INTERNATIONAL TELECOMMUNICATION UNION

ITU-T Z.140

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

(07/2001)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

Formal description techniques (FDT)

The Tree and Tabular Combined Notation version 3

CAUTION !

PREPUBLISHED RECOMMENDATION

This prepublication is an unedited version of a recently approved Recommendation. It will be replaced by the published version after editing. Therefore, there will be differences between this prepublication and the published version.

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications. 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.

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/had not] received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementors are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

Recommendation ITU-T Z.140

The Tree and Tabular Combined Notation version 3 (TTCN-3): Core Language

Contents

1 Scope

The present document defines the Core Language of TTCN Version 3 (or TTCN-3). TTCN-3 can be used for the specification of all types of reactive system tests over a variety of communication ports. Typical areas of application are protocol testing (including mobile and Internet protocols), service testing (including supplementary services), module testing, testing of CORBA based platforms, APIs etc. TTCN-3 is not restricted to conformance testing and can be used for many other kinds of testing including interoperability, robustness, regression, system and integration testing. The specification of test suites for physical layer protocols is outside the scope of the present document.

TTCN-3 is intended to be used for the specification of test suites which are independent of test methods, layers and protocols. Various presentation formats are defined for TTCN-3 such as a tabular presentation format [1] and a graphical presentation format [2]. The specification of these formats is outside the scope of the present document.

The present document defines a normative way of using of ASN.1 as defined in the ITU-T Recommendation X.680 series [7], [8], [9] and [10] with TTCN-3. The harmonization of other languages with TTCN-3 is outside the scope of the present document.

While the design of TTCN-3 has taken the eventual implementation of TTCN-3 translators and compilers into consideration the means of realization of executable test suites (ETS) from abstract test suites (ATS) is outside the scope of the present document.

2 References

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

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

[1] Recommendation ITU-T Z.141: “The Tree and Tabular Combined Notation version 3 (TTCN-3): Tabular Presentation Format”.

[2] Draft new Recommendation ITU-T Z.142: “The Tree and Tabular Combined Notation version 3 (TTCN-3): Graphical Format”.

[3] ISO/IEC 9646-1 (1994): "Information technology - Open systems interconnection - Conformance testing methodology and framework - Part 1: General Concepts".

[4] ISO/IEC 9646-3 (1998): "Information technology - Open systems interconnection - Conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation (TTCN) Edition 2".

[5] ISO/IEC 646 (1990): "Information technology - ISO 7-bit coded character set for information exchange".

[6] ISO/IEC 10646 (1993): "Information technology - Universal Multiple Octet-Coded Character Set (UCS) - Part 1: Architecture and basic multilingual plane".

[7] ITU-T Recommendation X.680 (1997): "Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation".

[8] ITU-T Recommendation X.681 (1997): "Information technology - Abstract Syntax Notation One (ASN.1): Information object specification".

[9] ITU-T Recommendation X.682 (1997): "Information technology - Abstract Syntax Notation One (ASN.1): Constraint specification".

[10] ITU-T Recommendation X.683 (1997): " Information technology - Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications"

[11] ITU-T Recommendation X.690 (1997): "Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)".

[12] ITU-T Recommendation X.691 (1997): "Information technology - ASN.1 encoding rules: Specification of Packed Encoding Rules (PER)".

3 Definitions and abbreviations

3.1 Definitions

For the purposes of the present document, the following terms and definitions apply:

compatible type: TTCN-3 is not strongly typed but the language does require type compatibility. Variables, constants, templates etc. have compatible types if they resolve to the same base type and, in the case of assignments, matching etc., no sub-typing (e.g., ranges, length restrictions) is violated

communication port: abstract mechanism facilitating communication between test components
A communication port is modelled as a FIFO queue in the receiving direction. Ports can be message-based, procedure-based or a mixture of the two.

exception: in cases of synchronous communication an exception (if defined) is raised by an answering entity if it cannot answer a remote procedure call with the normal expected response

test suite: TTCN-3 module that either explicitly or implicitly through import statements completely specifies all definitions and behaviour descriptions necessary to define a complete set of test cases

test system interface: test component that provides a mapping of the ports available in the (abstract) TTCN-3 test system to those offered by a real test system

type parameterization: ability to pass a type as an actual parameter into a parameterized object
This actual type parameter then completes the type specification of that object. Note that the parameter is not a value of a type but the type itself.

3.1.1 Definitions from ISO/IEC-9646-1

For the purposes of the present document, the following terms and definitions given in ISO/IEC-9646-1 [3] apply:

Implementation Conformance Statement (ICS)

Implementation eXtra Information for Testing (IXIT)

Implementation Under Test (IUT)

System Under Test (SUT)

test case

test case error

test system

3.1.2 Definitions from ISO/IEC-9646-3

For the purposes of the present document, the following terms and definitions given in ISO/IEC-9646-3 [4] apply:

Main Test Component (MTC)

Parallel Test Component (PTC)

snapshot semantics

3.2 Abbreviations

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

API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
ASP	Abstract service Primitive
ATS	Abstract Test Suite
BNF	Backus-Nauer Form
CORBA	Common Object Request Broker Architecture
ETS	Executable Test Suite
FIFO	First In First Out
IDL	Interface Description Language
IUT	Implementation Under Test
MTC	Master Test Component
PDU	Protocol Data Unit
PTC	Parallel Test Component
(P)ICS	(Protocol) Implementation Conformance Statement
(P)IXIT	(Protocol) Implementation eXtra Information for Testing
SUT	System Under Test
TTCN	Tree and Tabular Combined Notation

4 Introduction

TTCN-3 is a flexible and powerful language applicable to the specification of all types of reactive system tests over a variety of communication interfaces. Typical areas of application are protocol testing (including mobile and Internet protocols), service testing (including supplementary services), module testing, testing of CORBA based platforms, API testing etc. TTCN-3 is not restricted to conformance testing and can be used for many other kinds of testing including interoperability, robustness, regression, system and integration testing.

From a syntactical point of view TTCN-3 is very different from earlier versions of the language as defined in ISO/IEC 9646-3 [4]. However, much of the well-proven basic functionality of TTCN has been retained, and in some cases enhanced. TTCN-3 includes the following essential characteristics:

- the ability to specify dynamic concurrent testing configurations;
- operations for synchronous and asynchronous communication;
- the ability to specify encoding information and other attributes (including user extensibility);
- the ability to specify data and signature templates with powerful matching mechanisms;
- type and value parameterization;
- the assignment and handling of test verdicts;
- test suite parameterization and test case selection mechanisms;
- combined use of TTCN-3 with ASN.1 (and potential use with other languages such as IDL);
- well-defined syntax, interchange format and static semantics;
- different presentation formats (e.g., tabular and graphical presentation formats);
- a precise execution algorithm (operational semantics).

4.1 The core language and presentation formats

Historically, TTCN has always been associated with conformance testing. In order to open the language to a wider range of testing applications in both the standards domain and the industrial domain the present document separates the specification of TTCN-3 into several parts. The first part, defined in the present document, is the core language. The second part, defined in Recommendation Z.141 [1], is the tabular presentation format, similar in appearance and functionality to earlier versions of TTCN. The third part, defined in draft new Recommendation Z.142 [2] is the graphical presentation format.

The core language serves three purposes:

a) as a generalized text-based test language in its own right;

b) as a standardized interchange format of TTCN test suites between TTCN tools;

c) as the semantic basis (and where relevant, the syntactical basis) for various presentation formats.

The core language may be used independently of the presentation formats. However, neither the tabular format nor the graphical format can be used without the core language. Use and implementation of these presentation formats shall be done on the basis of the core language.

The tabular format and the graphical format are the first in an anticipated set of different presentation formats. These other formats may be standardized presentation formats or they may be proprietary presentation formats defined by TTCN-3 users themselves. These additional formats are not defined in the present document.

TTCN-3 is fully harmonized with ASN.1 which may optionally be used with TTCN-3 modules as an alternative data type and value syntax. Use of ASN.1 in TTCN-3 modules is defined in annex E of the present document. The approach used to combine ASN.1 and TTCN-3 could be applied to support the use of other type and value systems with TTCN-3. However, the details of this are not defined in the present document.

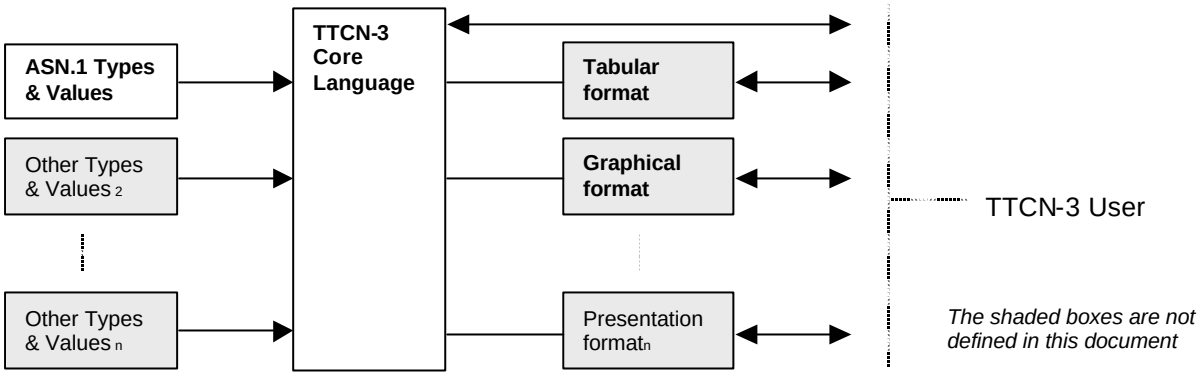


Figure 1: User's view of the core language and the various presentation formats

The core language is defined by a complete syntax (see annex A) and operational semantics (see annex B). It contains minimal static semantics (provided in the body of the present document and in annex A) which do not restrict the use of the language due to some underlying application domain or methodology. Functionality of previous versions of TTCN, such as test suite indexes, which can be achieved using proprietary tools is not part of TTCN-3.

5 Basic language elements

The top-level unit of TTCN-3 is a module. A module cannot be structured into sub-modules. A module can import definitions from other modules. Modules can have parameter lists to give a form of test suite parameterization similar to the PICS and PIXIT parameterization mechanisms of TTCN-2.

A module consists of a definitions part and a control part. The definitions part of a module defines test components, communication ports, data types, constants, test data templates, functions, signatures for procedure calls at ports, test cases etc.

The control part of a module calls the test cases and controls their execution. The control part may also declare (local) variables etc. Program statements (such as **`if-else`** and **`do-while`**) can be used to specify the selection and execution order of individual test cases. The concept of global variables is not supported in TTCN-3.

TTCN-3 has a number of pre-defined basic data types as well as structured types such as records, sets, unions, enumerated types and arrays. As an option, ASN.1 types and values may be used with TTCN-3 by importation.

A special kind of data value called a template provides parameterization and matching mechanisms for specifying test data to be sent or received over the test ports. The operations on these ports provide both asynchronous and synchronous communication capabilities. Procedure calls may be used for testing implementations which are not message based.

Dynamic test behaviour is expressed as test cases. TTCN-3 program statements include powerful behaviour description mechanisms such as alternative reception of communication and timer events, interleaving and default behaviour. Test verdict assignment and logging mechanisms are also supported.

Finally, most TTCN-3 language elements may be assigned attributes such as encoding information and display attributes. It is also possible to specify (non-standardized) user-defined attributes.

Table 1: Overview of TTCN-3 language elements

Language element	Associated keyword	Specified in module definitions	Specified in module control	Specified in functions/test cases
TTCN-3 module definition	**module**			
Import of definitions from other module	**import**	Yes		
Grouping of definitions	**group**	Yes		
Data type definitions	**type**	Yes		
Communication port definitions	**port**	Yes		
Test component definitions	**component**	Yes		
Signature definitions	**signature**	Yes		
External function/constant definitions	**external**	Yes		
Constant definitions	**const**	Yes	Yes	Yes
Data/signature template definitions	**template**	Yes		
Function definitions	**function**	Yes		
Named alternative definitions	**named alt**	Yes		
Test case definitions	**testcase**	Yes		
Variable declarations	**var**		Yes	Yes
Timer declarations	**timer**		Yes	Yes

5.1 Definitions, instances and declarations

In the present document the term declaration is used in a general manner to cover making a static definition or creating some kind of dynamic instantiation where a name is given to a TTCN-3 object. For example, types and constants are defined and a statement such as calling a function or declaring a variable is an instantiation. In both cases these actions can be referred to as making a declaration.

5.2 Ordering of language elements

Generally, the order in which declarations can be made and the mixing of declarations with program statements is arbitrary. However, inside a statement block, such as a branch of an **`if-else`** statement, all declarations (if any), shall be made at the beginning of the statement block only.

EXAMPLE:

```
// This is a legal mixing of TTCN-3 declarations
:
var MyVarType MyVar2 := 3;
const integer MyConst:= 1;
if (x > 10)
{
    var integer MyVar1:= 1;
    :
    MyVar1:= MyVar1 + 10;
    :
}

:
```

5.2.1 Forward references

Definitions in the module definitions part may be made in any order and while forward references should be avoided (for readability reasons) this is not mandatory. For example, recursive elements, such as functions that call other functions and module parameterization, may lead to unavoidable forward references.

Forward references are only allowed for declarations in the module definitions part. Forward references shall never be made inside the module control part, test case definitions, functions and named alternatives. This means forward references to local variables, local timers and local constants shall never occur.

5.3 Parameterization

TTCN-3 supports both *type* parameterization and *value* parameterization according to the following limitations:

a) language elements which cannot be parameterized are: **`const`**, **`var`**, **`timer`**, **`control`**, **`group`** and **`import`**;

b) the language element **`module`** allows *static* value parameterization to support test suite parameters i.e., this parameterization may or may not be resolvable at compile-time but shall be resolved by the commencement of run-time (i.e., *static* at run-time). This means that, at run-time, module parameter values are globally visible but not changeable;

c) all user-defined **`type`** definitions (including the structured type definitions such as **`record`**, **`set`** etc.), and the special configuration type **`address`** support *static* type and *static* value parameterization i.e., this parameterization shall be resolved at compile-time;

d) the language elements **`signature`**, **`testcase`** and **`function`** support *dynamic* value parameterization (i.e., this parameterization shall be resolvable at run-time);

e) named alternatives support *dynamic* value parameterization (i.e., this parameterization shall be resolvable at run-time). Since named alternatives are not a scope unit, the defined formal parameters are simply substituted by the given actual parameters when the (macro) expansion of the **`named alt`** is performed.

A summary of which language elements can be parameterized and what can be passed to them as parameters is given in table 2.

Table 2: Overview of parameterizable TTCN-3 language elements

Keyword	Type Parameterization	Value Parameterization	Types of values allowed to appear in formal/actual parameter lists
module		Static at start of run-time	*Values of:* all basic types, all user-defined types and **`address`** type.
type	Static at compile-time	Static at compile-time	*Values of:* all basic types, all user-defined types and **`address`** type. Note: **record of**, **set of**, **enumerated, port, component** and **subtype** definitions do not allow parameterization.
template		Dynamic at run-time	*Values of:* all basic types, all user-defined types, **`address`** type, **`component`** type and **`template.`**
function		Dynamic at run-time	*Values of:* all basic types, all user-defined types, **`address`** type, **`component`** type, **`port`** type, **`template`**.and **`timer`**.
testcase		Dynamic at run-time	*Values of:* all basic types and of all user-defined types, **`address`** typeand **`template`**.
signature		Dynamic at run-time	*Values of:* all basic types, all user-defined types and **`address`** type and **`component`** type.
named alt		Static macro expansion	*Values of:* all basic types, all user-defined types, **`address`** type, **`component`** type, **`port`** type, **`template`** and **`timer`**.

NOTE : Examples of syntax and specific use of parameterization with the different language elements are given in the relevant clauses in the present document.

5.3.1 Parameter passing by reference and by value

By default, all parameters of basic types, basic string types, user-defined structured types, address type and component type are passed by value. This may optionally be denoted by the keyword **in**. To pass parameters of the mentioned types by reference the keywords **out** or **inout** shall be used.

Timers and ports are always passed by reference and are identified by the keywords **timer** and **port**. The keyword **inout** may optionally be usedto denote passing by reference.

5.3.1.1 Parameters passed by reference

Passing parameters by reference has the following limitations:

a) only the formal parameter lists to **function**, **signature** and **testcase** may contain pass-by-reference parameters;

NOTE: There are further restrictions on how to use pass-by-reference parameters in signatures (see clause 22).

b) the actual parameters shall only be variables (e.g., not constants or templates);

c) only value parameters (i.e., not type parameters) shall be passed by reference.

EXAMPLE:

```
function MyFunction(inout boolean MyReferenceParameter){ … };
// MyReferenceParameter is passed by reference. The actual parameter can be read and set
// from within the function

function MyFunction(out boolean MyReferenceParameter){ … };
// MyReferenceParameter is passed by reference. The actual paramter can only be set
// from within the function
```

5.3.1.2 Parameters passed by value

Actual parameters that are passed by value may be variables as well as constants, temp lates etc.

```
function MyFunction(in template MyTemplateType MyValueParameter){ … };
// MyValueParameter is passed by value, the in keyword is optional
```

5.3.2 Formal and actual parameter lists

The number of elements and the order in which they appear in an actual parameter list shall be the same as the number of elements and their order in which they appear in the corresponding formal parameter list. Furthermore, the type of each actual parameter shall be compatible with the type of each corresponding formal parameter.

EXAMPLE:

```
// A function definition with a formal parameter list
function MyFunction(integer FormalPar1, boolean FormalPar2, bitstring FormalPar3) { … }

// A function call with an actual parameter list
MyFunction(123, true,'1100'B);
```

5.3.3 Empty formal parameter list

If the formal parameter list of a parameterizable TTCN-3 language element that is function-like (i.e., **function**, **testcase**, **signature, named alt** or **external** function) is empty then the empty parentheses shall be included both in the declaration and in the invocation of that element. In all other cases the empty parentheses shall be omitted.

EXAMPLE:

```
// A function definition with an empty parameter list shall be written as
function MyFunction(){ … }

// A record definition with an empty parameter list shall be written as
type record MyRecord { … }
```

5.3.4 Nested parameter lists

Generally, all parameterized entities specified as an actual parameter shall have their own parameters resolved in the actual parameter list.

