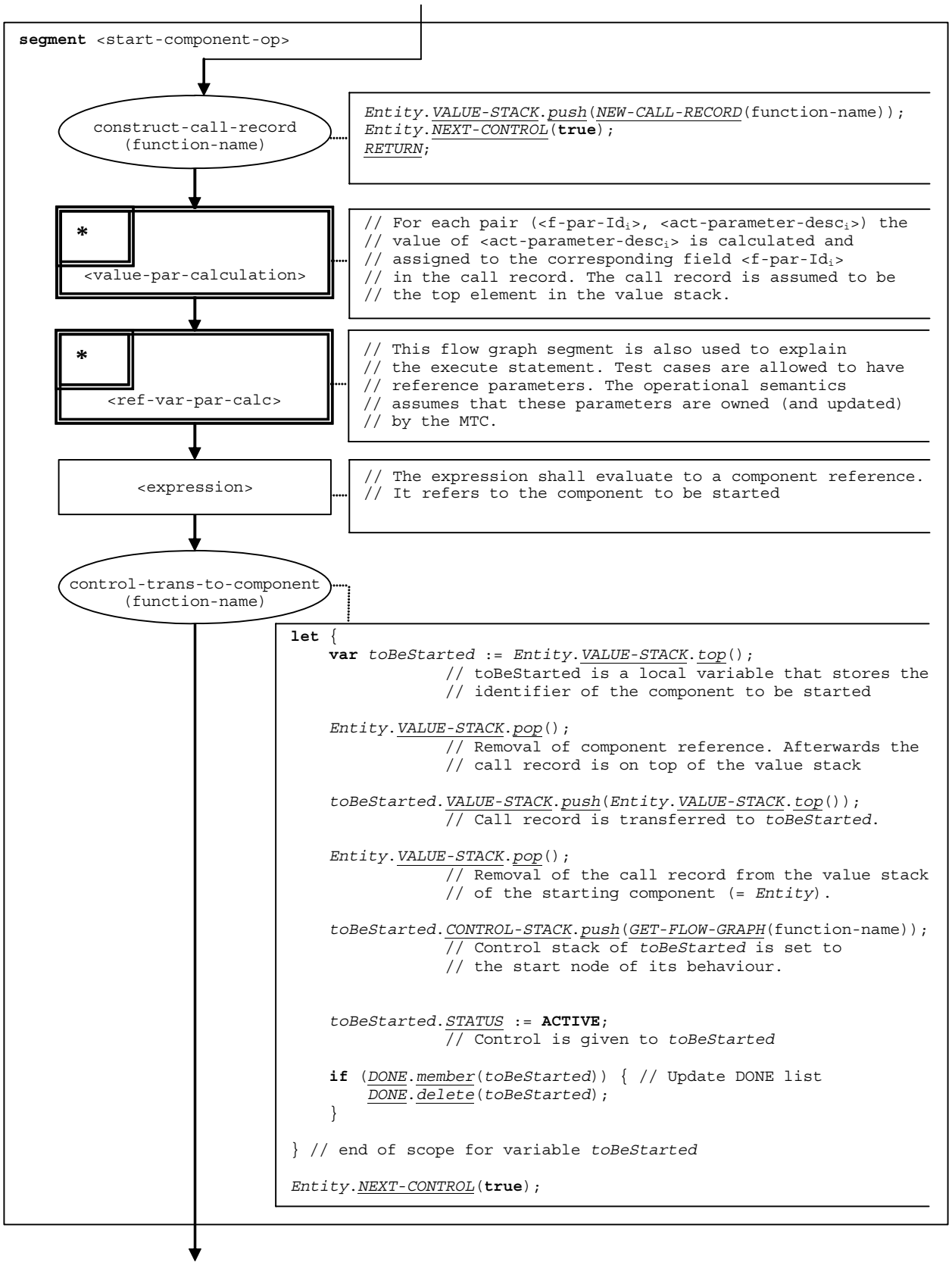


Figure 120 – Flow graph segment <start-component-op>

9.47 Start port operation

The syntactical structure of the **start** port operation is:

```
<portId>.start
```

The flow graph segment <start-port-op> in Figure 121 defines the execution of the **start** port operation.

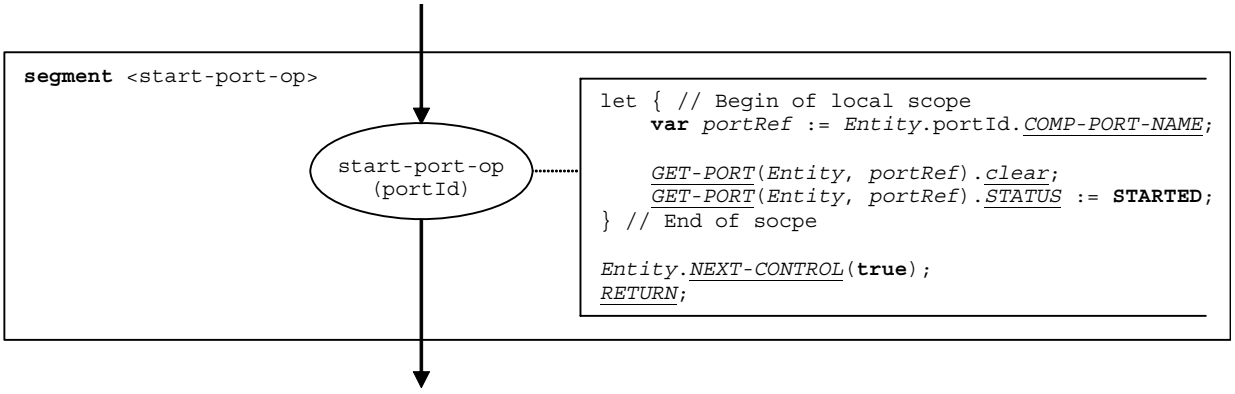


Figure 121 – Flow graph segment <start-port-op>

9.48 Start timer operation

The syntactical structure of the **start** timer operation is:

```
<timerId>.start [(<float-expression>)]
```

The optional <float-expression> parameter of the **start** timer operation denotes the actual duration of the timer. If it is not provided, the default duration will be used by the **start** operation. The expression shall evaluate to a value of type **float**. If provided, the expression shall be evaluated before the **start** operation is applied. The result of the evaluation is pushed onto the *VALUE-STACK* of *Entity*.

The flow graph segment <start-timer-op> in Figure 122 defines the execution of the **start** timer operation.

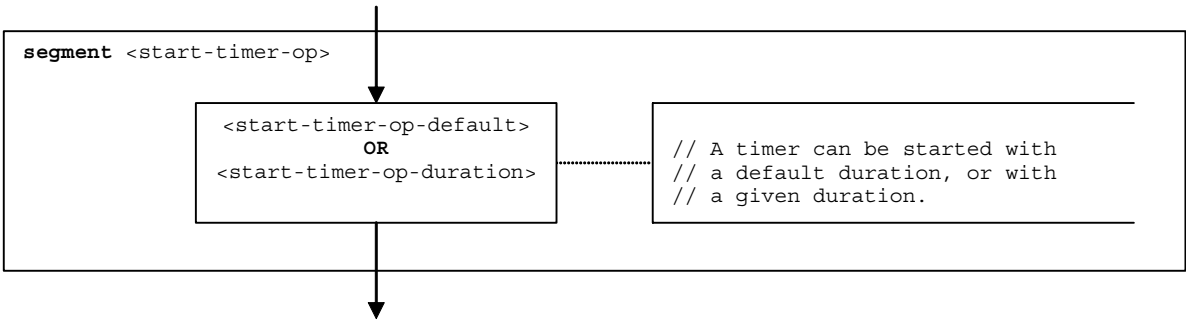


Figure 122 – Flow graph segment <start-timer-op>

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.