EXAMPLE:

```
// Given the message definition
type record MyMessageType
{
    integer    field1,
    charstring field2,
    boolean    field3
}

// A message template might be
template MyMessageType MyTemplate(integer MyValue) :=
{
    field1 := MyValue,
    field2 := pattern "abc*xyz",
    field3 := true
}

// A testcase parameterized with a template might be
testcase TC001(template MyMessageType RxMsg) runs on PTC1 system TS1
{   :
    MyPCO.receive(RxMsg);
}

// When the test case is called in the control part and the parameterized template is
// used as an actual parameter, the actual parameters for template must be provided
control
{   :
    TC001(MyTemplate(7));
    :
}
```

5.4 Scope rules

TTCN-3 provides five basic units of scope:

a) modules;

NOTE: There are additional scoping rules for groups (see clause 7.3.1).

b) control part of a module;

c) functions;

d) test cases;

e) statement blocks within control, functions and test cases.

Each unit of scope consists of (optional) declarations plus some form of (optional) functional description. All units of scope, except modules, are hierarchical, with each level of hierarchy defining its own local scope. Declarations in a higher level of scope are visible to the lower levels (within the same hierarchy of scope). Declarations in a lower level of scope are not visible to those in a higher scope.

EXAMPLE:

```
module MyModule
{   :
    const integer MyConst := 0; // MyConst is visible to MyBehaviourA and MyBehaviourB
    :
    function MyBehaviourA()
    {   :
        const integer A := 1; // The constant A is only visible to MyBehaviourA
        :
    }

    function MyBehaviourB()
    {   :
        const integer B := 1; // The constant B is only visible to MyBehaviourB
        :

    }
}
```

5.4.1 Scope and overloading of identifiers

TTCN-3 does not support overloading of identifiers i.e., all identifiers in the same scope hierarchy shall be unique. This means that a declaration in a lower level of scope shall not re-use the same identifier as a declaration in a higher level of scope (and in the same scope hierarchy).

EXAMPLE:

```
module MyModule
{   :
    const integer A := 1;
    :
    function MyBehaviourA()
    {   :
        const integer A := 1; // Is NOT allowed
        :
        if(…)
        {   :
            const boolean A := true; // Is NOT allowed
            :
        }
    }
}

// The following IS allowed as the constants are not declared in the same scope hierarchy
// (assuming there is no declaration of A in module header)
function MyBehaviourA()
{   :
    const integer A := 1;
    :
}

function MyBehaviourB()
{   :
    const integer A := 1;
    :
}
```

5.4.2 Scope of formal parameters

The scope of the formal parameters in a parameterized language element (e.g., in a function call) shall be restricted to the definition in which the parameters appear and to the lower levels of scope in the same scope hierarchy. That is they follow the normal scope rules (see clause 5.4). The rules of identifier overloading (see clause 5.4.1) shall also apply to formal parameters.

5.5 Identifiers and keywords

TTCN-3 identifiers are case sensitive and TTCN-3 keywords shall be written in all lowercase letters (see annex A).

6 Types and values

TTCN-3 supports a number of predefined basic types. These basic types include ones normally associated with a programming language, such as **integer**, **boolean** and string types, as well as some TTCN-3 specific ones such as **objid** and **verdicttype**. Structured types such as **record** types, **set** types and **enumerated** types can be constructed from these basic types.

Special types associated with configurations such as **address**, **port** and **component** may be used to define the architecture of the test system (see clause 21).

The TTCN-3 types are summarized in table 3.

Table 3: Overview of TTCN-3 types

Class of type	Keyword	Sub-type
Basic types	**integer**	range, list
	char	range, list
	universal char	range, list
	float	list
	boolean	list
	objid	list
	verdicttype	list
Basic string types	**bitstring**	list, length
	hexstring	list, length
	octetstring	list, length
	charstring	list, length
	universal charstring	list, length
User-defined structured types	**record**	list
	record of	list
	set	list
	set of	list
	enumerated	list
	union	list
Special configuration types	**address**	
	port	
	component	

6.1 Basic types and values

TTCN-3 supports the following basic types:

a) **integer:** a type with distinguished values which are the positive and negative whole numbers, including zero.

Values of integer type shall be denoted by one or more digits; the first digit shall not be zero unless the value is 0; the value zero shall be represented by a single zero.

b) **char:** a type whose distinguished values are characters from ISO/IEC 646 [5].

Values of the type **char** may be given enclosed in double quotes (") or calculated using a predefined conversion function with the positive integer value of their encoding as argument.

An order among the values of type **char** is defined by the integer value of their encoding, i.e., the relational operators ==, <, >, !=, >= and <= can be used to compare values of type **char**.

c) **universal char**: a type whose distinguished values are single characters from ISO/IEC 10646 [6].

Values of the type **universal char** may be given enclosed in double quotes (") or calculated using a predefined conversion function with the positive integer value of their encoding as argument.

An order among the values of type **char** is defined by the integer value of their encoding, i.e., the relational operators ==, <, >, !=, >= and <= can be used to compare values of type **universal char**.

d) **`float`:** a type to describe floating-point numbers.

Floating point numbers are represented as: *<mantissa>*<base>*$^{<exponent>}$

Where *<mantissa>* a positive or negative integer, *<base>* a positive integer (in most cases 2, 10 or 16) and *<exponent>* a positive or negative integer.

The floating-point number representation is restricted to a base with the value of 10. Floating point values can be expressed by using either:

- the normal notation with a dot in a sequence of numbers like, 1.23 (which represents $123*10^{-2}$), 2.783 (i.e., $2783*10^{-3}$) or -123.456789 (which represents $-123456789*10^{-6}$); or
- by two numbers separated by E where the first number specifies the mantissa and the second specifies the exponent, for example 12.3E4 (which represents $12.3*10^{4}$) or -12.3E-4 (which represents $-12.3*10^{-4}$).

e) **`boolean`:** a type consisting of two distinguished values.

Values of boolean type shall be denoted by **`true`** and **`false`**.

f) **`objid`**: a type whose distinguished values are the set of all object identifiers allocated in accordance with the rules of [7], [8], [9] and [10]. For example:

{itu-t(0) identified-organization(4) etsi(0)}

or alternatively {itu-t identified-organization etsi}

or alternatively { 0 4 0}

g) **`verdicttype`:** a type for use with test verdicts consisting of 4 distinguished values.

Values of **`verdicttype`** shall be denoted by **`pass`**, **`fail`**, **`inconc`**, **`none`** and **`error`.**

6.1.1 Basic string types and values

TTCN-3 supports the following basic string types:

NOTE: The general term string or string type in TTCN-3 refers to **`bitstring`**, **`hexstring`**, **`octetstring`**, **`charstring`** and **`universal charstring`**.

a) **`bitstring`:** a type whose distinguished values are the ordered sequences of zero, one, or more bits.

Values of type **`bitstring`** shall be denoted by an arbitrary number (possibly zero) of zeros and ones, preceded by a single quote (') and followed by the pair of characters 'B. For example:

'01101'B

b) **`hexstring`:** a type whose distinguished values are the ordered sequences of zero, one, or more hexadecimal digits, each corresponding to an ordered sequence of four bits.

Values of type **`hexstring`** shall be denoted by an arbitrary number (possibly zero) of the hexadecimal digits:

1 2 3 4 5 6 7 8 9 A B C D E F

preceded by a single quote (') and followed by the pair of characters 'H; each hexadecimal digit is used to denote the value of a semi-octet using a hexadecimal representation. For example:

'AB01D'H

c) **`octetstring`:** a type whose distinguished values are the ordered sequences of zero or a positive even number of hexadecimal digits (every pair of digits corresponding to an ordered sequence of eight bits).

Values of type **`octetstring`** shall be denoted by an arbitrary, but even, number (possibly zero) of the hexadecimal digits.

1 2 3 4 5 6 7 8 9 A B C D E F

preceded by a single quote (') and followed by the pair of characters 'O; each hexadecimal digit is used to denote the value of a semi-octet using a hexadecimal representation. For example:

'FF96'O

d) **`charstring`:** are types whose distinguished values are zero, one, or more characters from ISO/IEC 646 [5]. The character string type preceded by the keyword **`universal`** denotes types whose distinguished values are zero, one, or more characters from ISO/IEC 10646 [6].

Values of **`charstring`** type and **`universal charstring`** type shall be denoted by an arbitrary number (possibly zero) of characters from the relevant character set, preceded and followed by double quote (").

In cases where it is necessary to define strings that include the character double quote (") the character is represented by a pair of double quotes on the same line with no intervening space characters. For example, ""abcd"" represents the literal string "abcd".

6.1.2 Accessing individual string elements

Individual elements in a string type may be accessed using an array-like syntax. Only single elements of the string may be accessed.

Units of length of different string type elements are indicated in table 4.

Indexing shall begin with the value zero (0). For example:

```
// Given
MyBitString := '11110111'B;
// Then doing
MyBitString[4] := '1'B;
// Results in the bitstring '11111111'B
```

6.2 User-defined sub-types and values

User-defined types shall be denoted by the keyword **`type`**. With user-defined types it is possible to make sub-types (such as lists, ranges and length restrictions) on **`integer`** and the various string types.

6.2.1 Lists of values

TTCN-3 permits the specification of a list of distinguished values of any given type as listed in table 3. The values in the list shall be of the base type and shall be a true subset of the values defined by the base type. The subtype defined by this list restricts the allowed values of the subtype to those values in the list. For example:

```
type bitstring MyListOfBitStrings ('01'B, '10'B, '11'B);
```

6.2.2 Ranges

TTCN-3 permits the specification of a range of values of type **`integer, char`** and **`universal char`** (or derivations of these types). The subtype defined by this range comprises restricts the allowed values of the subtype to the values in the range including the lower boundary and the upper boundary. For example:

```
type integer MyIntegerRange (0 .. 255);
```

6.2.2.1 Infinite ranges

In order to specify an infinite integer range, the keyword **`infinity`** may be used instead of a value indicating that there is no lower or upper boundary. The upper boundary shall be greater than or equal to the lower boundary. For example:

```
type integer MyIntegerRange (-infinity .. -1); // All negative integer numbers
```

NOTE: The 'value' for infinity is implementation dependent. Use of this feature may lead to portability problems.

6.2.2.2 Mixing lists and ranges

For values of type **integer, char** and **universal char** (or derivations of these types) it is possible to mix lists and ranges. For example:

```
type integer MyIntegerRange (1, 2, 3, 10 .. 20, 99, 100);
```

6.2.3 String length restrictions

TTCN-3 permits the specification of length restrictions on string types. The length boundaries are of different complexity depending on the string type with which they are used. In all cases, these boundaries shall evaluate to non-negative **integer** values (or derived **integer** values). For example:

```
type bitstring MyByte length(8);            // Exactly length 8
type bitstring MyByte length(8 .. 8);       // Exactly length 8
type bitstring MyNibbleOrByte length(4 .. 8); // Minimum length 4, maximum length 8
```

Table 4 specifies the units of length for different string types.

Table 4: Units of length used in field length specifications

Type	Units of Length
bitstring	bits
hexstring	hexadecimal digits
octetstring	octets
character strings	characters

For the upper bound the keyword **infinity** may also be used to indicate that there is no upper limit for the length. The upper boundary shall be greater than or equal to the lower boundary.

6.3 Structured types and values

The **type** keyword is also used to specify structured types such as **record** types, **record of** types, **set** types, **set of** types, **enumerated** types and **union** types.

Values of these types may be given using an explicit assignment notation or a short-hand initializer. For example:

```
const MyRecordType MyRecordValue:=
{
   field1 := '11001'B,
   field2 := true,
   field3 := "A string"
}

// Or
const MyRecordType MyRecordValue:= {'11001'B, true, "A string"}
```

It is not allowed to mix the two value notations in the same (immediate) context. For example:

```
// This is disallowed
const MyRecordType MyRecordValue:= {MyIntegerValue, field2 := true, "A string"}
```

6.3.1 Record type and values

TTCN-3 supports ordered structured types known as **record**. The elements of a record type may be any of the base types or user-defined types such as other records, sets or arrays. The values of a record shall be compatible with the types of the record fields. The element identifiers are local to the record and shall be unique within the record. A constant that is of record type shall contain no variables (including module parameters) as field values, either directly or indirectly.

```
type record MyRecordType
{
   integer       field1,
   MyOtherStruct field2 optional,
   charstring    field3
```

```
}

type record MyOtherstructType
{
    bitstring  field1,
    boolean    field2
}
```

Records may be defined with no fields (i.e., as empty records). For example:

```
type record MyEmptyRecord {}
```

A **record** value is assigned on an individual element basis. For example:

```
var integer MyIntegerValue:= 1;

var MyRecordType MyRecordValue:=
{
    field1 := MyIntegerValue,
    field2 := MyOtherRecordValue,
    field3 := "A string"
}

const MyOtherRecordType MyOtherRecordValue:=
{
    field1 := '11001'B,
    field2 := true
}
```

Or using an initializer. For example:

```
MyRecordValue:= {MyIntegerValue, {'11001'B, true}, "A string"};
```

For optional fields it allowed to omit the value using the omit parameter symbol. For example:

```
MyRecordValue:= {MyIntegerValue, - , "A string"};

// Note that this is the same as writing, i.e., the value of field2 is undefined
MyRecordValue.field1 := MyIntegerValue;
MyRecordValue.field3 := "A string"
```

6.3.1.1 Referencing nested record fields

Elements of nested records are referenced by *RecordId.ElementId* pairs. For example:

```
MyVar1 := MyRecord1.MyElement1;
// If a record is nested then the reference may look like this
MyVar2 := MyRecord1.MyElement1.MyRecord2.MyElement2;
```

6.3.1.2 Optional elements in a record

Optional elements in a **record** shall be specified using the **optional** keyword. For example:

```
type record MyMessageType
{
    FieldType1 field1,
    FieldType2 field2 optional,
    :
    FieldTypeN fieldN
}
```

6.3.2 Set type and values

TTCN-3 supports unordered structured types known as **set**. Set types and values are similar to records except that the ordering of the set fields is not significant. For example:

```
type set MySetType
{
    integer        field1,
    charstring     field2
}
```

The initializer notation for setting values shall not be used for values of **set** types.

6.3.2.1 Optional elements in a set

Optional elements in a **set** shall be specified using the **optional** keyword.

6.3.3 Records and sets of single types

TTCN-3 supports the specification of records and sets whose elements are all of the same type. These are denoted using the keyword **of**. These records and sets do not have element identifiers and can be considered similar to an ordered array and an unordered array respectively.

The **length** keyword is used to restrict lengths of **record of** and **set of**. For example:

```
type record of length(10) integer MyRecordOfType; // is a record of a maximum of 10 integers
type set of boolean MySetOfType; // is an unlimited set of boolean values
type record of length(10) charstring StringArray length(10);
// is a record of a maximum of 10 strings each with a maximum length of 10 characters
```

The value notation for **record of** and **set of** is the same as the value notation for arrays (see clause 6.4).

6.3.4 Enumerated type and values

TTCN-3 supports enumerated types. Enumerated types are used to model types that take only a distinct named set of values. Operations on enumerated types shall only use the named identifiers and are restricted to assignment, equivalence and ordering operators.

Each named value may optionally have an associated integer value, which is defined after the name in parenthesis. These values are only used by the system to allow the use of relational operators. If no explicit integers are given the ordering is assumed to start with zero. For example:

```
type enumerated MyEnumType
{
    Monday, Tuesday, Wednesday, Thursday, Friday
}

// A valid instantiation of MyEnumType would be
var MyEnumType Today := Monday;
var MyEnumType Tomorrow := Tuesday;
// and the statement Today < Tomorrow is true
```

6.3.5 Unions

TTCN-3 supports **union** types. Union types are similar to records except that only one of the specified fields will ever be present in an actual union value. Union types are useful to model a structure which can take one of a finite number of known types. For example:

```
type union MyUnionType
{
    integer        number,
    charstring     string
}

// A valid instantiation of MyUnionType would be
var MyUnionType age;
age.number := 34;
```

The initializer notation for setting values shall not be used for values of **union** types.

The **optional** keyword shall not be used with union types.

6.4 Arrays

In common with many programming languages, arrays are not considered to be types in TTCN-3. Instead, they may be specified at the point of a variable declaration. For example:

```
var integer MyArray[3]; // Instantiates an integer array of 3 elements with the index 0 to 2
```

The values of array elements shall be compatible with the corresponding variable declaration. Values may be assigned individually or all at once. For example:

```
MyArray[0]:= 10;
MyArray[1]:= 20;
MyArray[2]:= 30;

// or using an initializer
MyArray:= {10, 20, 30};
```

Array indexes are expressions which shall evaluate to positive **integer** values, including the value zero. By default, indexing of TTCN-3 arrays shall start with the digit 0 (zero).

Array dimensions shall be specified using constant expressions which shall evaluate to a positive **integer** value. Array dimensions may also be specified using ranges. In such cases the lower and upper values of the range define the lower and upper index values. For example:

```
var integer MyArray[1 .. 5]; // Instantiates an integer array of 5 elements
                             // with the index 1 to 5
MyArray[1] := 10; // Lowest index
MyArray[5] := 50; // Highest index
```

Arrays of record of types allow the possibility to specify multi-dimensional arrays. For example:

```
// Given
type record MyRecordType
{
    integer       field1,
    MyOtherStruct  field2,
    charstring    field3
}
// An array of MyRecordType could be
var MyRecordType MyRecordArray[10];
// A reference to a particular element would look like this
MyRecordArray[1].field1 := 1;
```

6.5 Recursive types

Where applicable TTCN-3 type definitions may be recursive. The user, however, shall ensure that all type recursion is resolvable and that no infinite recursion occurs.

6.6 Type parameterization

Type parameterization allows dummy type identifiers which act as placeholders for any type. This means that a type can be left open by the TTCN-3 specifier as long as it is resolvable at compile-time.

NOTE: This is a generalization of the PDU meta-type concept of TTCN-2.

The actual type is only known when the type parameter is actually used. For example:

```
type record MyRecordType(MyMetaType)
{
    boolean    field1,
    MyMetaType field2 // MyMetaType is not of a particular type
}

var MyRecordType(integer) MyRecordValue :=
{
    field1 := true,
    field2 := 123  // MyMetaType is now of type integer
}
```

6.7 Type compatibility

TTCN-3 is not strongly typed but the language does require type compatability. Variables, constants, templates etc. have compatible types if they resolve to the same base type and, in the case of assignments, matching etc., no sub-typing (e.g., ranges, length restrictions) is violated.

For example:

```
// Given
type integer MyInteger(1 .. 10);
:
var integer x;
var MyInteger y;

// Then
x := 20; // is a valid assignment
y := 20; // is NOT a valid assignment because 20 is not in the range of y

y := 5; // is a valid assignment

x := y; // is a valid assignment, because the value of y is in the range of x
y := x; // is NOT valid assignment, because the value of x is not in the range of y

x := 5; // is a valid assignment
y := x; // is a valid assignment, because the value of x is now in the range of y
```

6.7.1 Type conversion

If it is necessary to convert values of one type to values of another type, where the types are not derived from the same base type, then either one of the predefined conversion functions defined in annex D or a user defined function shall be used. For example:

```
// To convert an integer value to a hexstring value use the predefined function int2hex
MyHstring := int2hex(123, 4);
```

7 Modules

The principal building blocks of TTCN-3 are modules. For example, a module may define a fully executable test suite or just a library. A module consists of a (optional) definitions part, and a (optional) module control part.

NOTE: The term test suite is synonymous with a complete TTCN-3 module containing test cases and a control part.

7.1 Naming of modules

Module names are of the form of a TTCN-3 identifier followed by an optional object identifier.

NOTE: The module identifier is the informal text name of the module.

7.2 Parameterization of modules

The **module** parameter list defines a set of values that are supplied by the test environment at run-time. During test execution these values shall be treated as constants. For example:

```
module MyParameterizedModule(integer TS_Par1, boolean TS_Par2, hexstring TS_Par3) { … }
```

NOTE: This provides functionality similar to TTCN-2 test suite parameters that provide PICS and PIXIT values to the test suite.

7.2.1 Default values for module parameters

For cases where actual module parameter values are not provided by the test environment at run-time, it is allowed to specify default values for module parameters. This shall be done by an assignment in the module parmeter list. For example:

```
module MyModuleDefaultParameter(integer Par1 := 1234, boolean Par2 := false) { … }
```

7.3 Module definitions part

The module definitions part specifies the top-level definitions of the module. These definitions may be used elsewhere in the module, including the control part. Those language elements which may be defined in a TTCN-3 module are listed in table 1. The module definitions may be imported by other modules.

EXAMPLE:

```
module MyModule
{   // This module contains definitions only
    :
    const integer MyConstant := 1;
    type record MyMessageType { … }
    :
    function TestStep(){ … }
    :
}
```

Declarations of dynamic language elements such as **`var`** or **`timer`** shall only be made in the control part, test cases or functions.

NOTE: TTCN-3 does not support the declaration of variables in the module definitions part, only in the control part. This means that global variables cannot be defined in TTCN-3.

7.3.1 Groups of definitions

In the module definitions part definitions can be collected in named groups. A group of declarations can be specified wherever a single declaration is allowed. Groups may be nested i.e., groups may contain other groups. This allows the test suite specifier to structure, among other things, collections of test data or functions describing test behaviour.

Grouping is done to aid readability and to add logical structure to the test suite if required. This means that all identifiers of the declarations in the set of groups (including any nested groups) at any given level of grouping shall be unique. In other words, groups and nested groups have no scoping *except* in the context of any attributes given to the group by an associated **`with`** statement. In such cases, a **`with`** statement on an outer group is overridden by a **`with`** statement on an inner group.

EXAMPLE:

```
// A collection of definitions
group MyGroup
{
    const integer MyConst:= 1;
    :
    type record MyMessageType { … }
}
// A group of test steps
group MyTestStepLibrary
{
    group MyGroup1
    {
        function MyTestStep11() { … }
        function MyTestStep12() { … }
        :
        function MyTestStep1n() { … }
    }
    group MyGroup2
    {
        function MyTestStep21() { … }
        function MyTestStep22() { … }
        :
        function MyTestStep2n() { … }
    }
```

```
}
```

7.4 Module control part

The module control part describes the execution order (possibly repetitious) of the actual test cases. A test case shall be defined in the module definitions part and called in the control part.

EXAMPLE:

```
module MyTestSuite
{   // This module contains definitions ...
    :
    const integer MyConstant := 1;
    type record MyMessageType { … }
    template MyMessageType MyMessage := { … }
    :
    function MyFunction1() { … }
    function MyFunction2() { … }
    :
    testcase MyTestcase1() runs on MyMTCType { … }
    testcase MyTestcase2() runs on MyMTCType { … }
    :
    // ... and a control part so it is executable
    control
    {
        var boolean MyVariable; // Local control variable
        :
        MyTestCase1(); // sequential execution of test cases
        MyTestCase2();
        :
    }
}
```

7.5 Importing from modules

It is possible to re-use definitions specified in different modules using the **`import`** statement. TTCN-3 has no explicit export construct thus, by default, all module definitions in the module definitions part may be imported. An **`import`** statement can be used anywhere in the module definitions part. It shall not be used in the control part.

If an imported definition has attributes (defined by means of a **`with`** statement) then the attributes shall also be imported.

NOTE: If the module has global attributes they are associated to definitions without these attributes.

EXAMPLE:

```
module MyModuleA
{   // This module contains definitions and imported definitions
    :
    const integer MyConstant := 1;
    import all from MyModuleB; // Scope of the imported definitions is global to MyModuleA
    type record MyMessageType { … }
    :
    function MyBehaviourC()
    {
        const integer MyConstant := 2;
        // import cannot be used here
        :
    }
    :
    control
    {   // import cannot be used here
        :
    }
}
```

7.5.1 Rules on using Import

On using import the following rules shall be applied:

a) only top-level definitions in the module may be explicitly imported. Definitions which occur at a lower scope (e.g., local constants defined in a function) shall not be imported;

b) by default, all definitions dependent on other definitions e.g., **record** types, are imported together with all the definitions on which they depend. If it is wished not to import these dependencies the **nonrecursive** directive may be used;

c) groups of definitions can also be imported. However, groups are only used for structuring purposes and do not have scope units. Therefore, it is allowed to import sub-groups i.e., a group which is defined within another group.

7.5.2 Importing single definitions

Single definitions may be imported. For example:

```
import type MyType from MyModuleC;
```

7.5.3 Importing all definitions of a module

The entire contents of a module definitions part (but not the actual module itself) may be imported, for example:

```
import all from MyModule;
```

7.5.4 Importing groups

Groups may be imported, for example:

```
import group MyGroup from MyModule;
```

Sub-groups i.e., groups which are defined within another group are also imported by this statement.

7.5.5 Importing definitions of the same kind

Blocks of the same kind of definition may be imported, for example:

```
import all template from MyModule;
```

7.5.6 Recursive import of complex definitions

By default, recursive definitions i.e., definitions that refer to other definitions, are implicitly imported by the **import** statement. Examples of recursive definitions are **record** types together with their component types or functions that call other functions, for example:

```
import type MyType from MyModuleC;
```

All definitions implicitly imported are visible at the top-level of scope and can be used subsequent to the import statement.

Note that local definitions within surrounding definitions e.g., local constant declarations within a function will never be visible.

EXAMPLE:

```
// Given
module MyModuleA
{   :
    function MyBehaviourB() { … }
    function MyBehaviourA()
    {   :
        MyBehaviourB();
        :
        const integer LocalConst:= 1000;
        :
    }
}

// Then
module MyModuleB
{   :
    import function MyBehaviourA from MyModuleA;
    :
}
// Will also import and make visible MyBehaviourB. Constant LocalConst will still
// be embedded in MyBehaviourA and will not be visible (outside of MyBehaviourA).
```

If definitions imported from one module depend on definitions in a further module then the definitions of the further module are imported too i.e., import shall implicitly import dependent definitions from the third-party module. This is due to the rule that an imported definition is handled in the same manner as a definition that is defined in the module itself.

If it is wished to inhibit recursive imports the **nonrecursive** directive shall be used. For example:

```
import type MyType from MyModuleC nonrecursive;
```

7.5.7 Handling name clashes on import

All TTCN-3 modules shall have their own name space in which all definitions shall be uniquely identified. Name clashes may occur due to import e.g., import from different modules, import of groups or import of recursive definitions. Name clashes shall be resolved by prefixing the imported definition (which causes the name clash) by the identifier of the module from which it is imported. The prefix and the identifier shall be separated by a dot (.).

In cases where there are no ambiguities the prefixing need not always be present when the imported definitions are used.

EXAMPLE:

```
module MyModuleA
{   :
    type bitstring MyTypeA;
    import type MyTypeA from SomeModuleC; // Where MyTypeA is of type character string
    import type MyTypeB from SomeModuleC; // Where MyTypeB is of type character string
    :
    control
    {   :
        var SomeModuleC.MyTypeA MyVar1 := "Test String"; // Prefix must be used
        var MyTypeA MyVar2 := '10110011'B; // This is the original MyTypeA
        :
        var MyTypeB MyVar3 := "Test String"; // Prefix need not be used …
        var SomeModuleC.MyTypeB MyVar3 := "Test String"; // … but it can be if wished
        :
    }
}
```

NOTE: Definitions with the same name defined in different modules are always assumed to be different, even if the actual definitions in the different modules are identical. For example, importing a type which is already defined locally, even with the same name, would lead to two different types being available in the module.

7.5.8 Handling multiple references to the same definition

The use of **`import`** on single definitions, groups of definitions, definitions of the same kind etc. may lead to situations where the same definition is referred to more than once. In such cases the definition shall be imported only once.

NOTE: The mechanisms to resolve such ambiguities e.g., overwriting and sending warnings to the user, are outside the scope of the present document and should be provided by TTCN-3 tools.

7.5.9 Import and module parameters

If an imported definition uses a module parameter then this parameter shall also be included in the module parameter list of the *importing* module.

7.5.10 Import definitions from non-TTCN modules

The **language** keyword is used to denote cases where type definitions are imported from non-TTCN modules. For example:

```
Import type MyASN1Type from MyASN1Module language "ASN.1:1997";
```

By default, the language is TTCN-3. For example:

```
import type MyType from MyModule;
// is the same as
import type MyType from MyModule language "TTCN-3";
```

8 Test configurations

TTCN-3 allows the (dynamic) specification of concurrent test configurations (or configuration for short). A configuration consists of a set of inter-connected test components with well-defined communication ports and an explicit test system interface which defines the borders of the test system.

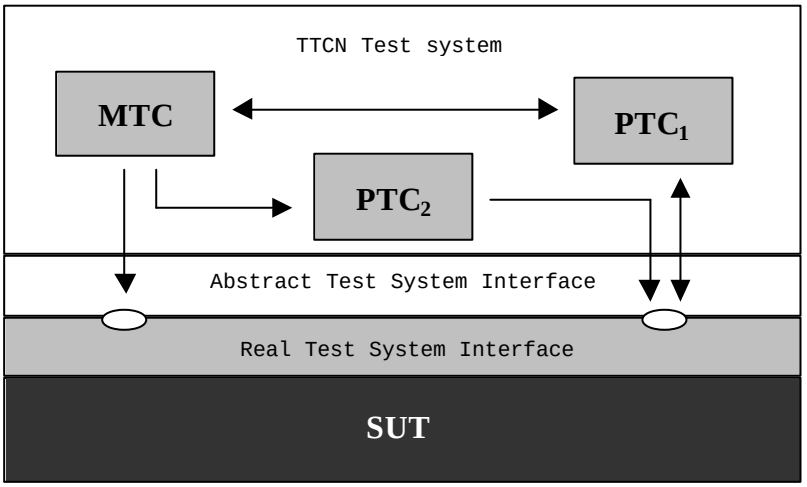


Figure 2: Conceptual view of a typical TTCN-3 test configuration

Within every configuration there shall be one (and only one) main test component (MTC). Test components that are not MTCs are called parallel test components or PTCs. The MTC shall be created automatically at the start of each test case execution. The behaviour defined in the body of the test case shall execute on this component. During execution of a test case other components can be created dynamically by the explicit use of the **`create`** operation.

Test case execution shall end when the MTC terminates. All other PTCs are treated equally i.e., there is no explicit hierarchical relationship among them and the termination of a single PTC terminates neither other components nor the MTC.

Communication is effected between the components within the test system and between the components and the test system interface via communication ports.

Test component types and port types, denoted by the keywords **component** and **port**, shall be defined in the module definitions part. The actual configuration of components and the connections between them is achieved by performing **create** and **connect** operations within the test case behaviour. The component ports are connected to the ports of the test system interface by means of the **map** operation (see clause 21.2).

8.1 Port communication model

Test components can be connected with other components and with the test system interface. There are no restrictions on the number of connections a component may have, but a component shall not connect to itself. One-to-many connections are allowed.

Test components are connected via their ports i.e., connections among components and between a component and the test system interface are port-oriented. Each port is modelled as an infinite FIFO queue which stores the incoming messages or procedure calls until they are processed by the component owning that port.

NOTE: While TTCN-3 ports are infinite in principle in a real test system they may overflow. This should be treated as a test case error (see clause 24.2.1).

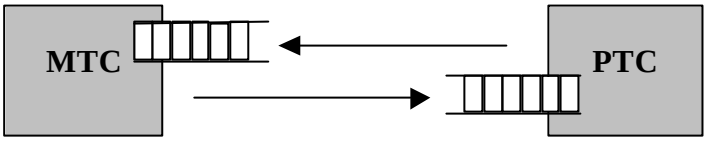


Figure 3: The TTCN-3 communication port model

8.2 Abstract test system interface

TTCN-3 is used to test implementations. The object being tested is known as the Implementation Under Test or IUT. The IUT may offer direct interfaces for testing or it may be part of system in which case the tested object is known as a System Under Test or SUT. In the minimal case the IUT and the SUT are equivalent. In the present document the term SUT is used in a general way to mean either SUT or IUT.

In a real test environment test cases need to communicate with the SUT. However, the specification of the real physical connection is outside the scope of TTCN-3. Instead, a well defined (but abstract) test system interface is associated with each test case. A test system interface definition is identical to a component definition i.e., it is a list of all possible communication ports through which the test case is connected to the SUT.

8.3 Defining communication port types

Ports facilitate communication between test components and between test components and the test system interface.

TTCN-3 supports message-based and procedure-based ports. Each port shall be defined as being either message-based or procedure-based or mixed. This shall be denoted by the keyword **message** or the keyword **procedure** within the associated port type definition.

Ports are directional. The directions are specified by the keywords **in** (for the in direction), **out** (for the out direction) and **inout** (for both directions). Each port type definition shall have one or more lists indicating the allowed collection of (message) types and/or procedures together with the allowed communication direction. For example:

```
// Message-based port which allows types MsgType1 and MsgType2 to be received at, MsgType3 to be
// sent via and any integer value to be send and received over the port
type port MyMessagePortType message
{
    in      MsgType1, MsgType2;
    out     MsgType3;
    inout   integer
}

// Procedure-based port which allows the remote call of the proceduress Proc1, Proc2 and Proc3.
// Note that Proc1, Proc2 and Proc3 are defined as signatures
type port MyProcedurePortType procedure
{
    out     Proc1, Proc2, Proc3
}
```

NOTE: The term message is used to mean both messages as defined by templates and actual values resulting from expressions. Thus, the list restricting what may be used on a message-based port is simply a list of type names.

Using the keyword **all** in one of the lists associated to a port type allows all types and/or all procedure signatures defined in the module to be passed over that communication port. For example:

```
// Message-based port which allows any value of all built-in types and user-defined types to be
// transferred in both directions over this port
type port MyAllMesssagesPortType message
{
    inout   all
}
```

8.3.1 Mixed ports

It is possible to define a port as allowing both kinds of communication. This is denoted by the keyword **mixed**. This means that the lists for mixed ports will also be mixed and include both, signatures and types. No separation is made in the definition.

```
// Mixed port, defining a message-based and a procedure-based port with the same name. The in,
// out and inout lists are also mixed: MsgType1, MsgType2, MsgType3 and integer refer to the
// message-based part of the mixed port and Proc1, Proc2, Proc3, Proc4 and Proc5 refer to the
// procedure-based port.
type port MyMixedPortType mixed
{
    in      MsgType1, MsgType2, Proc1, Proc2;
    out     MsgType3, Proc3, Proc4;
    inout   integer, Proc5;
}

// Mixed port, all types and all signatures defined in the module can be used at this port to
// communicate with either the SUT or other test components */

type port MyAllMixedPortType mixed
{
    inout   all
}
```

A mixed port in TTCN-3 is defined as a shorthand notation for two ports, i.e., a message-based port and a procedure-based port with the same name. At run-time the distinction between the two ports is made by the communication operations.

Operations used to control ports (see clause 21) i.e., **start**, **stop** and **clear** shall perform the operation on both queues (in arbitrary order) if called with an identifier of a mixed port.

8.4 Defining component types

The **component** type defines which ports are associated with a component. These definitions shall be made in the module definitions part. The port names in a component definition are local to that component i.e., another component may have ports with the same names. Ports of the same component shall all have unique names. However, this shall not be taken to mean that there is any connection between the components over these ports.

EXAMPLE:

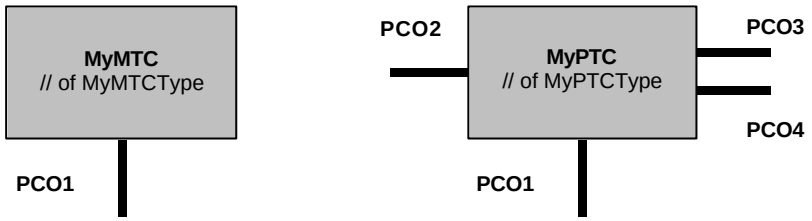


Figure 4: Typical components

```
type component MyMTCType
{
    port MyMessagePortTypePCO1
}

type component MyPTCType
{
    port MyMessagePortType    PCO1, PCO4;
    port MyProcedurePortType  PCO2;
    port MyAllMesssagesPortTypePCO3
}
```

8.4.1 Declaring local variables and timers in a component

It is possible to declare variables and timers local to a particular component. For example:

```
type component MyMTCType
{
    var integer MyLocalInteger;
    timer MyLocalTimer;
    port MyMessagePortTypePCO1
}
```

These declarations are visible to all functions that run on the component. This shall be explicitly stated using the **runs on** keyword (see clause 16).

Component variables and timers are associated with the component instance and follow the scope rules defined in clause 5.1. Each new instance of a component will thus have its own set of variables and timers as specified in the component definition (including any initial values, if stated).

8.4.2 Defining components with arrays of ports

It is possible to define arrays of ports in component type definitions (also see clause 21.9). For example:

```
type component My3pcoCompType
{
    port MyMessageInterfaceType PCO[3]
    // Defines a component type which has an array of 3 ports.
}
```

8.5 Addressing entities inside the SUT

An SUT may consist of several entities which have to be addressed individually. The address data type is a type for use with port operations to address SUT entities. The actual data representation of **address** is resolved either by an explicit type definition within the test suite or externally by the test system (i.e. the **address** type is left as an open type within the TTCN-3 specification). This allows abstract test cases to be specified independently of any real address mechanism specific to the SUT.