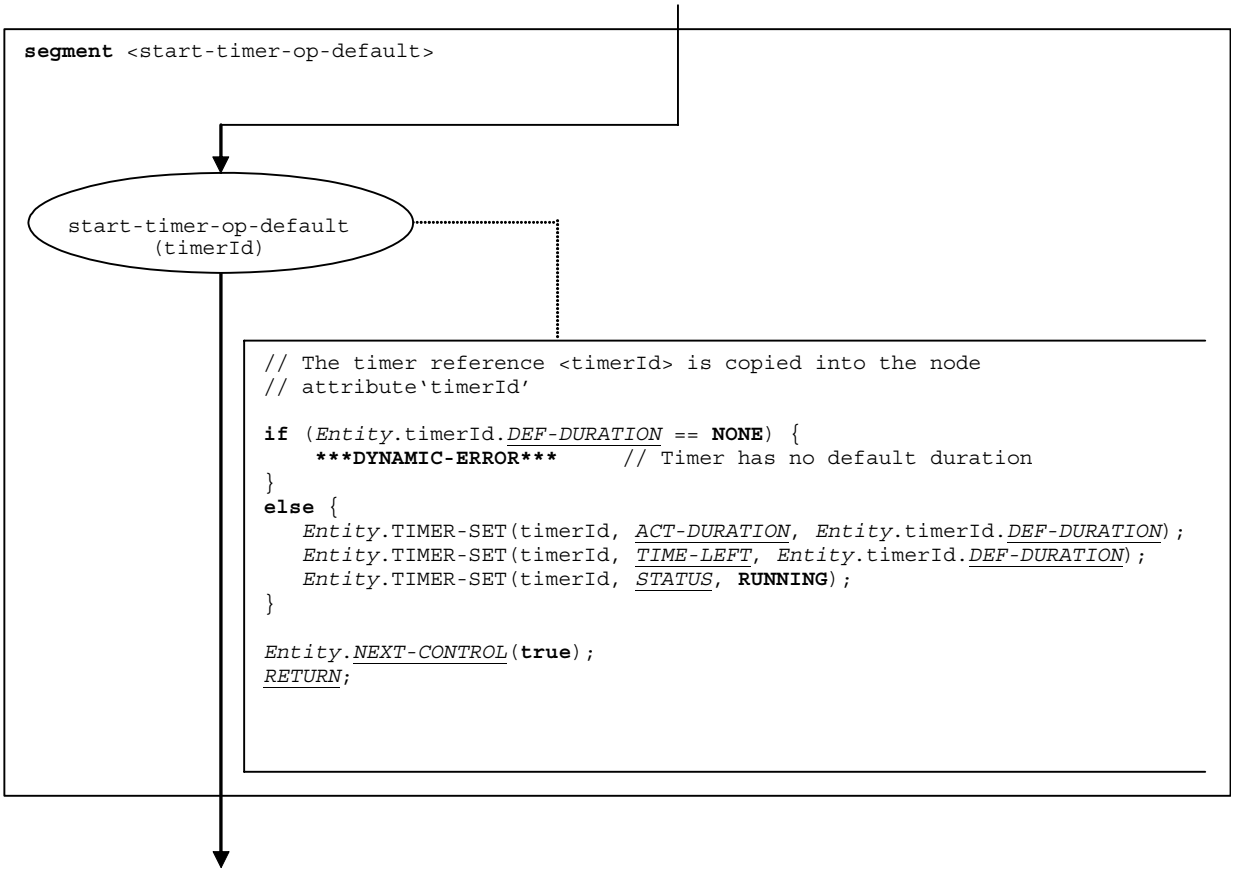


Figure 123 – Flow graph segment <start-timer-op-default>

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.

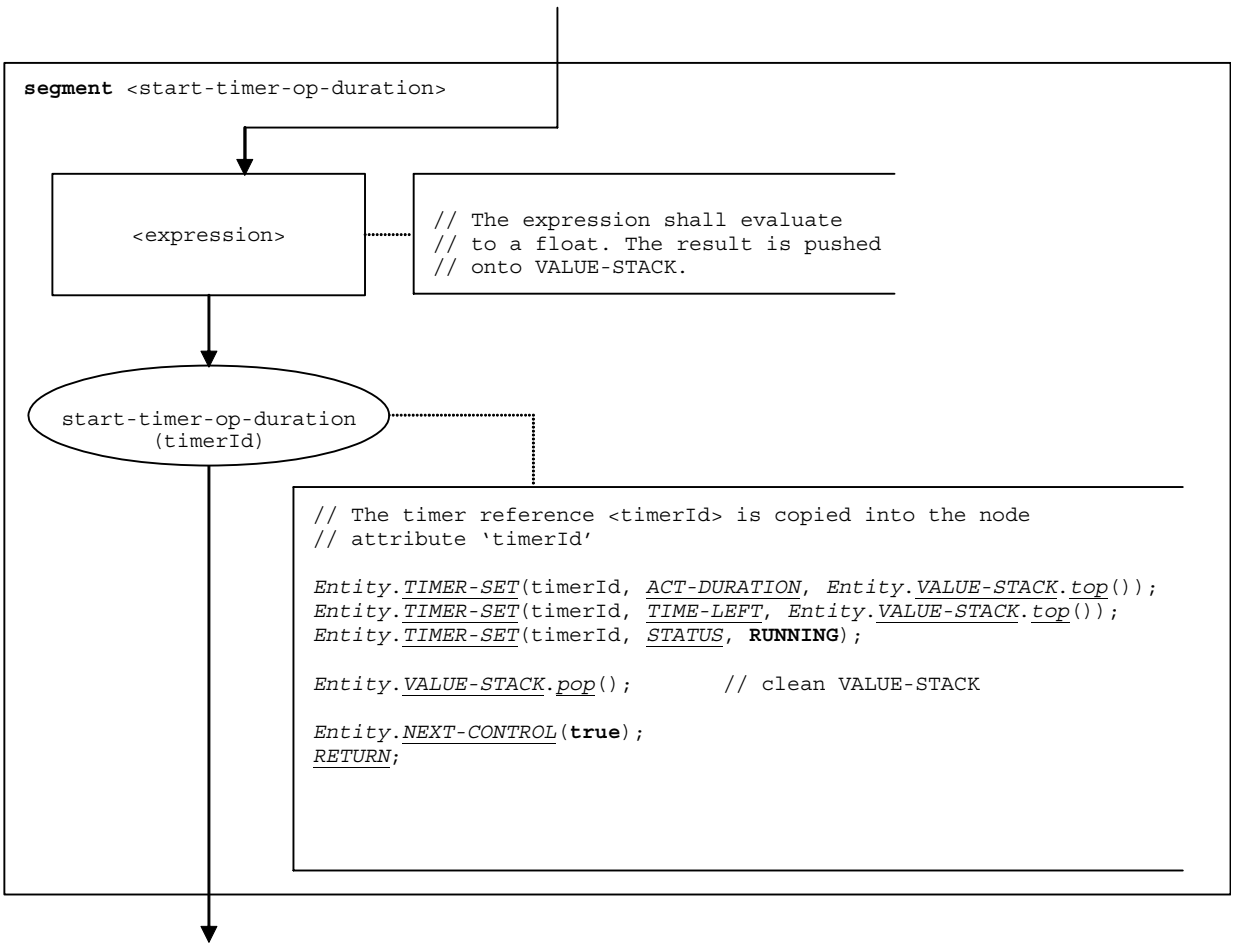


Figure 124 – Flow graph segment <start-timer-op-duration>

9.49 Stop component operation

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.

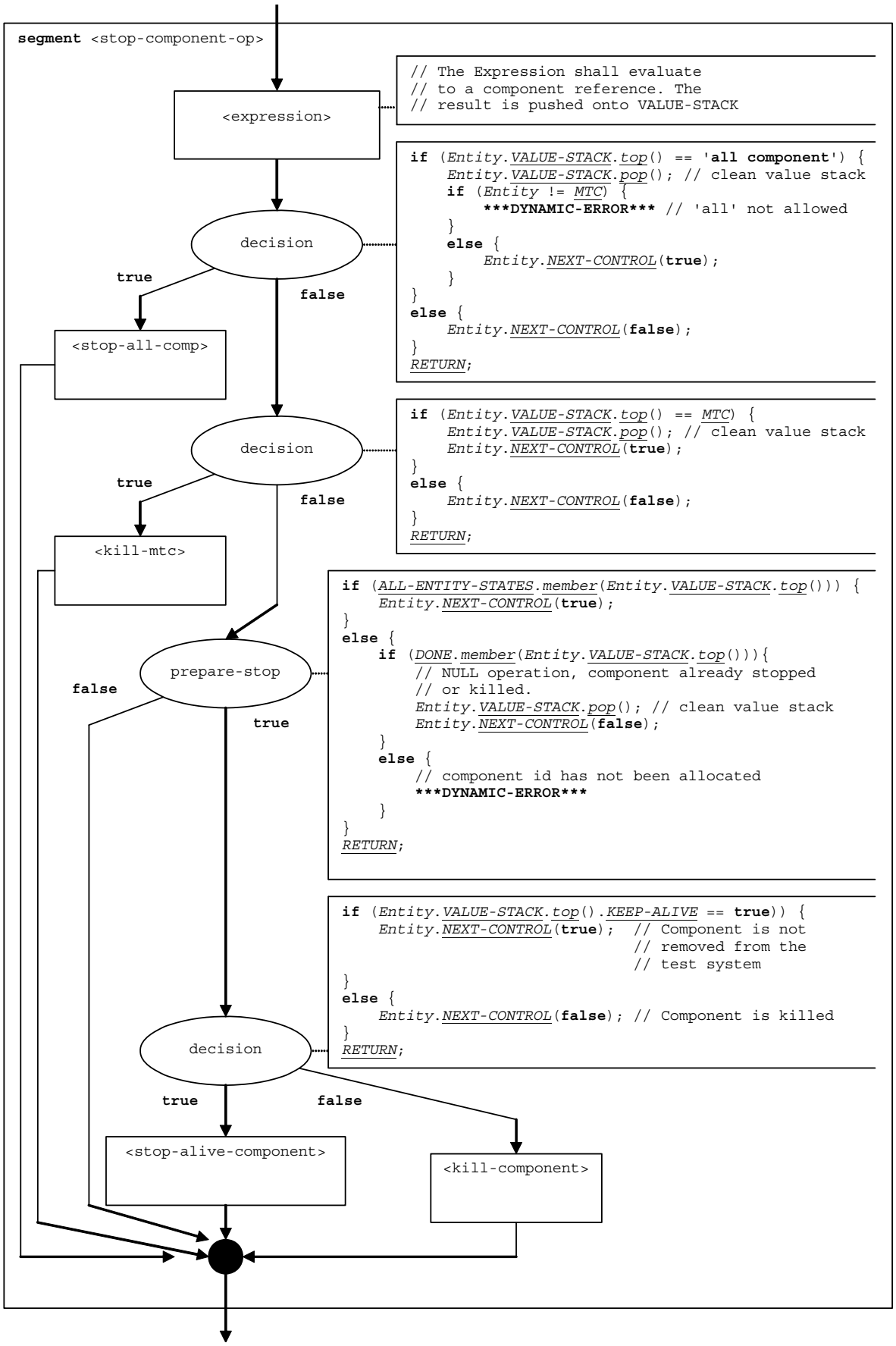


Figure 125 – Flow graph segment <stop-component-op>

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.