Explicit SUT addresses shall only be generated inside a TTCN-3 module if the type is defined inside the module. If the type is not defined inside the module explicit SUT addresses shall only be passed in as parameters or be received in message fields or as parameters of remote procedure calls.

In addition, the special value **null** is available to indicate an undefined address, e.g., for the initialization of variables of the address type.

EXAMPLE:

```
// Associates the type integer to the open type address
type integer address;
 :
// new address variable initialized with null
var address MySUTentity := null;
 :
// receiving an address value and assigning it to variable MySUTentity
PCO.receive(address:*) -> value MySUTentity;
 :
// usage of the received address for sending template MyResult
PCO.send(MyResult) to MySUTentity;
 :
// usage of the received address for receiving a confirmation template
PCO.receive(MyConfirmation) from MySUTentity;
```

8.6 Component references

Component references are unique references to the test components created during the execution of a test case. This unique component reference is generated by the test system at the time when a component is created, i.e., a component reference is the result of a **create** operation (see clause 21.1). In addition component references are returned by the predefined functions **system** (returns the component reference to identify the ports of the test system interface), **mtc** (returns the component reference of the MTC) and **self** (returns the component reference of the component in which **self** is called).

Component references are used in the configuration operations **connect**, **map** and **start** (see clause 21) to set-up test configurations and in the **from**, **to** and **sender** parts of communication operations for addressing purposes (see clause 22).

In addition, the special value **null** is available to indicate an undefined component reference, e.g., for the initialization of variables to handle component references.

The actual data representation of component references shall be resolved externally by the test system. This allows abstract test cases to be specified independently of any real TTCN-3 runtime environment, in other words TTCN-3 does not restrict the implementation of a test system with respect to the handling and identification of test components.

NOTE: A component reference includes component type information. This means, for example, that a variable for handling component references mu st use the corresponding component type name in its declaration.

EXAMPLE:

```
// A component type definition
type component MyCompType {
    port PortTypeOne PCO1;
    port PortTypeTwo PCO2
}

// Declaring two variable for the handling of references to components of type MyCompType
// and creating a component of this type
var MyCompType MyCompInst := MyCompType.create;
```

```
// Usage of component references in configuration operations
// allways referring to the component created above
connect(self:MyPCO1, MyCompInst:PCO1);
map(MyCompInst:PCO2, system:ExtPCO1);
MyCompInst.start(MyBehavior(self));// self is passed as a parameter to MyBehavior

// Usage of component references in from- and to- clauses
MyPCO1.receive from MyCompInst;
 :
MyPCO2.receive(integer:*) -> sender MyCompInst;
 :
MyPCO1.receive(MyTemplate) from MyCompInst;
 :
MPCO2.send(integer:5) to MyCompInst;

// The following example explains the case of a one-to-many connection at a Port PCO1
// where values of type M1 can be received from several components of the different types
// CompType1, CompType2 and CompType3 and where the sender has to be retrieved.
// In this case the following scheme may be used:
 :
var M1 MyMessage, MyResult;
var MyCompType1 MyInst1 := null;
var MyCompType2 MyInst2 := null;
var MyCompType3 MyInst3 := null;
 :
alt {
    [] PCO1.receive(M1:*) from MyCompType1 -> value MyMessage sender MyInst1 {}
    [] PCO1.receive(M1:*) from MyCompType2 -> value MyMessage sender MyInst2 {}
    [] PCO1.receive(M1:*) from MyCompType3 -> value MyMessage sender MyInst3 {}
}
 :
MyResult := MyMessageHandling(MyMessage);// some result is retrieved from a function
 :
if (MyInst1 != null) {PCO1.send(MyResult) to MyInst1};
if (MyInst2 != null) {PCO1.send(MyResult) to MyInst2};
if (MyInst3 != null) {PCO1.send(MyResult) to MyInst3};
 :
```

8.7 Defining the test system interface

A component type definition is used to define the test system interface because, conceptually, component type definitions and test system interface definitions have the same form (both are collections of ports defining possible connection points).

```
type component MyISDNTestSystemInterface
{
    port MyBchannelInterfaceType     B1;
    port MyBchannelInterfaceType     B2;
    port MyDchannelInterfaceType     D1
}
```

Generally, a component type reference defining the test system interface is associated with every test case. The ports of the test system interface are automatically instantiated together with the MTC when the test case execution starts i.e., when the test case is called from the control part of the module.

The operation returning the address of the test system interface is **`system`**. This can be used to address the ports of the test system. For example:

```
map(MyNewComponent:Port2, system:PCO1);
```

In the case where the MTC is the only component that is instantiated during test execution, a test system interface need not be associated to the test case. In this case, the component type definition associated with the MTC implicitly defines the corresponding test system interface.

9 Declaring constants

Constants can be declared and used in module headers, module control, test cases and functions. Constant definitions are denoted by the keyword **const**. The value of the constant shall be assigned at the point of declaration. For example:

```
const integer MyConst1 := 1;
const boolean MyConst2 := true, MyConst3 := false;
```

The assignment of the value to the constant may be done within the module or it may be done externally. The latter case is an external constant declaration denoted by the keyword **external**. External constants shall resolve to a value at compile-time. For example:

```
external const integer MyExternalConst;// external constant declaration
```

An external constant may have an arbitrary type but the type has to be known in the module i.e., a base type, defined in the module, or imported from some other module. The mapping of the type to the external representation of an external constant is again outside the scope of the present document. The mechanism of how the value of an external constant is passed into a module is outside the scope of the present document.

10 Declaring variables

Variables are denoted by the keyword **var**. Variables can be declared and used in module control, test cases and functions. They shall not be declared or used in a module header (i.e., global variables are not supported in TTCN-3). A variable declaration may have an optional initial value assigned to it. For example:

```
var integer MyVar1 := 1;
var boolean MyVar2 := true, MyVar3 := false;
```

Use of uninitialized variables at runtime shall cause a test case error.

11 Declaring timers

Timers can be declared and used in module control, test cases and functions. Timers shall not be declared or used in the module definitions part. A timer declaration may have an optional default duration value assigned to it. The timer shall be started with this value if no other value is specified. This value shall be of **float** type where the base unit is seconds. For example:

```
timer MyTimer1 := 5E-3;// declaration of the timer MyTimer1 with the default value of 5ms

timer MyTimer2;// declaration of MyTimer2 without a default timer value i.e., a value has
               // to be assigned when the timer is started
```

The timer operations **start**, **stop**, **read** and **timeout** may be used to control timers (see clause 23). For example:

```
// Uses of MyTimer2 might be
MyTimer2.start(10); // 10 seconds
MyTimer2.start(180); // 3 minutes
```

11.1 Timers as parameters

Timers can only be passed by reference to functions and named alternatives. Timers passed into a function or named alt are known inside the behaviour definition of the function or named alternative.

A timer passed in as reference parameter can be used like any other timer, i.e., it needs not to be declared. A started timer can also be passed into a function or named alternative. The timer continues its execution, i.e., it is not stopped implicitly. Thereby, possible timeout events can be handled inside the function or named alternative to which the timer is passed.

EXAMPLE:

```
// Function definition with a timer in the formal parameter list
function MyBehaviour (timer MyTimer)
{   :
    MyTimer.start;
    :
}
```

12 Declaring messages

One of the key elements of TTCN-3 is the ability to send and receive complex messages over the communication ports defined by the test configuration. These messages may be those explicitly concerned with testing the SUT or with the internal co-ordination and control messages specific to the relevant test configuration.

NOTE: In TTCN-2 these messages are the Abstract Service Primitives (ASPs), the Protocol Data Units (PDUs) and co-ordination messages. The core language of TTCN-3 is generic in the sense that it does not make any syntactic or semantic distinctions of this kind.

Complex messages may be defined as record types (see clause 6.3.1). For example:

```
type record MyMessageType
{
    FieldType1 field1,
    FieldType2 field2,
    :
    FieldTypeN fieldN
}
```

Messages can, of course, be sub-structured, for example:

```
// Information element type 1 (IEType1). Similar declarations for IEType2 to IETypeN
type record IEType1
{
    IEFieldType1   iefield1,
    IEFieldType2   iefield2,
    :
    IEFieldTypeN   iefieldN
}

// A message containing information elements
type record MyMessageType
{
    IEType1 field1,
    IEType2 field2,
    :
    IETypeN field3
}
```

12.1 Optional message fields

By default, all fields in a message shall be mandatory. Optional message fields shall be specified using the **optional** keyword. For example:

```
type record MyMessageType
{
    FieldType1 field1,
    FieldType2 field2 optional,
    :
    FieldTypeN fieldN
}
```

13 Declaring procedure signatures

Procedure signatures (or signatures for short) are needed for synchronous communication. A procedure may either be invoked in the SUT (i.e., the test system perfoms the call) or invoked in the test system (i.e., the SUT performs the call).

For both procedures called from the SUT and procedures called from the test system the complete procedure **signature** shall be defined in the TTCN-3 module.

Within the **signature** definition the parameter list may include parameter identifiers, parameter types and their direction i.e., **in**, **out**, or **inout**). Note that the direction of the parameters is as seen by the *called* party rather than the *calling* party. For example:

```
signature MyRemoteProc (in integer Par1, out float Par2, inout integer Par3) return integer;
// This defines the remote procedure MyRemoteProc. MyRemoteProc returns an integer value and
// has three parameters: one in parameter of type integer, one out parameter of type float
// and one inout parameter of type integer
```

A procedure **call** will result in the called party performing either a **reply** (the normal case) or raising an exception. The actions resulting from an accepted procedure call are defined by the receiving party (see clause 22).

13.1 Omitting actual parameters

It is allowed to omit actual parameters from a signature actual parameter list. This is indicated by representing the omitted actual parameter at its correct position by using the keyword **omit**. For example:

```
ParameterList(Par1, omit, Par3) // Par2 is omitted
```

NOTE: This is often necessary when using procedure signatures in synchronous communication.

13.2 Specifying exceptions

Exceptions are represented in TTCN-3 as values of a specific type, even templates and matching mechanisms can be used.

NOTE: The conversion of exceptions generated by the SUT into the corresponding type is tool and system specific and therefore beyond the scope of the present document.

The exceptions are defined in the form of an exception list included in the signature definition. This list defines all the possible different types associated with the set of possible exceptions (the meaning of exceptions themselves will usually only be distinguished by being represented by specific values of these types).

EXAMPLE:

```
signature MyRemoteProc (in integer Par1, out float Par2, inout integer Par3) return integer
        exception(ExceptionType1, ExceptionType2);

// A call of MyRemoteProc may raise exceptions of type ExceptionType1 or exceptions
// of ExceptionType2
```

14 Declaring templates

Templates are used to either transmit a set of distinct values or to test whether a set of received values matches the template specification.

Templates provide the following possibilities:

a) they are a way to organize and to re-use test data, including a simple form of inheritance;

b) they can be parameterized;

c) they allow matching mechanisms;

d) they can be used with either message-based or procedure-based communications.

Within a template values, ranges and matching attributes can be specified and then used in both message-based and procedure-based communications. Templates may be specified for any TTCN-3 type or procedure signature. The type-based templates are used for message-based communications and the signature templates are used in procedure-based communications.

14.1 Declaring message templates

Instances of messages with actual values may be specified using templates. A template can be thought of as being a set of instructions to build a message for sending or to match a received message.

Templates may be specified for any TTCN-3 type defined in table 3 except for the special types (**`port, component, address`**).

```
// When used in a receiving operation this template will match any integer value
template integer Mytemplate := *;
// This template will match only the integer values 1, 2 or 3
template integer Mytemplate := (1, 2, 3);
```

However, it is anticipated that the most common use will be with records, as shown by the examples in the following clauses.

14.1.1 Templates for sending messages

A template used in a **`send`** operation defines a complete set of field values comprising the message to be transmitted over a test port. At the time of the **`send`** operation, the template shall be fully defined i.e., all fields shall resolve to actual values and no matching mechanisms shall be used in the template fields, neither directly nor indirectly.

EXAMPLE:

```
// Given the message definition
type record MyMessageType
{
    integer    field1,
    charstring field2,
    boolean    field3
}

// a message template could be
template MyMessageType MyTemplate:=
{
    field1 := 1,
    field2 := "My string",
    field3 := true
}

// and a corresponding send operation could be
MyPCO.send(MyTemplate);
```

NOTE: Templates may also be used for exceptions if the corresponding type has been defined.

14.1.2 Templates for receiving messages

A template used in a **receive** operation defines a data template against which an incoming message is to be matched. Matching mechanisms, as defined in annex C, may be used in receive templates. No binding of the incoming values to the template shall occur.

EXAMPLE:

```
// Given the message definition
type record MyMessageType
{
    integer    field1,
    charstring field2,
    boolean    field3
}

// a message template might be
template MyMessageType MyTemplate:=
{
    field1 := 1,
    field2 := pattern "abc*xyz",
    field3 := true
}

// and a corresponding receive operation could be
MyPCO.receive(MyTemplate);
```

14.2 Declaring signature templates

Instances of procedure parameter lists with actual values may be specified using templates. Templates may be defined for any procedure by referencing the associated signature definition.

EXAMPLE:

```
// signature definition for a remote procedure
signature RemoteProc(in integer Par1, out integer Par2, inout integer Par3) return integer;

// example templates associted to defined procedure signature
template RemoteProc Template1:=
{
    Par1 :=1,
    Par2 :=2,
    Par3 :=3
}

template RemoteProc Template2:=
{
    Par1 := 1,
    Par2 := *,
    Par3 := 3
}

template RemoteProc Template3:=
{
    Par1 := 1,
    Par2 := *,
    Par3 := *
}
```

14.2.1 Templates for calling procedures

A template used in a **call** or **reply** operation defines a complete set of field values for all **in** and **inout** parameters. At the time of the **call** operation all **in** and **inout** parameters in the template shall resolve to actual values, no matching mechanisms shall be used in these fields, either directly or indirectly. Any template specification for **out** parameters is simply ignored, therefore it is allowed to specify matching mechanisms for these fields, or to omit them (see annex C).

EXAMPLE:

```
// Valid call since all in and inout parameters have a distinct value
MyPCO.call(RemoteProc:Template1);

// Valid call since all in and inout parameters have a distinct value
MyPCO.call(RemoteProc:Template2);

// Invalid call since Par3 parameters has a matching attribute not a value
MyPCO.call(RemoteProc:Template3);

// Templates never return values. In the case of Par2 and Par3 the values returned by the
// call must be retrived using an assignment clause at the end of the call statement
```

14.2.2 Templates for accepting procedure calls

A template used in a **getcall** operation defines a data template against which the incoming parameter fields are matched. Matching mechanisms, as defined in annex C, may be used in any templates used by this operation. No binding of incoming values to the template shall occur. Any **in** parameters shall be ignored in the matching process.

EXAMPLE:

```
// Valid getcall, it will match if Par2 == 2 and Par3 == 3
MyPCO.getcall(RemoteProc:Template1);

// Valid getcall, it will match if Par3 == 3 and Any value of Par2
MyPCO.getcall(RemoteProc:Template2);

// Valid getcall, it will match on Any value of Par3 and Par2
MyPCO.getcall(RemoteProc:Template3);
```

14.3 Template matching mechanisms

Generally, matching mechanisms will be used to replace values of single template fields or to replace even the entire contents of a template. Some of the mechanisms may be used in combination.

Matching mechanisms and wildcards may also be used in-line in received events only (i.e. **receive**, **getcall**, **getreply** and **catch** operations). They may appear in explicit values, for example:

```
MyPCO.receive(charstring:"abcxyz");
MyPCO.receive (integer:complement(1, 2, 3));
```

The type identifier is optional, for example:

```
MyPCO.receive("abcxyz");
```

However, the type of the in-line template shall be in the port list over which the template is received. In the case where there is an ambiguity (e.g., through sub-typing) then the type name shall be included in the receive statement.

Matching mechanisms are arranged in four groups:

a) specific values (i.e., an expression that evaluates to a specific value);

b) special symbols that can be used *instead* of values:

- **(…)**: a list of values;
- **`complement`** **(…)**: complement of a list of values;
- **`omit`**: value is omitted;
- **?**: wildcard for any value;
- *****: wildcard for any value or no value at all (i.e., an omitted value);
- **(**lower **`to`** upper**)**: a range of integer values between and including the lower- and upper bounds.

c) special symbols that can be used *inside* values:

- **?**: wildcard for any single element in a string, array, **`record of`** or **`set of`**;
- *****: wildcard for any number of consecutive elements in a string, array, **`record of`** or **`set of`**, or no element at all (i.e., an omitted element).

d) special symbols which describe *attributes* of values:

- **`length`**: restrictions for strings and arrays;
- **`ifpresent`**: for matching of optional field values (if not omitted).

The supported matching mechanisms and their associated symbols (if any) and the scope of their application are shown in table 5. The left-hand column of this table lists all the TTCN-3 and ASN.1 equivalent types as defined in the ITU-T Recommendation X.680 series [7], [8], [9] and [10] to which these matching mechanisms apply. A full description of each matching mechanism can be found in annex C.

Table 5: TTCN-3 Matching Mechanisms

Used with values of	Value	Instead of values								Attributes	
	SpecificValue	ValueList	ComplementedList	OmitValue	AnyValue (?)	AnyValueOrNone (*)	Range	AnyElement (?)	AnyElementsOrNone (*)	LengthRestriction	IfPresent
boolean	Yes	Yes	Yes	Yes	Yes	Yes					Yes
integer	Yes	Yes	Yes	Yes	Yes	Yes	Yes				Yes
char	Yes	Yes	Yes	Yes	Yes	Yes	Yes				Yes
universal char	Yes	Yes	Yes	Yes	Yes	Yes	Yes				Yes
float	Yes	Yes	Yes	Yes	Yes	Yes					Yes
bitstring	Yes	Yes	Yes	Yes	Yes	Yes		Yes	Yes	Yes	Yes
octetstring	Yes	Yes	Yes	Yes	Yes	Yes		Yes	Yes	Yes	Yes
hexstring	Yes	Yes	Yes	Yes	Yes	Yes		Yes	Yes	Yes	Yes
character strings	Yes	Yes	Yes	Yes	Yes	Yes		Yes	Yes	Yes	Yes
record	Yes	Yes	Yes	Yes	Yes	Yes					Yes
record of	Yes	Yes	Yes	Yes	Yes	Yes		Yes	Yes	Yes	Yes
array	Yes	Yes	Yes	Yes	Yes	Yes		Yes	Yes	Yes	Yes
set	Yes	Yes	Yes	Yes	Yes	Yes					Yes
set of	Yes	Yes	Yes	Yes	Yes	Yes		Yes	Yes	Yes	Yes
enumerated	Yes	Yes	Yes	Yes	Yes	Yes					Yes
union	Yes	Yes	Yes	Yes	Yes	Yes					Yes

14.4 Parameterization of templates

Templates for both sending and receiving operations can be parameterized. The formal parameters of a template can include templates, functions and the special matching symbols. The rules for formal and actual parameter lists shall be followed as defined in clause 5.3.

EXAMPLE:

```
// The template
template MyMessageType MyTemplate (integer MyFormalParam):=
{
    field1 := MyFormalParam,
    field2 := pattern "abc*xyz",
    field3 := true
}

// could be used as follows
pco1.send(MyTemplate(123));
```

14.4.1 Parameterization with matching attributes

To enable matching attributes to be passed as parameters the extra keyword **`template`** shall be added before the type field. This makes the parameter a template and in effect extends the allowed parameters for the associated type to include the appropriate set of matching attributes (see annex C) as well as the normal set of values. Template parameter fields shall not be called by reference.

EXAMPLE:

```
// The template
template MyMessageType MyTemplate (template integer MyFormalParam):=
{   field1 := MyFormalParam,
    field2 := pattern "abc*xyz",
    field3 := true
}

// could be used as follows
pco1.receive(MyTemplate(?));
// Or, if field1 has be defined as being optional
pco1.receive(MyTemplate(omit));
```

14.5 Passing templates as parameters

Only **`function`**, **`testcase`**, **`named alt`** and **`template`** definitions can have templates as formal parameters.

EXAMPLE:

```
function MyBehaviour(template MyMsgType MyFormalParameter)
runs on MyComponentType
{   :
    pco1.receive(MyFormalParameter);
    :
}
```

14.6 Modified templates

Normally a template will specify a set of base, or default, values or matching symbols for each and every field defined in the appropriate definition. In cases where small changes are needed to specify a new template it is possible to specify a modified template. A modified template specifies modifications to particular fields of the original template, either directly or indirectly.

The **`modifies`** keyword denotes the parent template from which the new, or modified template shall be derived. This parent template may be either the original template or a modified template.

The modifications occur in a linked fashion eventually tracing back to the original template. If a template field and its corresponding value or matching symbol is specified in the modified template, then the specified value or matching symbol replaces the one specified in the parent template. If a template field and its corresponding value or matching symbol is not specified in the modified template, then the value or matching symbol in the parent template shall be used.

A modified template shall not refer to itself, either directly or indirectly i.e., recursive derivation is not allowed.

EXAMPLE:

```
// Given
template MyRecordType MyTemplate1 :=
{
    field1 := 123,
    field2 := "A string",
    field3 := true
}

// then writing
template MyRecordType MyTemplate2 modifies MyTemplate1 :=
{
    field2 := "A modified string",
    field3 := omit // field3 must be specified as optional in the corresponding record type
}
```

```
// is the same as writing
template MyRecordType MyTemplate2 :=
{
    field1 := 123,
    field2 := "A modified string",
    field3 := omit
}
```

14.6.1 Parameterization of modified templates

If a base template is has a formal parameter list, the following rules apply to all modified templates derived from that base template, whether or not they are derived in one or several modification steps:

a) the derived template shall not omit parameters, however a derived template can have additional (appended) parameters if wished;

b) the formal parameter list shall follow the template name for every modified template;

c) parameterized templates in template fields shall not be modified or explicitly omitted in a modified template.

EXAMPLE:

```
// Given
template MyRecordType MyTemplate1(integer Mypar):=
{
    field1 := MyPar,
    field2 := "A string",
    field3 := true
}

// then a modification could be
    template MyRecordType MyTemplate2(integer MyPar)  modifies MyTemplate1 :=
{   // field1 is parameterized in Template1
    field2 := "A modified string",
    field3 := omit // field3 must be specified as optional in the corresponding record type
}
```

14.6.2 In-line modified templates

As well as creating explicitly named modified constraints TTCN-3 allows the definition of in-line modified constraints.

EXAMPLE:

```
// Given
template MyMessageType Setup :=
{   field1 := 75,
    field2 := "abc",
    field3 := true
}

// Could be used to define an in-line modified template of Setup
pco1.send (modifies Setup := {field1 76});
```

14.7 Changing template fields

All changes to template fields shall only be done via parameterization or by in-line derived templates at the time of performing a communication operation (e.g., **send**, **receive**, **call**, **getcall** etc.). The effects of these changes on the value of the template field do not persist in the template subsequent to the corresponding communication event.

The notation of the kind *MyTemplateId.Fieldid* shall not be used to set or retrieve values in templates in communication events. The "->" symbol shall be used for this purpose (see clause 22).

14.8 Match Operation

The **match** operation allows the value of a variable to be compared with a template. The operation returns a boolean value. If the type of the template and variable are not compatible the operation returns false. If the types are compatible the return value of the operation indicates whether the value of the variable conforms to the specified template.

```
template integer LessThan10 := (1..10);

testcase TC001()
runs on MyMTCType
{
    var integer RxValue;
    …
    PCO1.receive(integer:?) -> value RxValue;

    if( match( RxValue, LessThan10)) { … }
    …
}
```

14.9 Value of Operation

The **valueof** operation allows the value specified within a template to be assigned to the fields of a variable. The variable and template shall be type compatible (see 6.7) and each field of the template shall resolve to a single value.

```
type record ExampleType
{
 integer field1,
 boolean field2
}

template ExampleType SetupTemplate :=
{
    field1 := 1,
    field2 := true
}

…
    var ExampleType RxValue := valueof( SetupTemplate);
…
```

15 Operators

TTCN-3 supports a number of predefined operators that may be used in the terms of TTCN-3 expressions. The predefined operators fall into seven categories:

a) arithmetic operators;

b) string operators;

c) relational operators;

d) logical operators;

e) bitwise operators;

f) shift operators;

g) rotate operators.

These operators are listed in table 6.

Table 6: List of TTCN-3 operators

Category	Operator	Symbol or Keyword
Arithmetic operators	addition	**+**
	subtraction	**-**
	multiplication	*****
	division	**/**
	modulo	**mod**
	remainder	**rem**
String operators	concatenation	**&**
Relational operators	equal	**==**
	less than	**<**
	greater than	**>**
	not equal	**!=**
	greater than or equal	**>=**
	less than or equal	**<=**
Logical operators	logical not	**not**
	logical and	**and**
	logical or	**or**
	logical xor	**xor**
Bitwise operators	bitwise not	**not4b**
	bitwise and	**and4b**
	bitwise or	**or4b**
	bitwise xor	**xor4b**
Shift operators	shift left	**<<**
	shift right	**>>**
Rotate operators	rotate left	**<@**
	rotate right	**@>**

The precedence of these operators is shown in table 7. Within any row in this table, the listed operators have equal precedence. If more than one operator of equal precedence appears in an expression, the operations are evaluated from left to right. Parentheses may be used to group operands in expressions, in which case a parenthesized expression has the highest precedence for evaluation.

Table 7: Precedence of Operators

Priority	Operator type	Operator
highest		**(…)**
	Unary	**+, -, not, not4b**
	Binary	***, /, mod, rem**
		+, -
		<<, >>, <@, @>
		<, >, <=, >=
		==, !=
		and4b
		xor4b
		or4b
		and
		xor
		or
		&
Lowest		

15.1 Arithmetic operators

The arithmetic operators represent the operations of addition, subtraction, multiplication, division and modulo. Operands of these operators shall be of type **integer** (including derivations of **integer**) or **float** (including derivations of **float**), except for **mod** which shall be used with **integer** (including derivations of **integer**) types only.

With **integer** types the result type of arithmetic operations is **integer**. With float types the result type of arithmetic operations is **float**.

In the case where plus (+) or minus (-) is used as the unary operator the rules for operands apply as well. The result of using the minus operator is the negative value of the operand if it was positive and vice versa.

The result of performing the division operation (/) on two:

a) **integer** values gives the whole **integer** value resulting from dividing the first **integer** by the second (i.e., fractions are discarded);

b) **float** values gives the **float** value resulting from dividing the first **float** by the second (i.e., fractions are not discarded).

The operators **rem** and **mod** compute on operands of type **integer** and have a result of type **integer**. The operations `x rem y` and `x mod y` compute the rest that remains from an integer division of `x` by `y`. Therefore, they are only defined for non-zero operands `y`. For positive `x` and `y`, both `x rem y` and `x mod y` have the same result but for negative arguments they differ.

Formally, **mod** and **rem** are defined as follows:

```
x rem y  = x - y * (x/y)
x mod y  = x rem |y|        if  x >= 0
         = 0                if  x < 0   and x rem |y| = 0
         = y + x rem |y|    if  x < 0   and x rem |y| < 0
```

The following table illustrates the difference between the **mod** and **rem** operator:

Table 8: Effect of mod and rem operator

x	-3	-2	-1	0	1	2	3
x mod 3	0	1	2	0	1	2	0
x rem 3	0	-2	-1	0	1	2	0

15.2 String operators

The predefined relational operators perform concatenation of string types. The operands may be any string type values that are compatible. The operation is a simple concatenation from left to right. No form of arithmetic addition is implied. The result type is the compatible string type, for example:

```
'1111'B & '0000'B & '1111'B gives '111100001111'B
```

15.3 Relational operators

The predefined relational operators represent the relations of equality, less than, greater than, not equal to, greater than or equal to and less than or equal to. Operands of equality (==) and non-equality (!=) may be of an arbitrary type. All other relational operators shall have operands only of type **integer** (including derivatives of **integer**) or **float** (including derivations of **float**). In all cases the two operands shall be of compatible type. The result type of these operations is **boolean**.

15.4 Logical operators

The predefined **boolean** operators perform the operations of negation, logical **and**, logical **or** and logical **xor**. Their operands shall be of type **boolean**. The result type of the logical operators is **boolean**.

The logical **not** is the unary operator that returns the value **true** if its operand was of value **false** and returns the value **false** if the operand was of value **true**.

The logical **and** returns the value **true** if both its operands are **true**; otherwise it returns the value **false**.

The logical **or** returns the value **true** if at least one of its operands is **true**; it returns the value **false** only if both operands are **false**.

The logical **xor** returns the value **true** if one of its operands is **true**; it returns the value **false** if both operands are **false** or if both operands are **true**.

15.5 Bitwise operators

The predefined bitwise operators perform the operations of bitwise **not**, bitwise **and**, bitwise **or** and bitwise **xor**. These operators are known as **not4b**, **and4b**, **or4b** and **xor4b** respectively.

NOTE: To be read as "not for bit", "and for bit" etc.

Their operands shall be of `type` **bitstring, hexstring, octetstring**. The result type of the bitwise operators shall be the same as that of the operands.

The bitwise **not4b** unary operator inverts the individual bit values of its operand. For each bit in the operand a 1 bit is set to 0 and a 0 bit is set to 1. That is:

```
not4b  '1'B  gives  '0'B
not4b  '0'B  gives  '1'B
```

EXAMPLE:

```
not4b  '1010'B  gives  '0101'B
not4b  '1A5'H  gives 'E5A'H
not4b  '01A5'O gives 'FE5A'O
```

The bitwise **and4b** operator accepts two operands. For each corresponding bit position, the resulting value is a 1 if both bits are set to 1, otherwise the value for the resulting bit is 0. That is:

```
'1'B and4b '1'B gives  '1'B
'1'B and4b '0'B gives  '0'B
'0'B and4b '1'B gives  '0'B
'0'B and4b '0'B gives  '0'B
```

EXAMPLE:

```
'1001'B and4b '0101'B gives '0001'B
'B'H and4b '5'H  gives  '1'H
'FB'O and4b '15'O  gives  '11'O
```

The bitwise **or4b** operator accepts two operands. For each corresponding bit position, the resulting value is 0 if both bits are set to 0, otherwise the value for the resulting bit is 1. That is:

```
'1'B or4b '1'B gives  '1'B
'1'B or4b '0'B gives  '1'B
'0'B or4b '1'B gives  '1'B
'0'B or4b '0'B gives  '0'B
```

EXAMPLE:

```
'1001'B or4b '0101'B gives '1101'B
'9'H or4b '5'H gives  'D'H
'A9'O or4b 'F5'O gives  'FD'O
```

The bitwise **xor4b** operator accepts two operands. For each corresponding bit position, the resulting value is 0 if both bits are set to 0 or if both bits are set to 1, otherwise the value for the resulting bit is 0. That is:

```
'1'B xor4b '1'B gives  '0'B
'0'B xor4b '0'B gives  '0'B
'0'B xor4b '1'B gives  '1'B
'1'B xor4b '0'B gives  '1'B
```

EXAMPLE:

```
'1001'B xor4b '0101'B gives '1100'B
'9'H xor4b '5'H gives 'C'H
'39'O xor4b '15'O gives  '2C'O
```

15.6 Shift operators

The predefined shift operators perform the shift left (**<<**) and shift right (**>>**) operators. Their left-hand operand shall be of type **bitstring, hexstring, octetstring** or **integer.** Their right hand operand shall be of type **integer**. The result type of these operators shall be the same as that of the left operand.

The shift operators behave differently based upon the type of their left-hand operand. If the type of the left hand operand is:

a) **bitstring** or **integer** then the shift unit applied is 1 bit;

b) **hexstring** then the shift unit applied is 1 hexadecimal digit;

c) **octetstring** then the shift unit applied is 1 octet.

The shift left (**<<**) operator accepts two operands. It shifts the left-hand operand by the number of shift units to the left as specified by the right hand operand. Excess shift units (bits, hexadecimal digits or octets) are discarded. For each shift unit shifted to the left, a zero ('0'B, '0'H, or '00'O determined according to the type of the left-hand operand) is inserted from the right hand side of the left operand.

NOTE 1: If the left hand operand is of type **integer**, then for each bit shifted to the left, this is equivalent to multiplying the left hand operand by two.

NOTE 2: An error verdict shall be assigned if a system dependent overflow occurs when applying the shift left operation to the left hand operand.

EXAMPLE:

```
'111001'B  << 2 gives  '100100'B
'12345'H  << 2 gives  '34500'H
'1122334455'O  << (1+1)  gives  '3344550000'O
32  << 2  gives   128
-32 << 2 gives  -128
```

The shift right (**>>**) operator accepts two operands. It shifts the left-hand operand by the number of shift units to the right as specified by the right hand operand. Excess shift units (bits, hexadecimal digits or octets) are discarded. For each shift unit shifted to the right, a zero ('0'B, '0'H, or '00'O determined according to the type of the left-hand operand) is inserted from the left hand side of the left operand.

NOTE 1: If the left hand operand is of type **integer**, then for each bit shifted to the right, this is equivalent to doing integer division of the left hand operand by two (2).

NOTE 2: When the left operand is of type **integer** and its value is negative, when performing a right shift, the sign bit shall be propagated.

EXAMPLE:

```
'111001'B  >> 2 gives  '001110'B
'12345'H  >> 2 gives  '00123'H
'1122334455'O  >> (1+1)  gives  '0000112233'O
32  >> 2 gives  8
-32 >> 2 gives  -8
```

15.7 Rotate operators

The predefined rotate operators perform the rotate left (**<@)** and rotate right (**@>)** operators. Their left-hand operand shall be of `type` **bitstring, hexstring, octetstring, charstring or universal charstring.** Their right hand operand shall be of type **integer**. The result type of these operators shall be the same as that of the left operand.

The rotate operators behave differently based upon the type of their left-hand operand. If the type of the left hand operand is:

a) **bitstring** then the rotate unit applied is 1 bit;

b) **hexstring** then the rotate unit applied is 1 hexadecimal digit;

c) **octetstring** then the rotate unit applied is 1 octet;

d) **charstring** or **universal charstring** then the rotate unit applied is one character;

The rotate left **(<@)** operator accepts two operands. It rotates the left-hand operand by the number of shift units to the left as specified by the right hand operand. Excess shift units (bits, hexadecimal digits, octets, or characters) are re-inserted into the left-hand operand from its right-hand side.

EXAMPLE:

```
'101001'B  <@ 2 gives  '100110'B
'12345'H   <@ 2 gives  '34512'H
'1122334455'O  <@ (1+2)  gives  '4455112233'O
"abcdefg" <@ 3 gives "defgabc"
```

The rotate right **(@>)** operator accepts two operands. It rotates the left-hand operand by the number of shift units to the right as specified by the right hand operand. Excess shift units (bits, hexadecimal digits, octets, or characters) are re-inserted into the left-hand operand from its left-hand side.

EXAMPLE:

```
'100001'B  @> 2 gives  '0110001'B
'12345'H   @> 2 gives  '45123'H
'1122334455'O  @> (1+2)  gives  '3344551122'O
"abcdefg" @> 3 gives "efgabcd"
```

16 Functions

Functions are used in TTCN-3 to express test behaviour or to structure computation in a module, for example, to calculate a single value, to initialize a set of variables or to check some condition. Functions may return a value. This is denoted by the **return** keyword followed by a type identifier. If no **return** is specified then the function is void. An explicit keyword for void does not exist in TTCN-3. The keyword **return**, when used in the body of the function, causes the function to terminate and to return a value compatible with the return type. For example:

```
// Definition of MyFunction which has no parameters
function MyFunction() return integer
{

    return 7;  // return the integer value 7 when the function terminates
}
```

NOTE: The TTCN-3 functions replace Test Suite Operations and Test Suite Procedural Definitions in TTCN-2. Informal functions may be declared as external functions with explanatory comments or by using an empty formal function with comments.

A function may be defined within a module or be declared as being defined externally (i.e., **external**). For an external function only the function interface has to be provided in the TTCN-3 module. The realization of the external function is outside the scope of the present document. External functions are not allowed to contain port operations.

```
external function MyFunction4() return integer;// External function without parameters
                                               // which returns an integer value

external function InitTestDevices(); // An external function which only has an
                                     // effect outside the TTCN-3 module
```

In a module, the behaviour of a function can be defined by using the program statements and operations defined in clause 18. If a function includes port operations the associated component type shall be referenced using the **runs on** keywords in the function header to define the number, type and identifiers of the available ports. The one exception to this rule is if all ports used within the function are passed in as parameters.