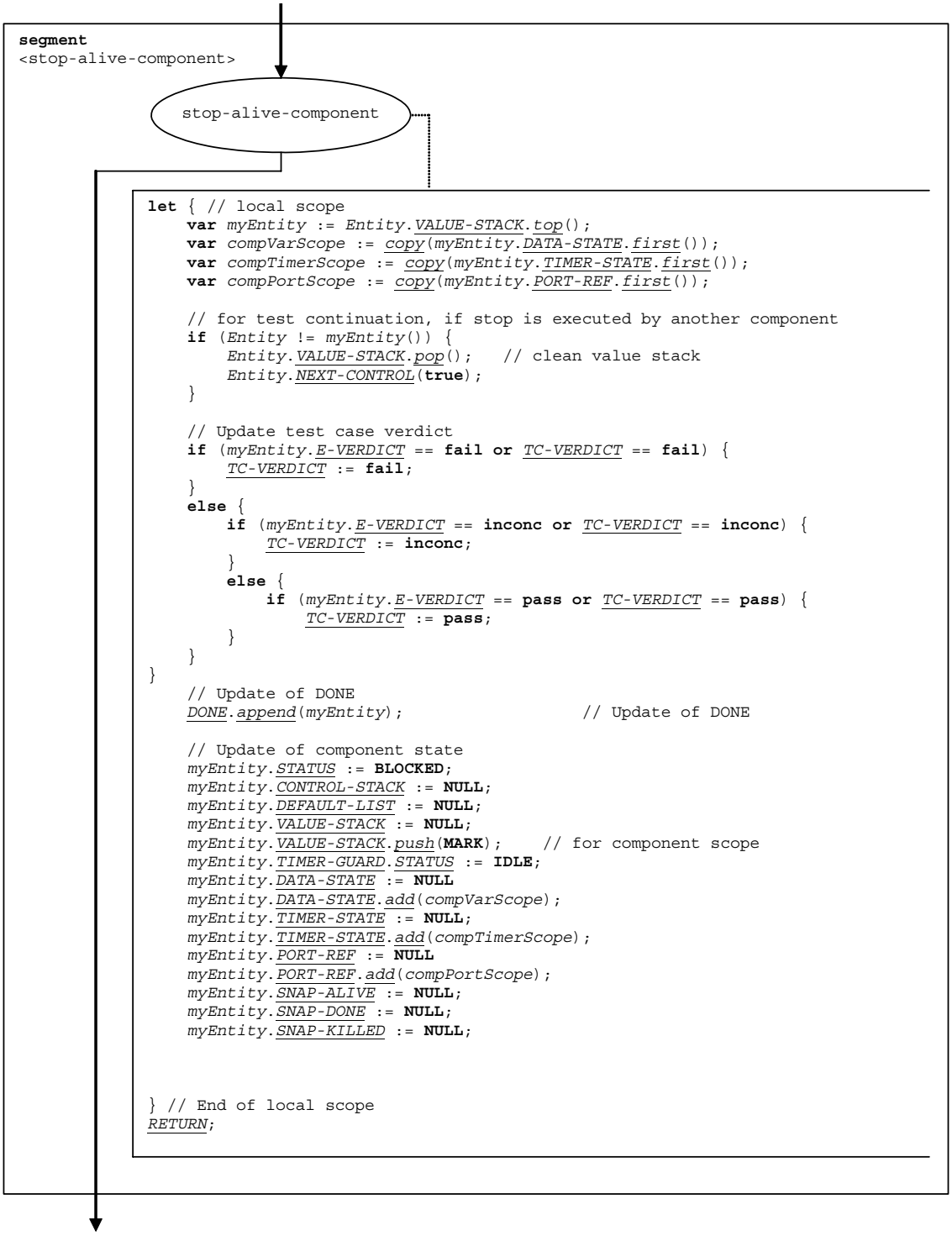


Figure 126 – Flow graph segment <stop-alive-component>

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.

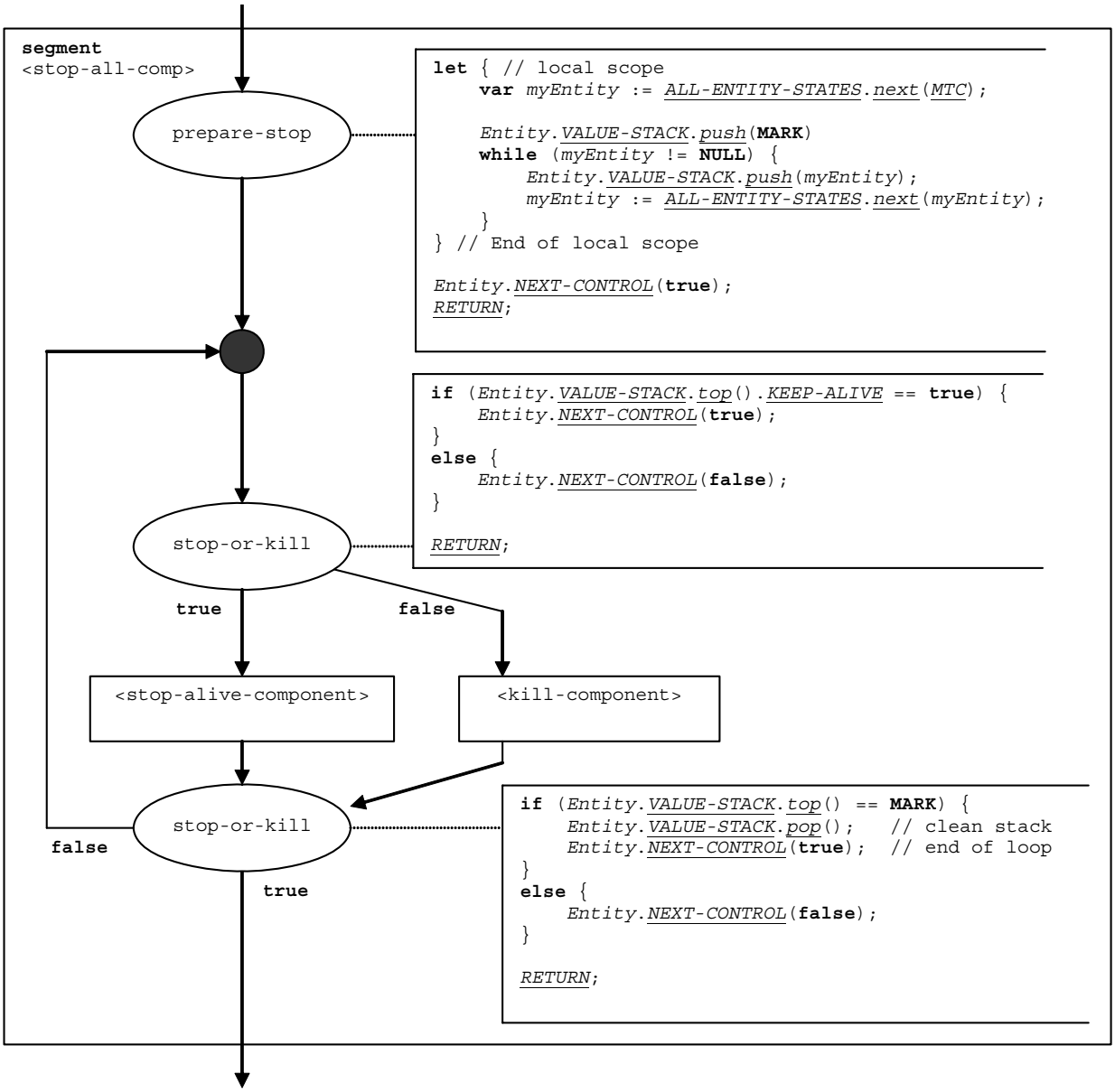


Figure 127 – Flow graph segment <stop-all-comp>

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.

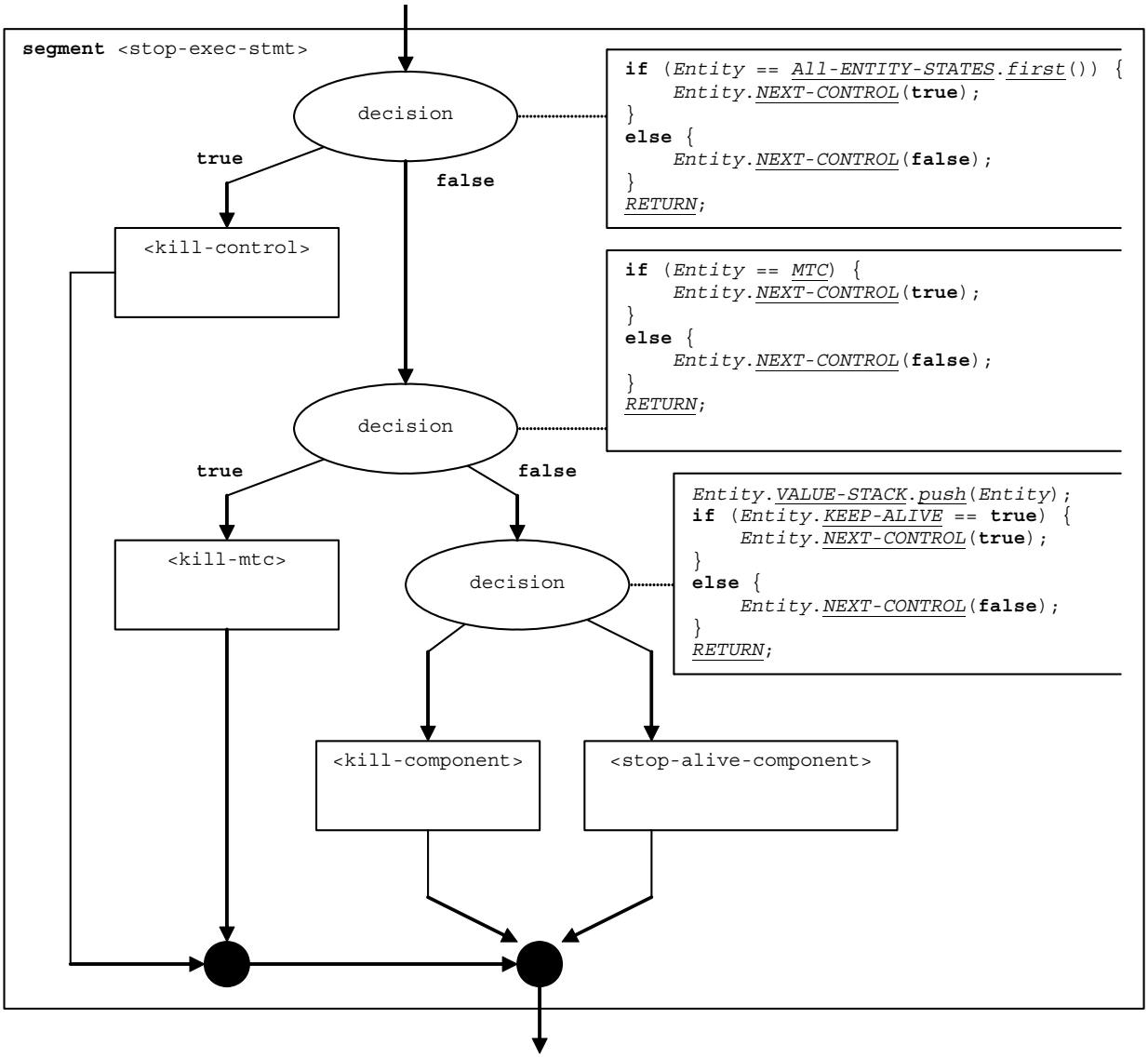


Figure 128 – Flow graph segment <stop-exec-stmt>

9.50.1 Void

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.

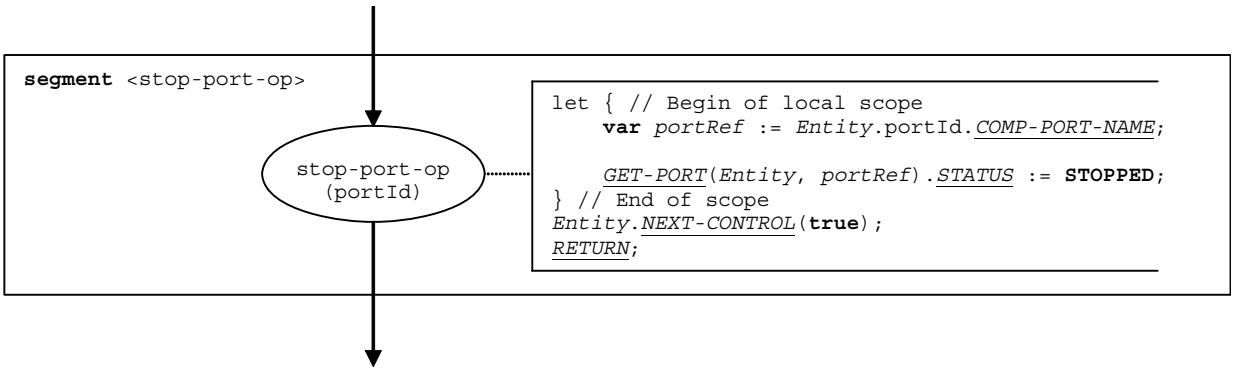


Figure 129 – Flow graph segment <stop-port-op>

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`.

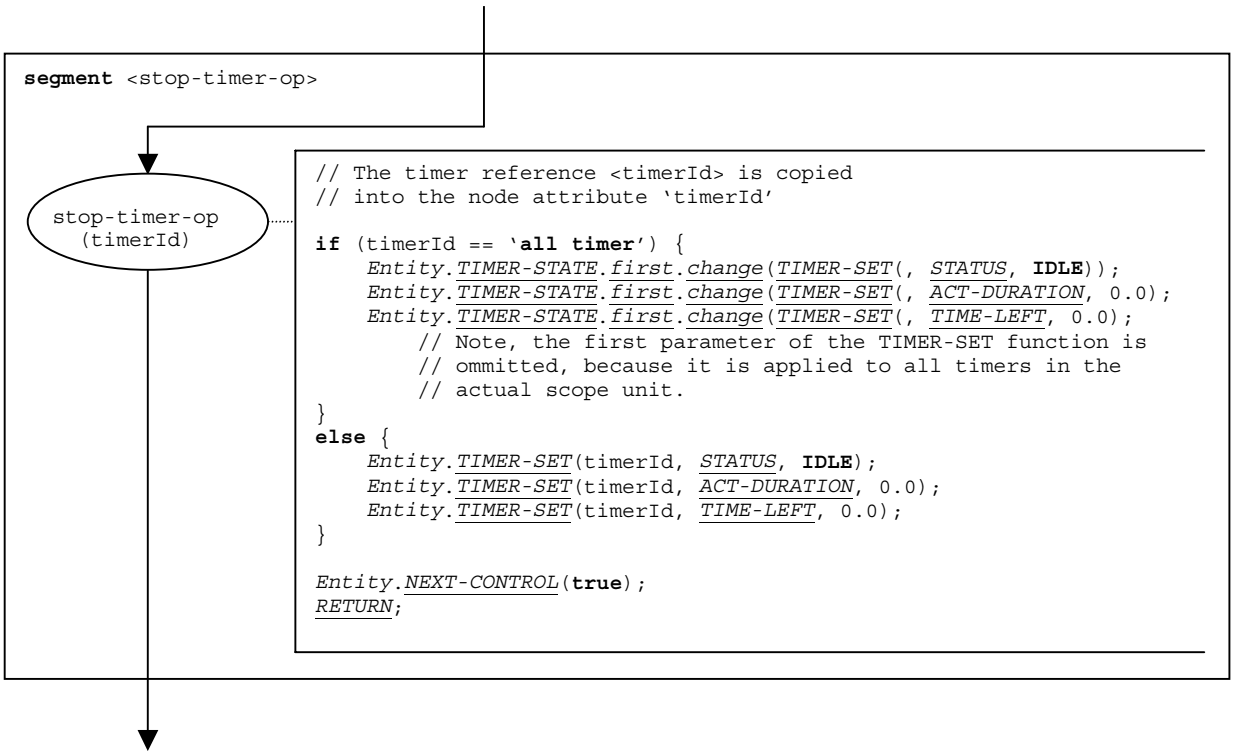


Figure 130 – Flow graph segment <stop-timer-op>

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.