If a function includes port operations either all ports used within the function shall be passed in as parameters or an associated component type shall be referenced using **runs on** in the function header to define the number, type and identifiers of the available ports. For example:

```
function MyFunction() runs on MyComponent return integer
{
    :
}
```

Instances of different component types may use the same function if they fulfil the following consistency rule:

"*Let C1 and C2 be two component types and FUNC be a function which refers to C1 in its* ***runs on*** *clause. An instance of component type C2 may use FUNC if the type definition C2 includes the entire type definition of C1. This means, C2 includes the same names to address ports of the same type as C1.*"

16.1 Parameterization of Functions

Functions may be parameterized. The rules for formal parameter lists shall be followed as defined in clause 5.3. For example:

```
function MyFunction2(inout integer MyPar1)
{
                          // MyFunction2 doesn't return a value
    MyPar1 := 10 * MyPar1;// but changes the value ofMyPar1 which
                          // is passed in by refefernce
}

function MyFunction3() runs on MyPTCType
{
                             // MyFunction3 doesn't return a value, but
    var integer MyVar := 5;  // does make use of the port operation
    PCO1.send(MyVar);          // send and therefore requires a runs on
                             // clause to resolve the port identifiers
}                            // by refernceing a component type
```

16.2 Invoking functions

A function is invoked by referring to its name and by the actual list of parameter. Functions that do not return values can be invoked directly. Functions that return values may be invoked inside expressions. The rules for actual parameter lists shall be followed as defined in clause 5.3.

```
MyVar := MyFunction4();// The value returned by MyFunction4 is assigned to MyVar.
                       // The types of the returned value and MyVar have to be the same

MyFunction2(MyVar2);  // MyFunction2 doesn't return a value and is called with the
                       // actual parameter MyVar2, which may be passed in by reference

MyVar3 := MyFunction6(4)+ MyFunction7(MyVar3);// Functions used in expressions
```

Special restrictions apply to functions bound to test components using the **start** operation. These restrictions are described in clause 21.5.

16.3 Predefined functions

TTCN-3 contains a number of predefined (built-in) functions that need not be declared before use.

Table 9: List of TTCN-3 predefined functions

Category	Function	Keyword
Conversion functions	Convert **integer** value to **char** value	**int2char**
	Convert **char** value to **int** value	**char2int**
	Convert **integer** value to **universal char** value	**int2unichar**
	Convert **universal char** value to **int** value	**unichar2int**
	Convert **bitstring** value to **integer** value	**bit2int**
	Convert **hexstring** value to **integer** value	**hex2int**
	Convert **octetstring** value to **Integer** value	**oct2int**
	Convert **charstring** value to **integer** value	**str2int**
	Convert **integer** value to **bitstring** value	**int2bit**
	Convert **integer** value to **hexstring** value	**int2hex**
	Convert **integer** value to **octetstring** value	**int2oct**
	Convert **integer** value to **charstring** value	**int2str**
Length/size functions	Return the length of a value of any string type	**lengthof**
	Return the number of elements in a **record, record of, template**, **set**, **set of** or **array**	**sizeof**
Presence/choice functions	Determine if an optional field in a **record, record of, template, set** or **set of** is present	**ispresent**
	Determine which choice has been made in a **union** type	**ischosen**

When a predefined function is invoked:

1) the number of the actual parameters shall be the same as the number of the formal parameters; and

2) each actual parameter shall evaluate to an element of its corresponding formal parameter's type; and

3) all variables appearing in the parameter list shall be bound.

The full description of predefined functions is given in annex D.

17 Test cases

Test cases are a special kind of function. Their execution in the module control part is related to the **execute** statement (see clause 26.1). The result of an executed test case is always a value of type **verdicttype**. Every test case shall contain one and only one MTC the type of which is referenced in the header of the test case definition. The behaviour defined in the test case body is the behaviour of the MTC.

When a test case is invoked the ports of the test system interface are instantiated, the MTC is created and the behaviour specified in the test case definition is started on the MTC. All these actions shall be performed implicitly i.e., without the explicit **create** and **start** operations.

To provide the information to allow these implicit operations to occur a test case definition has two parts:

a) interface part (mandatory): denoted by the keyword **runs on** which references the required component type for the MTC and makes the associated port names visible within the MTC behaviour; and

b) test system part (optional): denoted by the keyword **system** which references the component type which defines the required ports for the test system interface. The test system part shall only be omitted if, during test execution, only the MTC is instantiated. In this case, the MTC type defines the test system interface ports implicitly.

EXAMPLE:

```
testcase MyTestCaseOne()
runs on MyMtcType1       // defines the type of the MTC
system  MyTestSystemType  // makes the port names of the TSI visible to the MTC
{
    :   // The behaviour defined here executes on the mtc when the test case invoked
}

// or, a test case where only the MTC is instantiated
testcase MyTestCaseTwo() runs on MyMtcType2
{
    :   // The behaviour defined here executes on the mtc when the test case invoked
}
```

18 Program statements and operations

The fundamental program elements of the control part of TTCN-3 modules and functions are basic program statements such as expressions, assignments, loop constructs etc., behavioural statements such as sequential behaviour, alternative behaviour, interleaving, defaults etc., and operations such as **`send`**, **`receive`**, **`create`**, etc.

Statements can be either single statements (which do not include other program statements) or compound statements (which may include other statements).

Statement blocks are a mechanism to group statements. Statement blocks may be used in different scope units i.e., module control, functions and test behaviours. The kind of statements that may be used in a block will depend on the scope unit in which the block is used. For example, a statement block appearing in a function shall only use those program statements which may be used in functions.

General scoping rules are described in clause 5.4.

A statement block is syntactically equivalent to a single statement, thus, wherever a statement is allowed in a function a block may appear. This implies that blocks may be nested. Declarations, if any, shall be made at the beginning of the block. These declarations are only visible inside the block and to nested sub-blocks.

The statements in the block shall be executed in the order of their appearance. The specification of an empty statement block i.e., `{}`, is allowed. An empty statement block implies that no actions are taken.

Table 10: Overview of TTCN-3 statements and operations

Statement	Associated keyword or symbol	Can be used in module control	Can be used in functions, test cases and named alts
Basic program statements			
Expressions	**(…)**	Yes	Yes
Assignments	**:=**	Yes	Yes
Logging	**log**	Yes	Yes
Label and Goto	**label / goto**	Yes	Yes
If-else	**if (…) {…} else {…}**	Yes	Yes
For loop	**for (…) {…}**	Yes	Yes
While loop	**while (…) {…}**	Yes	Yes
Do while loop	**do {…} while (…)**	Yes	Yes
Stop execution	**stop**	Yes	Yes
Behavioural program statements			
Alternative behaviour	**alt {…}**	Yes (see note 1)	Yes
Named alternative	**named alt {…}**	Yes (see note 1)	Yes
Interleaved behaviour	**interleave {…}**	Yes (see note 1)	Yes
Activate a default	**activate**	Yes (see note 1)	Yes
Deactivate a default	**deactivate**	Yes (see note 1)	Yes
Returning control	**return**		Yes
Configuration operations			
Create parallel test component	**create**		Yes
Connect component to component	**connect**		Yes
Disconnect two components	**disconnect**		Yes
Map port to test interface	**map**		Yes
Unmap port from test system interface	**unmap**		Yes
Get MTC address	**mtc**		Yes
Get test system interface address	**system**		Yes
Get own address	**self**		Yes
Start execution of test component	**start**		Yes
Stop execution of test component	**stop**		Yes
Check termination of a PTC	**running**		Yes
Wait for termination of a PTC	**done**		Yes
Communication operations			
Send message	**send**		Yes
Invoke procedure call	**call**		Yes
Reply to procedure call from remote entity	**reply**		Yes
Raise exception (to an accepted call)	**raise**		Yes
Receive message	**receive**		Yes
Trigger on message	**trigger**		Yes
Accept procedure call from remote entity	**getcall**		Yes
Handle response from a previous call	**getreply**		Yes
Catch exception (from called entity)	**catch**		Yes
Check (current) message/call received	**check**		Yes
Clear port	**clear**		Yes
Clear and give access to port	**start**		Yes
Stop access (receiving & sending) at port	**stop**		Yes
Timer operations			
Start timer	**start**	Yes	Yes
Stop timer	**stop**	Yes	Yes
Read elapsed time	**read**	Yes	Yes
Check if timer running	**running**	Yes	Yes
Timeout event	**timeout**	Yes	Yes
Verdict operations			
Set local verdict	**verdict.set**		Yes
Get local verdict	**verdict.get**		Yes
SUT operations			
Remote action to be done by the SUT	**sut.action**		Yes
Execution of test cases			
Execute test case	**execute**	Yes	Yes (see note 2)
NOTE 1: Can be used in control with timer operations only.			
NOTE 2: Can only be used in functions and named alternatives that are used in module control.			

19 Basic program statements

Basic program statements are expressions, assignments, operations, loop constructs etc. All basic program statements can be used in the control part of a module and in TTCN-3 functions.

Table 11: Overview of TTCN-3 basic program statements

Basic program statements	
Statement	**Associated keyword or symbol**
Expressions	**(…)**
Assignments	**:=**
Logging	**log**
Label and Goto	**label / goto**
If-else	**if (…) { … } else { … }**
For loop	**for (…) { … }**
While loop	**while (…) { … }**
Do while loop	**do { … } while (…)**
Stop execution	**stop**

19.1 Expressions

TTCN-3 allows the specification of expressions using the operators defined in clause 15. Expressions are built from other (simple) expressions. Expressions may contain functions. The result of an expression shall be the value of a specific type and the operators used shall be compatible with the type of the operands. For example:

```
(x + y - increment(z))*3;
```

19.1.1 Boolean expressions

A **boolean** expression shall only contain **boolean** values and/or **boolean** operators and/or relational operators and shall evaluate to a **boolean** value of either **true** or **false**. For example:

```
((A and B) or (not C) or (j<10));
```

19.2 Assignments

Values may be assigned to variables. This is indicated by the symbol ":=". During execution of an assignment the right-hand side of the assignment shall evaluate to an element of the same type of the left-hand side. The effect of an assignment is to bind the variable (which may also be the element of a **record** or **set** etc.) to the value of the expression. The expression shall contain no unbound variables. All assignments occur in the order in which they appear, that is left to right processing. For example:

```
MyVariable := (x + y - increment(z))*3;
```

19.3 The Log statement

The **log** statement provides the means to write a character string to some logging device associated with test control or the test component in which the statement is used. For example:

```
log("Line 248 in PTC_A");
// The string "Line 248 in PTC_A" is written to some log device of the test system
```

NOTE: It is outside the scope of the present document to define complex logging and trace capabilities which may be tool dependent.

19.4 The Label statement

The **label** statement allows the specification of labels in test cases, functions, named alternatives and the control part of a module. A **label** statement can be used freely like other TTCN-3 behavioural program statements according to

the syntax rules defined in annex A. It can be used before or after a TTCN-3 statement but, for example, not as first statement of an alternative in an **alt** or **interleave** statement (see clause 20.2.7).

19.5 The Goto statement

The **goto** statement can be used in functions, test cases, named alternatives and the control part of a TTCN module. The **goto** statement performs a jump to a **label** or to the beginning of an **alt** statement in order to force repeated behaviour (see clause 20.2.8).

19.6 The If-else statement

The **if-else** statement, also known as the conditional statement, is used to denote branching in the control flow due to **boolean** expressions. Schematically the conditional looks as follows:

```
if (expression1)
    statementblock1
else
    statementblock2
```

Where $\texttt{statementblock}_x$ refers to a block of statements.

EXAMPLE:

```
if (date == "1.1.2000") return { fail };

if (MyVar < 10) {
        MyVar := MyVar * 10;
        log ("MyVar < 10");
}
else {
    MyVar := MyVar/5;
}
```

A more complex scheme could be:

```
if (expression1)
    statementblock1
else if (expression2)
    statementblock2
:
else if (expressionn)
    statementblockn
else
    statementblockn+1
```

In such cases readability heavily depends on the formatting but formatting shall have no syntactic or semantic meaning.

19.7 The For statement

The **for** statement defines a counter loop. The value of the index variable is increased, decreased or manipulated in such a manner that after a certain number of execution loops a termination criteria is reached.

The **for** statement contains two assignments and a **boolean** expression. The first assignment is necessary to initialize the index (or counter) variable of the loop. The **boolean** expression terminates the loop and the second assignment is used to manipulate the index variable. For example:

```
for (j:=1; j<=10; j:= j+1) { … }
```

The termination criterion of the loop shall be expressed by the **boolean** expression. It is checked at the beginning of each new loop iteration. If it evaluates to **true**, the execution continues with the statement which immediately follows the **for** loop.

The index variable of a **for** loop can be declared before being used in the for statement or can be declared and initialised in the **for** statement header. If the index variable is declared and initialised in the **for** statement header, the scope of the index variable is limited to the loop body, i.e., it is only visible inside the loop body. For example:

```
    var integer j;                        // Declaration of integer variable j
    for (j:=1; j<=10; j:= j+1) { … }      // Usage of variable j as index variable of the for loop


    for (var float i:=1.0; i<7.9; i:= i*1.35) { … }    // Index variable i is declared and
initialized
                                                       // in the for loop header. Variable i only is
                                                       // visible in the loop body.
```

19.8 The While statement

A **while** loop is executed as long as the loop condition holds. The loop condition shall be checked at the beginning of each new loop iteration. If the loop condition does not hold, then the loop is exited and execution shall continue with the statement, which immediately follows the **while** loop. For example:

```
    while (j<10){ … }
```

19.9 The Do-while statement

The **do-while** loop is identical to a **while** loop with the exception that the loop condition shall be checked at the *end* of each loop iteration. This means when using a **do-while** loop the behaviour is executed at least once before the loop condition is evaluated for the first time. For example:

```
    do { … } while (j<10);
```

19.10 The Stop execution statement

The **stop** statement terminates execution in different ways depending on the context in which it is used. When used in the control part of a module it terminates execution of the entire module. When used in a function that is executing behaviour it terminates the relevant test component.

20 Behavioural program statements

Behavioural program statements may be used in test cases, functions and module control, except for the return statement which shall only be used in test cases and functions. Behavioural program statements specify the dynamic behaviour of the test components over the communication ports. Test behaviour can be expressed, sequentially, as a set of alternatives or combinations of both. An interleaving operator allows the specification of interleaved sequences or alternatives.

Table 12: Overview of TTCN-3 behavioural program statements

Behavioural program statements	
Statement	**Associated keyword or symbol**
Alternative behaviour	**alt { … }**
Named alternative	**named alt { … }**
Interleaved behaviour	**interleave { … }**
Activate a default	**activate**
Deactivate a default	**deactivate**
Returning control	**return**

20.1 Sequential behaviour

The simplest form of behaviour is a set of statements that are executed sequentially, as illustrated below:

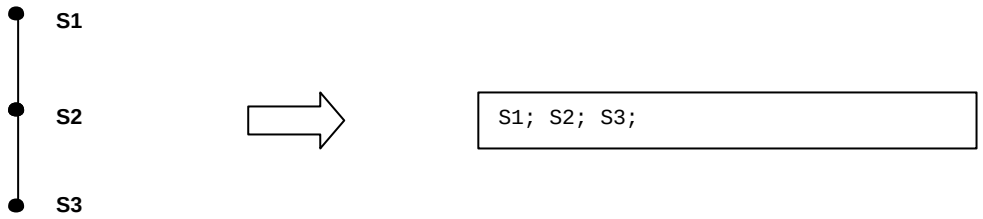


Figure 5: Illustration of sequential behaviour

The individual statements in the sequence shall be separated by the delimiter ";". For example:

```
MyPort.send(Mymessage); MyTimer.start; log("Done!");
```

20.2 Alternative behaviour

A more complex form of behaviour is where sequences of statements are expressed as sets of possible alternatives to form a tree of execution paths, as illustrated below:

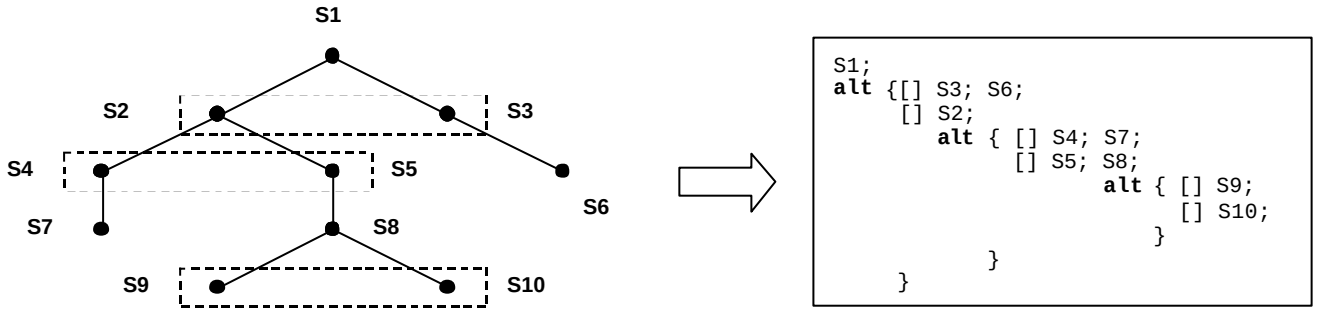


Figure 6: Illustration of alternative behaviour

The **alt** statement denotes branching of test behaviour due to the reception and handling of communication and/or timer events and/or the termination of parallel test components, i.e., it is related to the use of the TTCN-3 operations **receive**, **trigger**, **getcall**, **getreply**, **catch**, **check**, **timeout** and **done**. The **alt** statement denotes a set of possible events that are to be matched against a particular snapshot (see clause 20.2.1).

NOTE: The **alt** statement corresponds to the alternatives at the same level of indentation in TTCN-2. However, there are three significant differences:

a) **boolean** expressions to disable alternatives can only be made in an alternative statement;
b) it is not possible to examine the port queue by using the **boolean** expression and then to disable an alternative;
c) It is not possible to call a function as an alternative in the **alt** statement, except in the case where an else guard (i.e., [**else**]) is the last choice in the alternative (see clause 20.2.3).