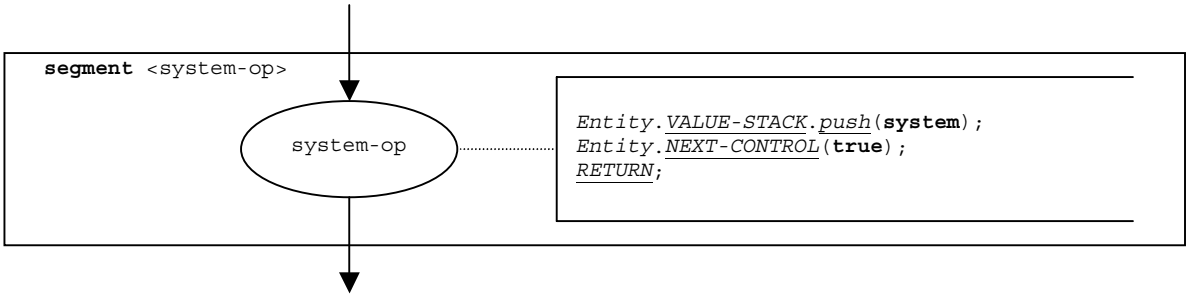


Figure 131 – Flow graph segment <system-op>

9.54 Timer declaration

The syntactical structure of a **timer** declaration is:

```
timer <timerId> [:= <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 <timer-declaration> in Figure 132 defines the execution of a **timer** declaration.

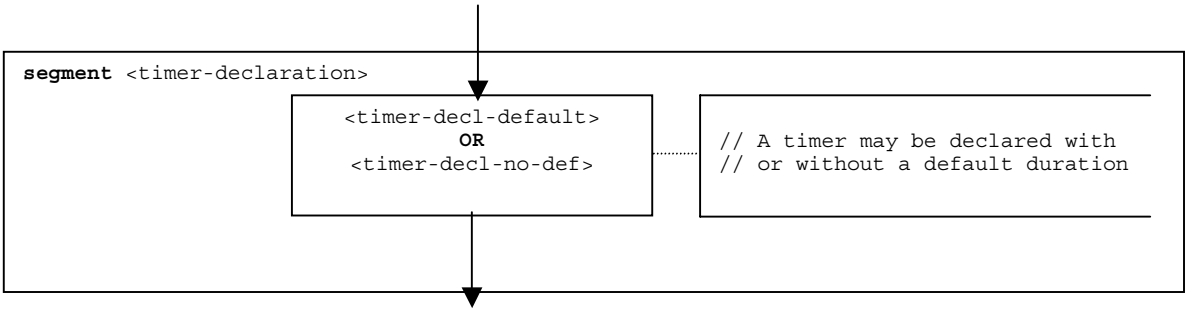


Figure 132 – Flow graph segment <timer-declaration>

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.

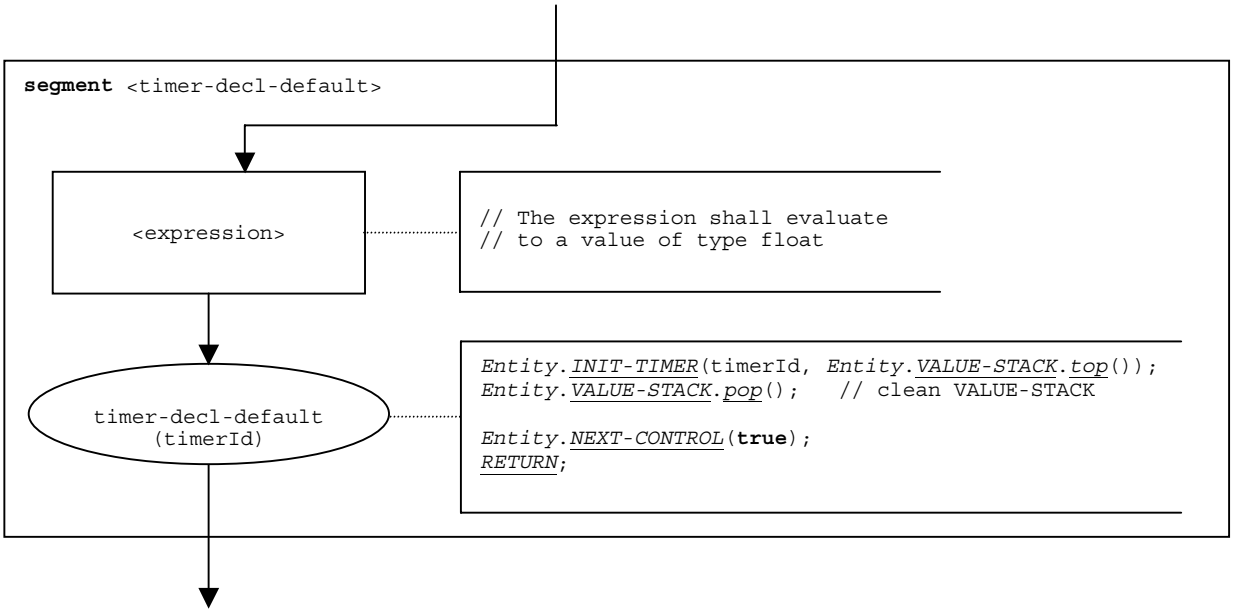


Figure 133 – Flow graph segment <timer-decl-default>

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.

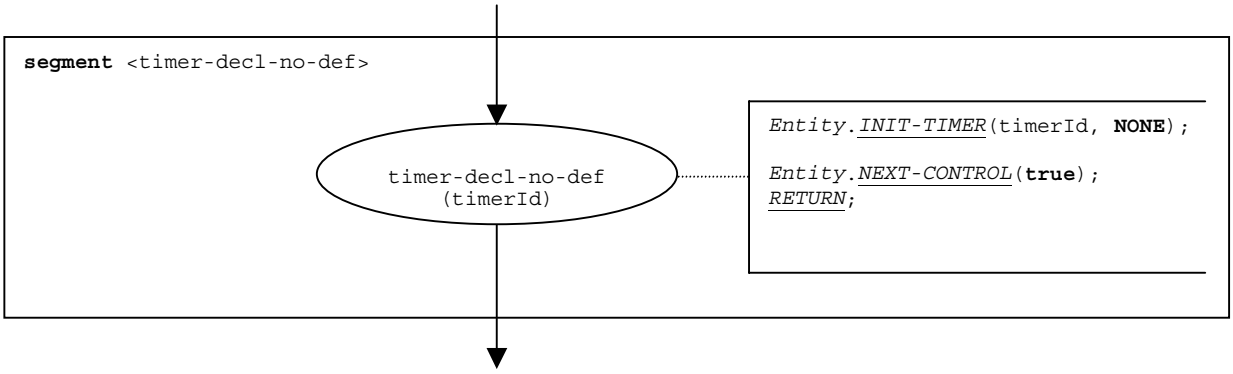


Figure 134 – Flow graph segment <timer-decl-no-def>

9.55 Timeout timer operation

The syntactical structure of the **`timeout`** timer operation is:

```
<timerId>.timeout
```

The flow graph segment `<timeout-timer-op>` in Figure 135 defines the execution of the **`timeout`** timer operation.

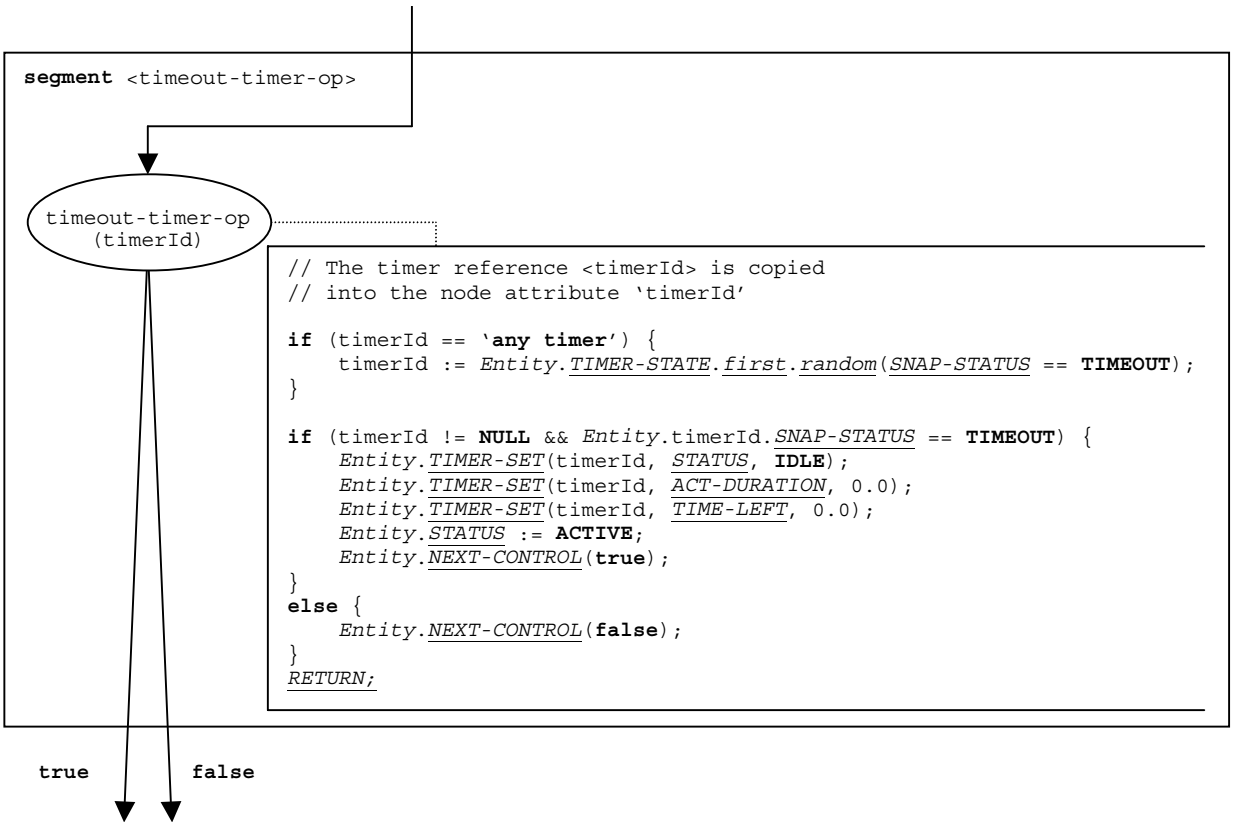


Figure 135 – Flow graph segment <timeout-timer-op>

9.56 Unmap operation

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

```
unmap(<component_expression>:<portId1>,system:<portId2>)
```

The identifiers `<portId1>` and `<portId2>` are considered to be port identifiers of the corresponding test component and test system interface. The components 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 **unmap** 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 **unmap** operation is defined by the flow graph segment `<unmap-op>` shown in Figure 136.

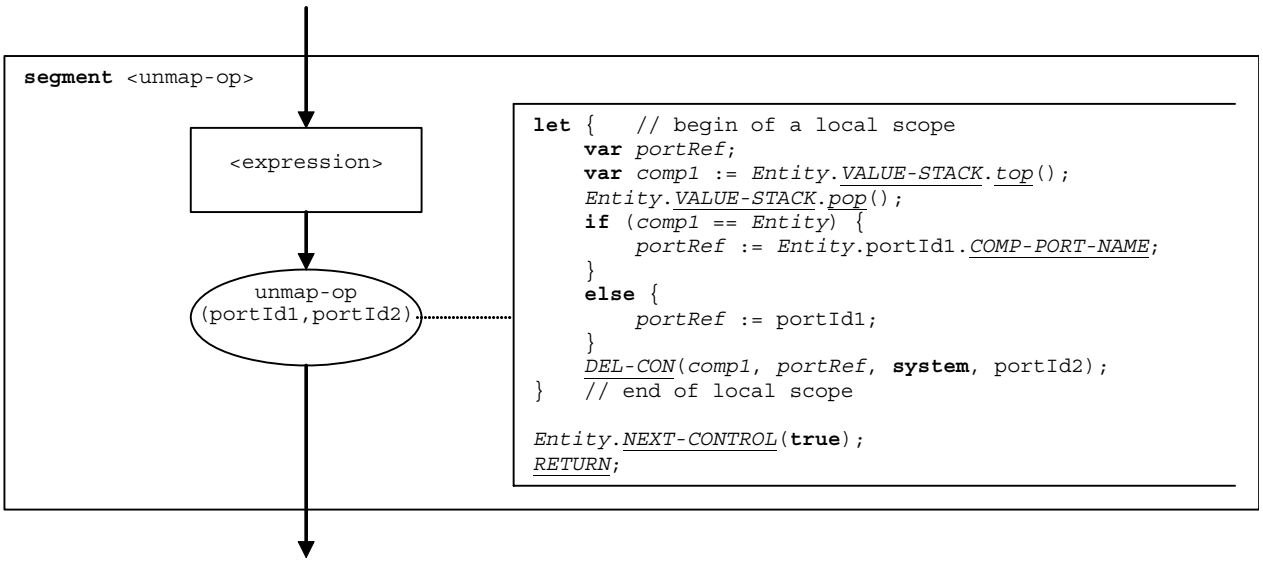


Figure 136 – Flow graph segment <unmap-op>

9.57 Variable declaration

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.

The flow graph segment `<variable-declaration>` in Figure 137 defines the execution of the declaration of a variable.

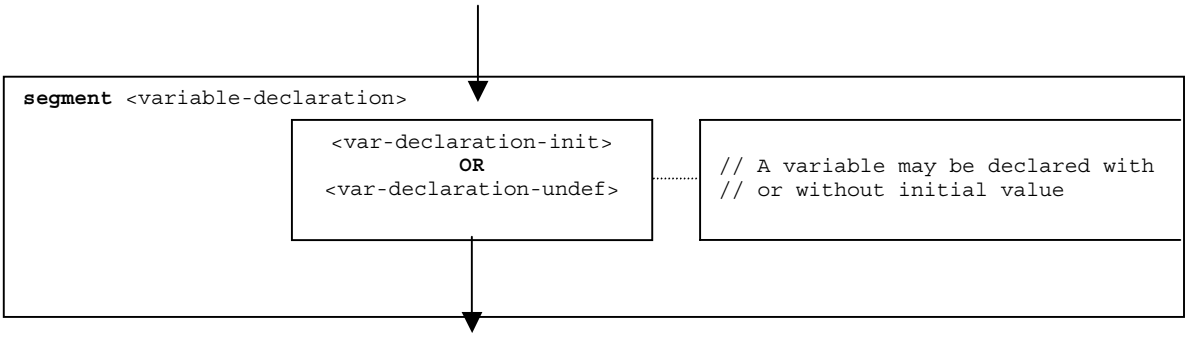


Figure 137 – Flow graph segment <variable-declaration>

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.

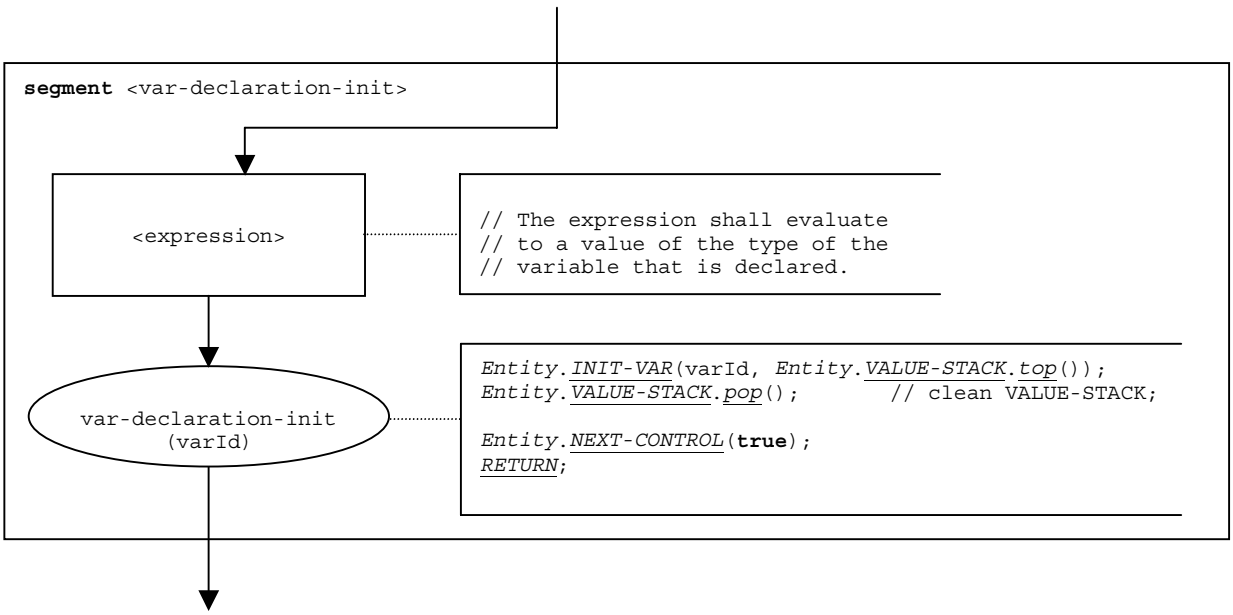


Figure 138 – Flow graph segment <var-declaration-init>

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.

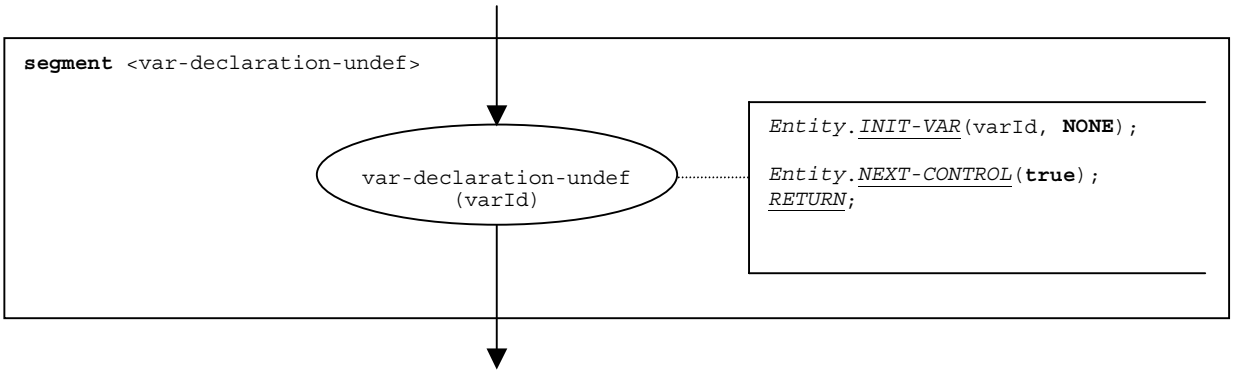


Figure 139 – Flow graph segment <var-declaration-undef>

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.

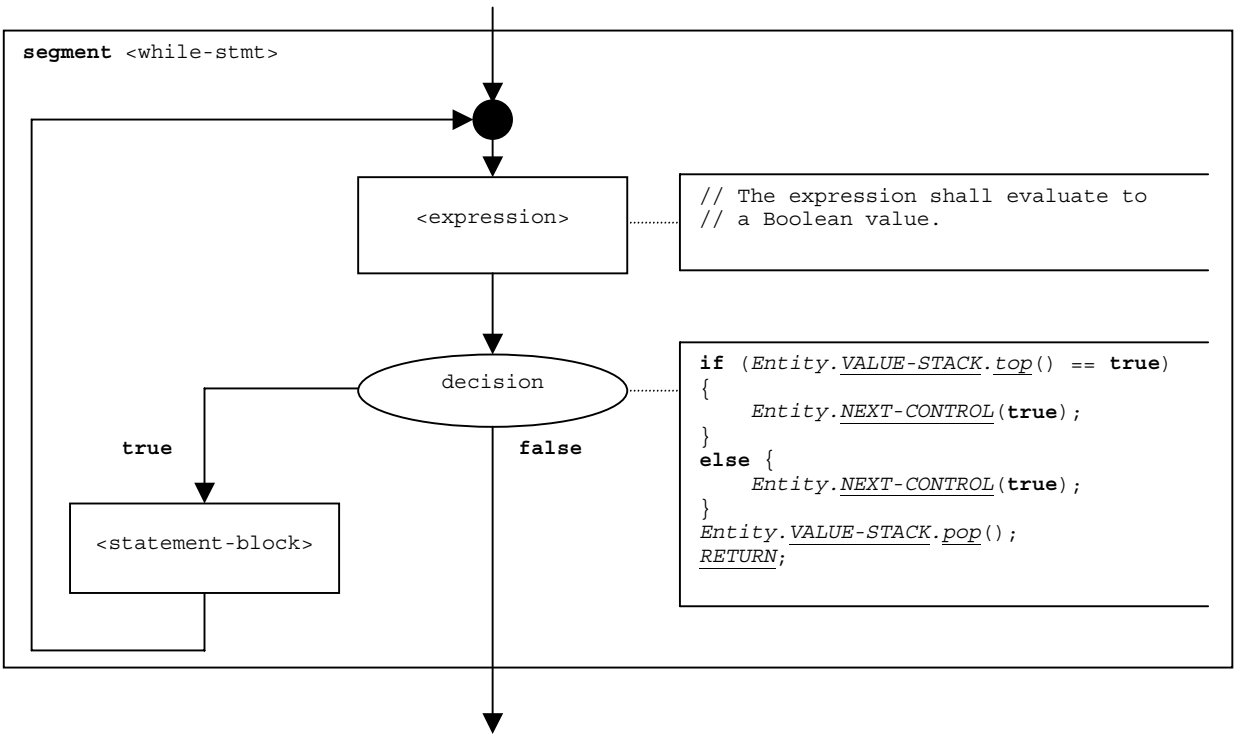


Figure 140 – Flow graph segment <while-stmt>

10 Lists of operational semantic components

10.1 Functions and states

Name	Description	Clause
ACT-DURATION	Duration with which an active timer has been started	8.3.2.4
add	List operation: adds an item as first element to a list	8.3.1.1
ADD-CON	Adds a connection to a port state	8.3.3.2
ALL-ENTITY-STATES	Component states in module state	8.3.1
ALL-PORT-STATES	Port states in module state	8.3.1
append	List operation: appends an item as last element to a list	8.3.1.1
APPLY-OPERATOR	Application of operators like +, – or /	8.6.2
change	List operation: changes all elements of a list	8.3.1.1
clear	Stack operation 'clear': clears a stack	8.3.2.1
clear	Queue operation 'clear': removes all elements from a queue	8.3.3.2
clear-until	Stack operation 'clear-until': pops items until a specific item is top element in the stack.	8.3.2.1
CONNECTIONS-LIST	List of connections of a port	8.3.3
CONSTRUCT-ITEM	Constructs an item to be sent	8.4.4
CONTINUE-COMPONENT	The actual component continues its execution	8.6.2
CONTROL-STACK	Stack of flow graph nodes denoting the actual control state of an entity	8.3.2

Name	Description	Clause
DATA-STATE	Data state in an entity state	8.3.2
DEF-DURATION	Default Duration of a timer	8.3.2.4
DEFAULT-LIST	List of active defaults in an entity state	8.3.2
DEFAULT-POINTER	Points to the actual default during the default evaluation	8.3.2
DEL-CON	Deletes a connection from a port state	8.3.3.2
DEL-ENTITY	Deletes an entity from a module state	8.3.4
DEL-TIMER-SCOPE	Deletes a timer scope	8.3.2.5
DEL-VAR-SCOPE	Deletes a variable scope	8.3.2.3
delete	List operation: deletes an item from a list	8.3.1.1
dequeue	Queue operation: deletes the first element from a queue	8.3.3.2
DONE	Identifiers of terminated test components (part of module state)	8.3.1
E-VERDICT	Local test verdict of a test component	8.3.2
enqueue	Queue operation: puts an item as last element into a queue	8.3.3.2
first	Queue operation 'first': returns the first element of a queue	8.3.3.2
first	List operation: returns the first element of a list	8.3.1.1
GET-FLOW-GRAPH	Retrieves the start node of a flow graph	8.2.7
GET-PORT	Retrieves a port reference	8.3.3.2
GET-REMOTE-PORT	Retrieves the reference of a remote port	8.3.3.2
GET-TIMER-LOC	Retrieves location of a timer	8.3.2.5
GET-UNIQUE-ID	Returns a new unique identifier when it is called	8.6.2
GET-VAR-LOC	Retrieves location of a variable	8.3.2.3
INIT-CALL-RECORD	Initializes variables for parameters for procedure-based communication in the actual scope unit of the test component	8.5.1
INIT-FLOW-GRAPHS	Initializes the flow graph handling	8.6.2
INIT-TIMER	Creates a new timer binding	8.3.2.5
INIT-TIMER-LOC	Creates a new timer binding with an existing location	8.3.2.5
INIT-TIMER-SCOPE	Initializes a new timer scope	8.3.2.5
INIT-VAR	Creates a new variable binding	8.3.2.3
INIT-VAR-LOC	Creates a new variable binding with an existing location	8.3.2.3
INIT-VAR-SCOPE	Initializes a new variable scope	8.3.2.3
length	List operation: returns the length of a list	8.3.1.1
M-CONTROL	Identifier of module control in module state	8.3.1
MATCH-ITEM	Checks if a received message, call, reply or exception matches with a receiving operation	8.4.5
member	List operation: checks if an item is element of a list	8.3.1.1
MTC	Reference to MTC in module state	8.3.1
NEW-CALL-RECORD	Creates a call record for a function call	8.5.1
NEW-ENTITY	Creates a new entity state	8.3.2.1
NEW-PORT	Creates a new port	8.3.3.2
NEXT	Retrieves the successor node of a given node in a flow graph	8.1.6

Name	Description	Clause
next	List operation: returns next element in a list	8.3.1.1
NEXT-CONTROL	Pops the top flow graph node from the control stack and pushes the next flow graph node onto the control stack	8.3.2.1
OWNER	Owner of a port	8.3.3
pop	Stack operation 'pop': pops an item from a stack	8.3.2.1
PORT-NAME	Name of a port	8.3.3