EXAMPLE:

```
// Use of nested alternative statements
:
alt
{
[]  L1.receive(DL_REL_CO:*)          // UA or DM received; layer 2 released
    {   verdict.set(pass);
        TAC.stop;
        TNOAC.start;
        alt {
        []  L1.receive(DL_EST_IN)     // SABME received
            {   TNOAC.stop;
                verdict.set(pass);
            }
        []  TNOAC.timeout
            {   L1.send(DEL_EST_RQ:*);
                TAC.start;
                alt {
                []  L1.receive(DL_EST_CO:*) // UA received; data link established
                    {   TAC.stop;
                        verdict.set(pass)
                    }
                []  TAC.timeout       // no response
                    {verdict.set(inconc)}
                []  L1.receive            // like OTHERWISE in TTCN-2
                    {verdict.set(inconc)}
                }
            }
        []  L1.receive                    // like OTHERWISE in TTCN-2
            {verdict.set(inconc)}
        }
    }
[]  TAC.timeout                       // no response
    {verdict.set(inconc)}
[]  L1.receive                        // like OTHERWISE in TTCN-2
    {verdict.set(inconc)}
}
:


// Use of alternative with Boolean expressions (or guard)
:
alt {
[]      L1.receive(MyMessage1)
        {verdict.set(fail)}
[x>1]   L2.receive(MyMessage2)    // Boolean guard/expression
        {verdict.set(pass)}
[x<=1]  L2.receive(MyMessage3)    // Boolean guard/expression
        {verdict.set(inconc)}
}
:

// Use of done in alternatives
:
alt {
    []  MyPTC.done {
            verdict.set(pass)
        }

    []  any port.receive {
            goto alt
        }
}
:
```

20.2.1 Execution of alternative behaviour

The alternative statements in an **alt** statement are processed in their order of appearance. TTCN-3 operational semantics (see annex B) assume that the status of any of the events cannot change during the process of trying to match one alternative in a set of alternatives. This implies that snapshot semantics are used for received events and timeouts i.e., each time around a set of alternatives a snapshot is taken of which events have been received and which timeouts have fired. Only those identified in the snapshot can match on the next cycle through the alternatives.

NOTE 1: These semantics are exactly the same as for TTCN-2.

NOTE 2: Synchronous events (e.g., **call**) block the loop until a call is completed.

20.2.2 Selecting/deselecting an alternative

If necessary, it is possible to enable/disable an alternative by means of a **boolean** expression placed between the '[ ]' brackets of the alternative. For example:

```
[MyVar==3] PCO.receive(MyMessage) {}
```

The open and close square brackets '[' ']' shall be present at the start of each alternative, even if they are empty. This not only aids readability but also is necessary to syntactically distinguish one alternative from another.

20.2.3 Else branch in alternatives

If necessary, it is possible to define one branch in the alternative statement which is always taken if no other previously defined alternative can be taken. If an **else** branch is defined all subsequently defined alternatives are redundant i.e., they can never be reached. For example:

```
:
alt {
[]      L1.receive(MyMessage1)
        {   verdict.set(fail);
            MyComponent.stop
        }
[x>1]   L2.receive(MyMessage2)    // Boolean guard/expression
        {   verdict.set(pass);
            :
        }
[x<=1]  L2.receive(MyMessage3)    // Boolean guard/expression
        {   verdict.set(inconc);
            :
        }
[else]  {   MyErrorHandling();        // else branch
            verdict.set(fail);
            MyComponent.stop;
        }
}
:
```

It should be noted that defaults are always appended to the end of all alternatives. If an **else** branch is defined, an activated **default** will never be entered.

NOTE: It is also possible to use **else** in named alternatives.

20.2.4 Declaring named alternatives

Alternatives which are used in a number of places can be defined in a named alternative denoted by the keyword pair **named alt**. Named alternatives shall be defined globally in the module definitions. When invoked a **named alt** is identical to the behaviour **alt** construct except that it has an identifier and allows parameterization.

A **named alt** when referenced has the same effect as a macro substitution. A **named alt** can be referenced at any place in a behaviour definition where it is valid to include a normal **alt** construct.

EXAMPLE:

```
// Definition of the named alternatives macro
named alt HandlePCO2()
{
    []  PCO2.receive(DL_EST_IN)
        {PCO2.send(DL_EST_CO)}

    []  PCO2.receive(DL_EST_CO) {}
        // do nothing
}

// Using a named alt in-line
testcase TC001() runs on MyPTCtype
{
  :
```

```
  HandlePCO2();       // Call named alt
:
}

// Which expands to
testcase TC001() runs on MyPTCtype
{
    :
    alt {
        []  PCO2.receive(DL_EST_IN)
            {PCO2.send(DL_EST_CO)}
        []  PCO2.receive(DL_EST_CO) {}
            // do nothing
    }
    :
}
```

20.2.5 Expanding alternatives with named alternatives

In addition to direct in-line referencing it is also possible to explicitly expand the alternatives specified in the **named alt** construct using the **expand** statement. The **expand** statement can be placed at any position within an **alt** statement and will insert the associated guards from the **named alt** at that position.

EXAMPLE:

```
// Using a named alt by expanding
testcase TC002()runs on MyPTCtype
{
  :

    alt {
        []  PCO1.receive(DL_EST_IN)
            {PCO1.send(DL_EST_CO)}
        []  PCO1.receive(DL_EST_CO) {}
            // do nothing
        [expand] HandlePCO2() // Expand named alt alternatives to specified alt statement
    }
}

// Which expands to
testcase TC002()runs on MyPTCtype
{
    :
    alt {
        []  PCO1.receive(DL_EST_IN)
            {PCO1.send(DL_EST_CO)}
        []  PCO1.receive(DL_EST_CO) {}
            // do nothing
        []  PCO2.receive(DL_EST_IN)
            {PCO2.send(DL_EST_CO)}
        []  PCO2.receive(DL_EST_CO) {}
            // do nothing
    }
}
```

20.2.6 Parameterization of named alternatives

Named alternatives can be parameterized with types, values, functions and templates. Since named alternatives are not a scope unit, the defined formal parameters are simply substituted by the given actual parameters when the macro expansion is performed.

EXAMPLE:

```
named alt HandleAnyPCO(MyPortT PCO)
{
    []  PCO.receive(DL_EST_IN)
        {PCO.send(DL_EST_CO)}
    []  PCO.receive(DL_EST_CO) {}
        // do nothing
}

testcase TC001() runs on MyPTCtype
{
```

```
    HandleAnyPCO(PCO2);
    :
    alt {
        [expand] HandleAnyPCO(PCO1);
        [expand] HandleAnyPCO(PCO2);
    }
}
```

20.2.7 The Label statement in behaviour

The **label** statement allows the specification of labels in test cases, functions, named alternatives and the control part of a module. It can be used before or after any TTCN-3 statement but shall not be the first statement of an alternative in an **alt** or **interleave** statement.

EXAMPLE:

```
label MyLabel;
// Defines the label MyLabel

// The labels L1, L2 and L3 are defined in the following TTCN-3 code fragment
 :
label L1;                           // Definition of label L1
alt{
[]  PCO1.receive(MySig1)
    {   label L2;                   // Definition of label L2
        PCO1.send(MySig2);
        PCO1.receive(MySig3)
    }
[]  PCO2.receive(MySig4)
    {   PCO2.send(MySig5);
        PCO2.send(MySig6);
        label L3;                   // Definition of label L3
        PCO2.receive(MySig7);
        goto L1;                    // Jump to label L1
    }
}
:
```

20.2.8 The Goto statement in behaviour

The **goto** statement can be used in functions, test cases, named alternatives and the control part of a TTCN module. The **goto** statement performs a jump to a **label** or to the beginning of an **alt** statement in order to force repeated behaviour.

The re-evaluation of an **alt** statement can be achieved by either:

a) using **goto** <*LabelId*> where the relevant label statement should be placed immediately before the **alt** keyword of the actual alternative that is to be jumped to; or

b) by using **goto alt** within the **alt** statement which should be re-evaluated. In this case the keyword **alt** can be seen as an implicit label for the **alt** statement within which the **goto** is used.

20.2.8.1 Restricting the use of Goto

The **goto** statement provides the possibility to jump freely, i.e., forwards and backwards, within a sequence of statements, to jump out of a single compound statement (e.g., a **while** loop) and to jump over several levels out of nested compound statements (e.g., nested alternatives). However, the use of the **goto** statement shall be restricted by the following rules:

a) It is not allowed to jump out of or into functions, test cases, named alternatives and the control part of a TTCN module.

b) It is not allowed to jump into a sequence of statements defined in a compound statement (i.e., **alt** statement, **while** loop, **for** loop, **if-else** statement, **do-while** loop and the **interleave** statement).

c) As an exception to rule a) for named alternatives, it is allowed to use **goto alt** inside a named alternative in order to force the re-evaluation of an **alt** statement within which the named alternative may be expanded.

NOTE: This rule provides the possibility to jump out of a named alternative in a restricted manner to provide the functionality to describe defaults.

d) It is not allowed to use the **goto** statement within an **interleave** statement.

EXAMPLE:

```
// The following TTCN-3 code fragment includes
 :
label L1;
MyVar := 2 * MyVar;
if (MyVar < 2000) { goto L1; }       // ... a jump backward to L1 and
MyVar2 := Myfunction(MyVar);
if (MyVar2 > MyVar) { goto L2; }     // ... a jump forward to L2,
PCO1.send(MyVar);
PCO1.receive -> value MyVar2;
label L2;
PCO2.send(integer: 21);
alt {
    []  PCO1.receive
        { goto alt; }                // ... a jump which forces the re-evaluation of
                                     // the previous alt statement
    []  PCO2.receive(integer: 67)
        {   label L3;
            PCO2.send(MyVar);
            alt {
                []  PCO1.receive
                    { goto alt; }  // ... again a jump which forces the re-evaluation of the
                                   // the previous alt statement (not the same as for the
                                   // goto before),
                []  PCO2.receive(integer: 90)
                    {   PCO2.send(integer: 33);
                        PCO2.receive(integer: 13);
                        goto L4;   // ... a jump forward out of two nested alt statemens,
                    }
                []  PCO2.receive(MyError)
                    { goto L3; }   // ... a jump backward out of the current alt statement,
                []  any port.receive
                    { goto L2; }   // ... a jump backward out of two nested alt statements,
            }
        }
    []  any port.receive
        { goto L2; }      // ... and  a long jump backward out of an alt statement
}
label L4;
:
```

20.3 Interleaved behaviour

Control transfer statements **for**, **while**, **do-while**, **goto**, **activate**, **deactivate**, **stop**, **return** and (direct and indirect) calls of user-defined functions, which include communication operations, shall not be used in **interleave** statements. In addition, it is not allowed to guard branches of an **interleave** statement with Boolean expressions (i.e., the '[ ]' shall always be empty). It is also not allowed to expand **interleave** statements with named alternatives or to specify **else** branches in interleaved behaviour.

Interleaved behaviour can always be replaced by an equivalent set of nested alternatives. The procedures for this replacement are described in annex B.

The rule for the evaluation of an interleaving statement is the following:

a) whenever a reception statement is executed, the following non-reception statements are subsequently executed until the next reception statement is reached or the interleaved sequence ends;

NOTE: Reception statements are TTCN-3 statements which may occur in sets of alternatives i.e., **receive**, **check**, **trigger**, **getcall**, **getreply**, **catch** and **timeout**. Non-reception statements denote all other non-control-transfer statements which can be used within the interleaving statement.

b) the evaluation then continues by taking the next snapshot.

The operational semantics of interleaving are fully defined in annex B.

EXAMPLE:

```
// The following TTCN-3 code fragment
 :
interleave {
[]  PCO1.receive(MySig1)
    {   PCO1.send(MySig2);
        PCO1.receive(MySig3);
    }
[]  PCO2.receive(MySig4)
    {   PCO2.send(MySig5);
        PCO2.send(MySig6);
        PCO2.receive(MySig7);
    }
}
:

// can be interpreted as a shorthand for
:
alt {
[]  PCO1.receive(MySig1)
    {   PCO1.send(MySig2);
        alt {
        []  PCO1.receive(MySig3)
            {   PCO2.receive(MySig4);
                PCO2.send(MySig5);
                PCO2.send(MySig6);
                PCO2.receive(MySig7)
            }
        []  PCO2.receive(MySig4)
            {   PCO2.send(MySig5);
                PCO2.send(MySig6);
                alt {
                []  PCO1.receive(MySig3) {
                        PCO2.receive(MySig7); }
                []  PCO2.receive(MySig7) {
                        PCO1.receive(MySig3); }
                }
            }
        }
    }
[]  PCO2.receive(MySig4)
    {   PCO2.send(MySig5);
        PCO2.send(MySig6);
        alt {
        []  PCO1.receive(MySig1)
            {   PCO1.send(MySig2);
                alt {
                []  PCO1.receive(MySig3)
                    {   PCO2.receive(MySig7);
                    }
                []  PCO2.receive(MySig7)
                    {   PCO1.receive(MySig3);
                    }
                }
            }
        []  PCO2.receive(MySig7)
            {   PCO1.receive(MySig1);
                PCO1.send(MySig2);
                PCO1.receive(MySig3);
            }
        }
    }
}

:
```

20.4 Default behaviour

Default behaviour can be seen as an extension to an **alt** statement or a single receive operation which is defined in a special manner. A default behaviour shall be defined by specifying a named **alt** and activated before it can be invoked and executed.

Activation of a default means that the alternatives defined in the relevant named alt are appended to the top-level of all subsequent alternatives.

The default behaviour is also appended to any single (i.e., not in an **alt** statement) receiving operations, timeouts or done statements. This is because these operations are conceptually the same as one single alternative. For example:

```
:
MyPort.receive(MyMsg);
:

// Is the same as
:
alt {
    [] MyPort.receive(MyMsg) {}
}
:
```

20.4.1 The Activate and Deactivate operations

A default behaviour is activated by using the **activate** operation and deactivated by using the **deactivate** operation. An empty **deactivate** operation deactivates all active default behaviours.

In the case of multiple activation of multiple named alternatives the **alt** elements shall be expanded in the order of activation.

In the case where the argument to an activate operation is a list of named alternatives the **alt** elements shall be expanded in the order indicated by the list.

EXAMPLE:

```
named alt Default1() // named alt definition
{
    []  MyPort.check
        {MyBehaviour1()}
}

:

// inside behaviour definition
activate( Default1() );

CL2.receive(MySetup);

alt{
    []  CL2.receive(MySig1)
        {CL2.send(MySig2)}

    []  CL2.receive(MySig2)
        {CL2.send(mySig1)}
}

// This statement deactivates the default behaviour Default1
deactivate(Default1);
// This statement deactivates all previously activated default behaviour
deactivate;


// Conceptually, after definition and activation the default alt is expanded to the end of
// any following alt or receive statements

activate ( Default1() );
:
CL2.receive(MySetup);
```

```
alt{
    []  CL2.receive(MySig1)
        {CL2.send(MySig2)}
    []  CL2.receive(MySig2)
        {CL2.send(mySig1)}
}

// is equivalent to
:
alt{
    []  CL2.receive(MySetup); // The single receive now becomes an alt in its own right

    []  MyPort.check
        {MyBehaviour1()}
}

alt{
    []  CL2.receive(MySig1)
        {CL2.send(MySig2)}
    []  CL2.receive(MySig2)
        {CL2.send(mySig1)}

    []  MyPort.check
        {MyBehaviour1()}
}
```

20.5 The Return statement

The **return** statement terminates execution of a function and returns control to the point from which the function was called. A **return** statement may be optionally associated with a return value. Using **return** in a test case or control is equivalent to **stop**.

EXAMPLE:

```
function MyFunction() return boolean
{
    :
    if (date == "1.1.2000") { return false; }
// execution stops on the 1.1.2000 and returns false as a failure indication
    :
    return true;   // true is returned
}

function MyBehaviour() return verdicttype
{
    :
    if (MyFunction()) { verdict.set(pass); } // use of MyFunction in an if statement
    else { verdict.set(inconc); }
    :
    return verdict.get; // explicit return of the verdict
}
```

21 Configuration operations

Configuration operations are used to set up and control test components. These operations shall only be used in TTCN-3 test cases and functions (i.e., not in the module control part).

Table 13: Overview of TTCN-3 configuration operations

Configuration operations	
Statement	**Operation Name**
Create parallel test component	**create**
Connect one component to another component	**connect**
Disconnect two components	**disconnect**
Map component port to test interface port	**map**
Unmap port from test system interface	**unmap**
Get MTC address	**mtc**
Get test system interface address	**system**
Get own address	**self**
Start execution of test component	**start**
Stop execution of test component	**stop**
Check termination of a PTC	**running**
Wait for termination of a PTC	**done**

21.1 The Create operation

The MTC is the only test component which is automatically created when a test case starts. All other test components are created explicitly during test execution by **`create`** operations. A component is created with its full set of ports of which the input queues are empty. Furthermore, if a port is defined to be of the type **`in`** or **`inout`** it shall be in a listening state ready to receive traffic over the connection.

Since all components and ports are implicitly destroyed at the termination of each test case, each test case shall completely create its required configuration of components and connections when it is invoked.

```
// This example declares a variable of type address, which is used to store the reference of a
// newly created component of type MyComponentType which is the result of the create function.
  :
 var MyComponenttype MyNewComponent;
  :
 MyNewComponent := MyComponentType.create;
  :
```

The **`create`** operation shall return the unique component reference of the newly created instance. The unique reference to the component will typically be stored in a variable (see clause 8.6) and can be used for connecting instances and for communication purposes such as sending and receiving.

Components can be created at any point in a behaviour definition providing full flexibility with regard to dynamic configurations (i.e. any component can create any other component). The visibility of component references shall follow the same scope rules as that of variables and in order to reference components outside their scope of creation the component reference shall be passed as a parameter or as a field in a message.

21.2 The Connect and Map operations

The ports of a test component can be connected to other components or to the ports of the test system interface. In the case of connections between two test components the **connect** operation shall be used. When connecting a test component to a test system interface the **map** operation shall be used. The **connect** operation directly connects one port to another with the **in** side connected to the **out** side and vice versa. The **map** operation on the other hand can be seen purely as a name translation defining how communications streams should be referenced.

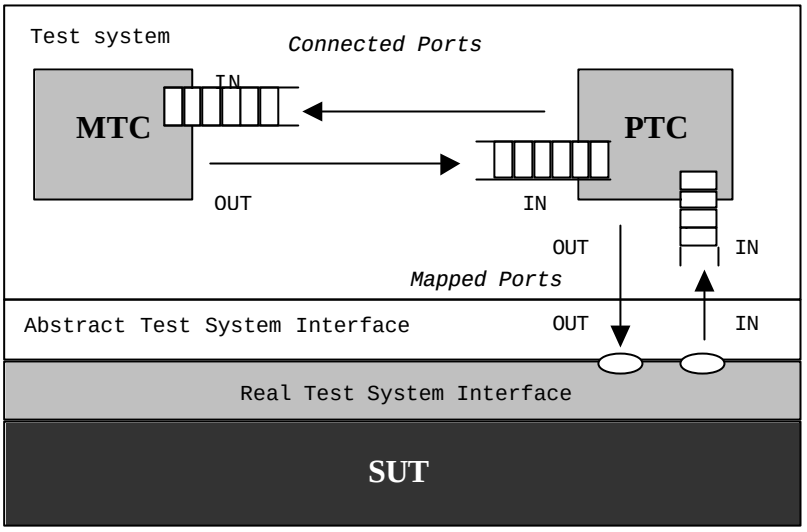


Figure 7: Illustration of the connect and map operations

With both the **connect** operation and the **map** operation, the ports to be connected are identified by the component references of the components to be connected and the names of the ports to be connected.