push	Stack operation 'push': pushes an item onto a stack	8.3.2.1
random	List operation: returns randomly an element of a list	8.3.1.1
REMOTE-ENTITY	Remote entity in a connection in a port state	8.3.3.1
REMOTE-PORT-NAME	Name of a port in a connection in a port state	8.3.3.1
RETRIEVE-INFO	Retrieves information from a received message, call, reply or exception	8.4.6
RETURN	Returns the control to the module evaluation procedure	8.6.2
SNAP-ACTIVE	Number of active test components when the MTC takes a snapshot (part of module state)	8.3.1
SNAP-DONE	List of terminated test components at the time when a snapshot is taken	8.3.2
SNAP-PORTS	Provides the snapshot functionality, i.e., updates the SNAP-VALUE	8.3.3.2
SNAP-STATUS	Snapshot status of a timer	8.3.2.4
SNAP-TIMER	Provides the snapshot functionality and updates SNAP-VALUE and SNAP-STATUS	8.3.2.5
SNAP-VALUE	Snapshot value of a timer	8.3.2.4
SNAP-VALUE	For snapshot semantics, updated when a snapshot is taken	8.3.3
STATUS	Status (**ACTIVE**, **SNAPSHOT**, **REPEAT** or **BLOCKED**) of module control or a test component	8.3.2
STATUS	Status (**IDLE**, **RUNNING** or **TIMEOUT**) of a timer	8.3.2.4
STATUS	Status (**STARTED** or **STOPPED**) of a port	8.3.3
TC-VERDICT	Test case verdict in module state	8.3.1
TIME-LEFT	Time a running timer has left to run before it times out	8.3.2.4
TIMER-GUARD	Timer that guards `execute` statements and `call` operations	8.3.2
TIMER-NAME	Name of a timer	8.3.2.4
TIMER-SET	Setting values of a timer	8.3.2.5
TIMER-STATE	Timer state in an entity state	8.3.2
top	Stack operation 'top': returns the top item from a stack	8.3.2.1
UPDATE-REMOTE-REFERENCES	Updates timers and variables with the same location in different entities to the same value	8.3.4
VALUE	Value of a variable	8.3.2.2
VALUE-QUEUE	Port queue	8.3.3
VALUE-STACK	Stack of values for the storage of results of expressions, operands, operations and functions	8.3.2
VAR-NAME	Name of a variable	8.3.2.2
VAR-SET	Setting the value of a variable	8.3.2.3

Name	Description	Clause
DYNAMIC-ERROR	Describes the occurrence of a dynamic error	8.6.2
<identifier>	Unique identifier of a test component	8.3.2
<location>	Supports scope units, reference and timer parameters. Represents a storage location for timers and variables.	8.3.2.2, 8.3.2.4

10.2 Special keywords

Keyword	Description	Clause
ACTIVE	*STATUS* of an entity state	8.3.2
BLOCKED	*STATUS* of an entity state	8.3.2
IDLE	*STATUS* of a timer state	8.3.2.4
MARK	Used as mark for *VALUE-STACK*	8.3.2
NONE	Used to describe an undefined value	8.3.2.3, 8.3.2.5, 8.3.3.2
NULL	Symbolic value for pointer and pointer-like types to indicate that nothing is addressed	8.3.1.1, 8.3.2.1, 8.3.3, 8.3.3.2, 8.6.1.1
REPEAT	*STATUS* of an entity state	8.3.2
RUNNING	*STATUS* of a timer state	8.3.2.4
SNAPSHOT	*STATUS* of an entity state	8.3.2
STARTED	*STATUS* of a port	8.3.3
STOPPED	*STATUS* of a port	8.3.3
TIMEOUT	*STATUS* of a timer state	8.3.2.4

10.3 Flow graphs of TTCN-3 behaviour descriptions

	Reference	
	Figure	Clause
Module control	18	8.2.2
Test cases	19	8.2.3
Functions	20	8.2.4
Altsteps	21	8.2.5
Component type definitions	22	8.2.6

10.4 Flow graph segments

Identifier	Related TTCN-3 construct	Reference	
		Figure	Clause
<action-stmt>	**action** statement	36	9.1
<activate-stmt>	**activate** statement	37	9.2
<alive-component-op>	**alive** component operation	37-a	9.2a
<alive-comp-act>	**alive** component operation	37-b	9.2a.1

Identifier	Related TTCN-3 construct	Reference	
		Figure	Clause
<alive-comp-snap>	**alive** component operation	37-c	9.2a.2
<alt-stmt>	**alt** statement	38	9.3
<altstep-call>	invocation of an altstep	44	9.4
<altstep-call-branch>	**alt** statement	41	9.3.3
<assignment-stmt>	assignment	45	9.5
<b-call-with-duration>	**call** operation	52	9.6.4
<b-call-without-duration>	**call** operation	51	9.6.3
<blocking-call-op>	**call** operation	47	9.6
<call-op>	**call** operation	46	9.6
<call-reception-part>	**call** operation	53	9.6.5
<catch-op>	**catch** operation	55	9.7
<catch-timeout-exception>	**call** operation	54	9.6.6
<check-op>	**check** operation	56	9.8
<check-with-sender>	**check** operation	57	9.8.1
<check-without-sender>	**check** operation	58	9.8.2
<clear-port-op>	**clear** port operation	59	9.9
<connect-op>	**connect** operation	60	9.10
<constant-definition>	constant definition	61	9.11
<create-op>	**create** operation	62	9.12
<deactivate-all-defaults>	**deactivate** statement	63-c	9.13.2
<deactivate-one-default>	**deactivate** statement	63-b	9.13.1
<deactivate-stmt>	**deactivate** statement	63-a	9.13
<default-evocation>	**alt** statement	43	9.3.5
<disconnect-op>	**disconnect** operation	64	9.14
<do-while-stmt>	**do-while** statement	65	9.15
<done-component-op>	**done** component operation	66	9.16
<dynamic-error>	expression	74-b	9.18.5
<else-branch>	**alt** statement	42	9.3.4
<execute-stmt>	**execute** statement	67	9.17
<execute-timeout>	**execute** statement	69	9.17.2
<execute-without-timeout>	**execute** statement	68	9.17.1
<expression>	expression	70	9.18
<finalise-component-init>	used in component type definitions	75	9.19
<for-stmt>	**for** statement	79	9.23
<func-op-call>	expression	73	9.18.3
<function-call>	call of a function	80	9.24
<getcall-op>	**getcall** operation	86	9.25
<getreply-op>	**getreply** operation	87	9.26

Identifier	Related TTCN-3 construct	Reference	
		Figure	Clause
<getverdict-op>	**getverdict** operation	88	9.27
<goto-stmt>	**goto** statement	89	9.28
<if-else-stmt>	**if-else** statement	90	9.29
<init-component-scope>	used in component type definitions	76	9.20
<init-scope-with-runs-on>	used in component type definitions	76-a	9.20a
<init-scope-without-runs-on>	used in component type definitions	76-b	9.20b
<killed-op>	**killed** component operation	90-g	9.29c
<kill-all-comp>	**kill** component operation	90-d	9.29a.3
<kill-component>	**kill** component operation	90-c	9.29a.2
<kill-component-op>	**kill** component operation	90-a	9.29a
<kill-control>	**kill** execution statement	90-f	9.29b.1
<kill-exec-stmt>	**kill** execution statement	90-e	9.29b
<kill-mtc>	**kill** component operation	90-b	9.29a.1
<label-stmt>	**label** statement	91	9.30
<lit-value>	expression	71	9.18.1
<log-stmt>	**log** statement	92	9.31
<map-op>	**map** operation	93	9.32
<mtc-op>	**mtc** operation	94	9.33
<nb-call-without-receiver>	**call** operation	50	9.6.2
<nb-call-with-multiple-receivers>	**call** operation	49-a	9.6.1a
<nb-call-with-one-receiver>	**call** operation	49	9.6.1
<non-blocking-call-op>	**call** operation	48	9.6
<operator-appl>	expression	74-a	9.18.4
<parameter-handling>	handling of parameters of functions, altsteps and test cases	77	9.21
<port-declaration>	port declaration	95	9.34
<predef-ext-func-call>	call of a function (call of a pre-defined or external function)	85	9.24.5
<raise-op>	**raise** operation	96	9.35
<raise-with-one-receiver-op>	**raise** operation	97	9.35.1
<raise-with-multiple-receivers-op>	**raise** operation	97-a	9.35.1a
<raise-without-receiver-op>	**raise** operation	98	9.35.2
<read-timer-op>	**read** timer operation	99	9.36
<receive-assignment>	**receive** operation	103	9.37.3

Identifier	Related TTCN-3 construct	Reference	
		Figure	Clause
<receive-op>	**receive** operation	100	9.37
<receive-with-sender>	**receive** operation	101	9.37.1
<receive-without-sender>	**receive** operation	102	9.37.2
<receiving-branch>	**alt** statement	40	9.3.2
<ref-par-var-calc>	call of a function (handling of reference parameters)	82	9.24.2
<ref-par-timer-calc>	call of a function (handling of timer parameters)	83	9.24.3
<ref-par-port-calc>	call of a function (handling of port parameters)	83-a	9.24.3a
<repeat-stmt>	**repeat** statement	104	9.38
<reply-op>	**reply** operation	105	9.39
<reply-with-one-receiver-op>	**reply** operation	106	9.39.1
<reply-with-multiple-receivers-op>	**reply** operation	106-a	9.39.1a
<reply-without-receiver-op>	**reply** operation	107	9.39.2
<return-stmt>	**return** statement	108	9.40
<return-with-value>	**return** statement	109	9.40.1
<return-without-value>	**return** statement	110	9.40.2
<running-component-op>	component **running** operation	111	9.41
<running-comp-act>	component **running** operation	112	9.41.1
<running-comp-snap>	component **running** operation	113	9.41.2
<running-timer-op>	timer **running** operation	114	9.42
<self-op>	**self** operation	115	9.43
<send-op>	**send** operation	116	9.44
<send-with-one-receiver-op>	**send** operation	117	9.44.1
<send-with-multiple-receivers-op>	**send** operation	117-a	9.44.1a
<send-without-receiver-op>	**send** operation	118	9.44.2
<setverdict-op>	**setverdict** operation	119	9.45
<start-component-op>	**start** component operation	120	9.46
<start-port-op>	**start** port operation	121	9.47
<start-timer-op>	**start** timer operation	122	9.48
<start-timer-op-default>	**start** timer operation	123	9.48.1
<start-timer-op-duration>	**start** timer operation	124	9.48.2
<statement-block>	block of statements in compound statements	78	9.22
<stop-component-op>	**stop** component operation	125	9.49
<stop-alive-component>	**stop** component operation (stop single test component)	126	9.49.2
<stop-all-comp>	**stop** component operation (all component.stop)	127	9.49.3

Identifier	Related TTCN-3 construct	Reference	
		Figure	Clause
<stop-exec-stmt>	**stop** execution statement	128	9.50
<stop-port-op>	**stop** port operation	129	9.51
<stop-timer-op>	**stop** timer operation	130	9.52
<system-op>	**system** operation	131	9.53
<take-snapshot>	**alt** statement	39	9.3.1
<timeout-timer-op>	**timeout** operation	135	9.55
<timer-declaration>	timer declaration	132	9.54
<timer-decl-default>	timer declaration	133	9.54.1
<timer-decl-no-def>	timer declaration	134	9.54.2
<unmap-op>	**unmap** operation	136	9.56
<user-def-func-call>	call of a function (call of a user-defined function)	84	9.24.4
<value-par-calculation>	call of a function (handling of value parameters)	81	9.24.1
<var-declaration-init>	variable declaration	138	9.57.1
<var-declaration-undef>	variable declaration	139	9.57.2
<var-value>	expression	72	9.18.2
<variable-declaration>	variable declaration	137	9.57
<while-stmt>	**while** statement	140	9.58

SERIES OF ITU-T RECOMMENDATIONS

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

Printed in Switzerland
Geneva, 2008