There are two operations for identifying the MTC i.e., **mtc**, and for identifying ports of the test system interface i.e., **system** (see clause 8.6). Both these operations can be used for identifying and connecting ports.

Both the **connect** and **map** operations can be called from any behaviour definition (function). However before either operation is called the components to be connected shall have been created and their component references shall be known together with the names of the relevant ports.

Both the **map** and **connect** operations allow the connection of a port to more than one other port. It is not allowed to connect to a mapped port or to map to a connected port.

EXAMPLE:

```
// It is assumed that the ports Port1, Port2, Port3 and PCO1 are properly defined and declared
// in the corresponding port type and component type definitions
 :
var MyComponentType  MyNewComponent;
 :
MyNewComponent := MyComponentType.create;
 :
 :
connect(MyNewComponent:Port1, mtc:Port3);
map(MyNewComponent:Port2, system:PCO1);
 :
 :
// In this example a new component of type MyComponentType is created and its reference stored
// in variable MyNewComponent. Afterwards in the connect operation, Port1 of this new component
// is connected with Port3 of the MTC. By means of the map operation, Port2 of the new component
// is then connected to port PCO1 of the test system interface
```

21.2.1 Consistent connections

For both the **connect** and **map** operations only consistent connections are allowed.

Assuming the following:

a) ports PORT1 and PORT2 are the ports to be connected;

b) inlist-PORT1 defines the messages or procedures of the in-direction of PORT1;

c) outlist-PORT1defines the messages or procedures of the out-direction of PORT1;

d) inlist-PORT2 defines the messages or procedures of the in-direction of PORT2; and

e) outlist-PORT2 defines the messages or procedures of the out-direction of PORT2.

The **connect** operation is allowed if and only if:

- outlist-PORT1 $\subseteq$ inlist-PORT2 *and* outlist-PORT2 $\subseteq$ inlist-PORT1.

The **map** operation (assuming PORT2 is the test system interface port) is allowed if and only if:

- outlist-PORT1 $\subseteq$ outlist-PORT2 *and* inlist-PORT2 $\subseteq$ inlist-PORT1.

In all other cases, the operations shall not be allowed.

Since TTCN-3 allows dynamic configurations and addresses, not all of these consistency checks can be made statically at compile-time. All checks, which could not be made at compile-time, shall be made at run-time and shall lead to a test case error when failing.

21.3 The Disconnect and Unmap operations

The **disconnect** and **unmap** operations are the opposite operations of **connect** and **map**. They perform the disconnection (of previously connected) ports of test components and the unmapping of (previously mapped) ports of test components and ports in the test system interface.

Both, the **disconnect** and **unmap** operations can be called from any component if the relevant component references together with the names of the relevant ports are known. A **disconnect** or **unmap** operation has only an effect if the connection or mapping to be removed has been created beforehand.

EXAMPLE:

```
  :
  :
connect(MyNewComponent:Port1, mtc:Port3);
map(MyNewComponent:Port2, system:PCO1);
  :
  :
disconnect(MyNewComponent:Port1, mtc:Port3);// disconnect previously made connection
unmap(MyNewComponent:Port2, system:PCO1);    // unmap previously made mapping
```

21.4 The MTC, System and Self operations

The component reference (see clause 8.6) has two operations, **mtc** and **system** which return the reference of the master test component and the test system interface respectively. In addition, the operation **self** can be used to return the reference of the component in which it is called. For example:

```
var MyComponentType MyAddress;
MyAddress := self; // Store the current component reference
```

The only operations allowed on component references are assignment and equivalence.

21.5 The Start test component operation

Once a component has been created and connected behaviour has to be bound to the component and the execution of its behaviour has to be started. This is done by using the **start** operation (component creation does not start execution of the component behaviour). The reason for the distinction between **create** and **start** is to allow connection operations to be done before actually running the test component.

The **start** operation shall bind the required behaviour to the test component. This behaviour is defined by reference to an already defined function. For example:

```
    // It is assumed that the ports Port1, Port2, Port3 and PCO1 are properly defined and declared
    // in the corresponding port type and component type definitions
     :
    var MyComponentType MyNewComponent;
     :
    MyNewComponent := MyComponentType.create;
     :
    connect(MyNewComponent:Port1, mtc:Port3);
    connect(MyNewComponent:Port2, system:PCO1);
     :
     :
    MyNewComponent.start(MyComponentBehaviour());
     :

// In this example, a new component is first created, then connected to its environment and lastly
// it is started by means of the start operation. For identifying the component to be executed its
// reference is used
```

The following restrictions apply to a function invoked in a **start** test component operation:

- If this function has parameters they shall only be **in** parameters, i.e., value parameters.
- This function shall either have a **runs on** definition referencing the same component type as the newly created component or shall pass in all information needed from the component type definition as parameters.
- Ports and timers can only be passed into this function if they refer to ports and timers in the component type definition of the newly created component, i.e., ports and timers are local to component instances and shall not be passed to other components.

NOTE: The ability to pass ports in as parameters allows the specification of generic functions that are not tied to one specific component type.

21.6 The Stop test component operation

The **stop** test component statement explicitly stops the execution of the test component in which the stop is called. The operation has no arguments. For example:

```
    if (date == "1.1.2000") { stop; }// execution stops on the 1.1.2000
```

If the test component that is stopped is the MTC all remaining PTCs that are still running shall also be stopped and the test case terminates.

NOTE: The concrete mechanism for stopping all remaining running PTCs is outside the scope of the present document.

All resources shall be released when a test component terminates, either explicitly using the **stop** operation or through reaching a **return** statement in the function that originally started the test component or implicitly when the component reaches the end of its behaviour tree. Any variables storing a stopped component reference shall refer to nothing.

The rules for the termination of test cases and the calculation of the final test verdict are described in clause 24.

21.7 The Running operation

The **running** operation allows behaviour executing on a test component to ascertain whether behaviour running on a different test component has completed. The **running** operation is considered to be a **boolean** expression and, thus, returns a **boolean** value to indicate whether the specified test component (or all test components) has terminated. In contrast to the **done** operation, the **running** operation can be used freely in **boolean** expressions. For example:

```
if (PTC1.running)               // usage of running in an if statement
{
    // Do something!
}


while (all component.running != true) {// usage of running in a loop condition
    MySpecialFunction()
}
```

21.8 The Done operation

The **done** operation allows behaviour executing on a test component to ascertain whether the behaviour running on a different test component has completed.

The **done** operation shall be used in the same manner as a receiving operation or a **timeout** operation. This means it shall not be used in a **boolean** expression, but it can be used to determine an alternative in an **alt** statement or as stand-alone statement in a behaviour description. In the latter case a **done** operation is considered to be a shorthand for an **alt** statement with only one alternative, i.e., it has blocking semantics, and therefore provides the ability of passive waiting for the termination of test components.

NOTE: The TTCN-3 **done** operation and the DONE operation TTCN-2 have identical semantics.

EXAMPLE:

```
// Use of done in alternatives
:
alt {
    []  MyPTC.done {
            verdict.set(pass)
        }

    []  any port.receive {
            goto alt
        }
}
:


// the following done as stand-alone statement:
 :
all component.done;
 :

// has the following meaning:
 :
alt {
    []  all component.done {}
}
 :

// and thus, blocks the execution until all parallel test components have terminated
```

21.9 Using component arrays

The **create**, **connect**, **start** and **stop** operations do not work directly on arrays of components. Instead a specific element of the array shall be provided as the parameter. For components the effect of an array is achieved by using an array of component references and assigning the relevant array element to the result of the **create** operation.

```
// This example shows how to model the effect of creating, connecting and running arrays of
// components using a loop and by storing the created component reference in an array of
// component references.

testcase MyTestCase() runs on MyMtcType system MyTestSystemInterface
{
     :
    var integer i;
    var MyPTCType1 MyPtcType[11];
     :
    for (i:= 1; i<=10; i:=i+1)
    {
        MyPtcAddresses[i] := MyPtcType1.create;
        connect(self:PtcCoordination, MyPtcAddresses[i]:MtcCordination);
        MyPtcAddresses[i].start(MyPtcBehaviour());
    }
     :
}
```

21.10 Use of Any and All with components

The keywords **any** and **all** may be used with configuration operations as indicated in table 14.

Table 14: Any and All with components

Operation	Allowed		Example
	any	all	
create			
start			
running	Yes but from MTC only	Yes but from MTC only	any component.running all component.running
done	Yes but from MTC only	Yes but from MTC only	any component.done all component.done
stop			

22 Communication operations

TTCN-3 supports message-based (asynchronous) and procedure-based (synchronous) communication (see clause 8.1). Asynchronous communication is non-blocking on the **send** operation, as illustrated in figure 8 where processing in the MTC continues immediately after the **send** operation occurs. The SUT is blocked on the **receive** operation until it receives a sent message.

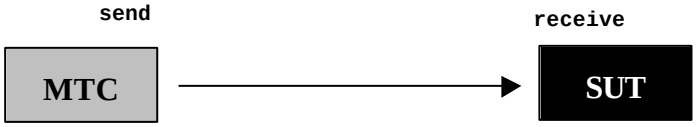


Figure 8: Illustration of the asynchronous send and receive

Synchronous communication is blocking on the **call** operation, as illustrated in figure 9 where the **call** operation blocks processing in the MTC until either a **reply** or exception is received from the SUT. Similar to the **receive**, the **getcall** blocks the SUT until the call is received.

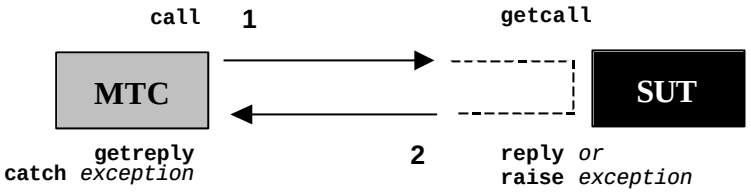


Figure 9: Illustration of a complete synchronous call

Operations such as **send** and **call** are collectively known as communication operations. These operations shall only be used in TTCN-3 test cases and functions (i.e., not directly in the module control part). The communication operations are divided into three groups:

a) a component sends a message, calls a procedure, or replies to an accepted call or raises an exception. These actions are collectively referred to as *sending operations*;

b) a component receives a message, accepts a procedure call, receives a reply for a previously called procedure or catches an exception. These actions are collectively referred to as *receiving operations*;

c) control of access to a port by doing a **clear**, **start** or **stop**. These actions are collectively referred to as *controlling operations.*

These operations can be used on the communication ports of a test component as summarized in table 15. In cases of mixed ports all the operations are applicable.

Table 15: Overview of TTCN-3 communication operations

Communication operations			
Communication operation	**Keyword**	**Can be used at message-based ports**	**Can be used at procedure-based ports**
Sending operations			
Send message	**send**	Yes	
Invoke procedure call	**call**		Yes
Reply to procedure call from remote entity	**reply**		Yes
Raise exception (to an accepted call)	**raise**		Yes
Receiving operations			
Receive message	**receive**	Yes	
Trigger on message	**trigger**	Yes	
Accept procedure call from remote entity	**getcall**		Yes
Handle response from a previous call	**getreply**		Yes
Catch exception (from called entity)	**catch**		Yes
Check msg/call/exception/reply received	**check**	Yes	Yes
Controlling operations			
Clear port	**clear**	Yes	Yes
Clear and give access to port	**start**	Yes	Yes
Stop access (receiving & sending) to port	**stop**	Yes	Yes

22.1 Sending operations

The sending operations are:

a) **send**: send a message asynchronously;

b) **call**: call a procedure;

c) **reply**: reply to an accepted procedure call from the SUT; and

d) **raise**: raise an exception in cases where a procedure call is received.

22.1.1 General format of the sending operations

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

The send part:

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

The port name, operation name and value shall be present in all sending operations. The identification of the communication partner (denoted by the **to** keyword) is optional and need only be specified in cases of one-to-many connections where the receiving entity shall be explicitly identified.

22.1.1.1 Response and exception handling

Response and exception handling is only needed in cases of synchronous communication. The response and exception handling part of the **call** operation is optional and is required for cases where the called procedure returns a value or has **out** or **inout** parameters whose values are needed within the calling component and for cases where the called procedure may raise exceptions which need to be handled by the calling component.

The response and exception handling part of the call operation makes use of **getreply** and **catch** operations to provide the required functionality.

22.1.2 The Send operation

The **send** operation is used to place a value on an outgoing message port queue. The value may be specified by referencing a template, a variable, or a constant or can be defined in-line from an expression (which of course can be an explicit value). When defining the value in-line the optional type field shall be used if there is ambiguity of the type of the value being sent.

The **send** operation shall only be used on message-based (or mixed) ports and the type of the value to be sent shall be in the list of outgoing types of the port type definition. For example:

```
MyPort.send(MyTemplate(5,MyVar));
// Sends the template MyTemplate with the actual parameters 5 and MyVar via MyPort.

MyPort.send(integer:5);
// Sends the integer value 5
```

In cases of one-to-many connections the communication partner shall be specified uniquely. This shall be denoted using the **to** keyword. For example:

```
MyPort.send("My string") to MyPartner;
// Sends the string "My string" to a component with a component reference stored in the
// variable MyPartner.

MyPCO.send(MyVariable + YourVariable - 2) to MyPartner;
// Sends the result of the arithmetic expression to MyPartner.
```

22.2.1 The Call operation

The **call** operation is used to specify that a test component calls a procedure in the SUT or in another test component. The **call** is a blocking operation in that it shall wait until it receives a response (i.e., a **reply**) or an exception from the called entity. In other words the **call** operation works in a synchronous manner.

NOTE: This is comparable with the testing of server functionality i.e., the SUT is the server and the component plays the role of a client.

The **call** operation shall only be used on procedure-based (or mixed) ports. The type definition of the port at which the call operation takes place shall include the procedure name in its **out** or **inout** list i.e., it must be allowed to call this procedure at this port.

The value of the **call** operation is a signature that may either be defined in the form of a signature template or be defined in-line. For example:

```
signature MyProc (out integer MyPar1, inout boolean MyPar2);
:
MyPort.call(MyProc:{MyVar1,MyVar2});
// Calls the remote procedure MyProc at MyCL with the in and inout parameters 5 and MyVar.
// Neither a return value nor an exception is expected from this call. If one (or both) of the
// two parameters is defined to be an inout parameter, its value will not be considered i.e.,
// it is not assigned to a variable.


// The following example explains the possibilities to assign values to in and inout parameters
// on the call argument. The following signature is assumed for the procedure to be called.
// Note: MyProc2 has no return value and no exceptions
signature MyProc2 (in integer A, out integer B, inout integer C);
:
MyPort.call(MyProc2:{1, - , 3});
// Only values of in and inout parameters are specified The returned values of out and inout
// parameters are not used after the call and, thus, not assigned to variables.
```

All **in** and **inout** parameters of the signature shall have a specific value i.e., the use of matching mechanisms such as *AnyValue* is not allowed.

The signature arguments of the **call** operation are not used to retrieve variable names for **out** and **inout** parameters. The actual assignment of the procedure return value and **out** and **inout** parameter values to variables shall explicitly be made in the response (**getreply**) and exception handling (**catch**) part of the **call** operation. This is denoted by the keywords **value** and **param** respectively. This allows the use of signature templates in **call** operations in the same manner as templates can be used for types.

In general, a **call** operation is assumed to have blocking-semantics. However, TTCN-3 also supports non-blocking calls. A call, which has no return values, is assumed to be a non-blocking call. Exceptions (if specified) raised by a call without return values shall be caught within a following **alt** statement. In addition, it is also possible to force non-blocking semantics by the **nowait** keyword (see clause 22.2.12).

In cases of one-to-many connections the communication partner shall be specified uniquely. This shall be denoted using the keyword **to**. For example:

```
MyPort.call(MyProc:{MyVar1, MyVar2}) to MyPartner;
// In this example the called party is explicitly identified by the component reference stored
// in the variable MyPartner.
```

22.2.1.1 Handling responses to a Call

The handling of the response to a call is done by means of the **getreply** operation (see clause 22.3.5). This operation defines the alternative behaviour depending on the response that has been generated as a result of the **call** operation. For example:

```
MyPort.call(MyProc:{MyVar1, MyVar2}) to MyPartner // Where { … } is an inline template
{
    [] MyCl.getreply(MyProc:{MyVar1, MyVar2}) {}
}
```

If needed, the return value of the called procedure shall be picked up explicitly in the **getreply** operation. This is expressed using '->' and the (optional) keyword **value**. For example:

```
MyPort.call(MyProc:{MyVar1, MyVar2}) to MyPartner
{
    [] MyCl.getreply(MyProc:{MyVar1, MyVar2}) -> value MyResult {}
}
// A value shall be returned by MyProc which will be stored in the variable MyResult.
```

The signature arguments of the **call** operation are not used to retrieve variable names for **out** and **inout** parameters. The actual assignment of the procedure return value and **out** and **inout** parameter values to variables shall explicitly be made in the response (**getreply**) and exception handling (**catch**) part of the **call** operation. This is denoted by the keywords **value** and **param** respectively. This allows the use of signature templates in **call** operations in the same manner as templates can be used for types. For example:

```
MyPort.call(MyProc:{5,MyVar}) to MyPartner
{
    []MyCl.getreply(MyProc:{MyVar1, MyVar2}) -> value MyResult param (MyPar1Var,MyPar2Var) {}
}
// In this example both parameters of MyProc are specified as inout parameters and their values
// after the termination of MyProc are assigned to MyPar1Var and MyPar2Var.
```

22.2.1.2 Handling exceptions to a Call

The handling of exceptions to a call is done by means of the **catch** operation (see clause 22.3.6). This operation defines the alternative behaviour depending on the exception (if any) that has been generated as a result of the **call** operation. For example:

```
signature MyProc3 (out integer MyPar1, inout boolean MyPar2) return MyResultType
    exception (ExceptionTypeOne, ExceptionTypeTwo, ExceptionTypeThree);
:
// The following call operation shows the getreply and exception handling mechanism of the
// call operation

MyPort.call(MyProc3:{5,MyVar}, 30E-3) to MyPartner
{
    []  MyCl.getreply(MyProc3:{MyVar1, MyVar2}) -> value MyResult param (MyPar1Var,MyPar2Var) {}
    []  MyPort.catch(MyProc3, MyExceptionOne)
        {                                          // catch an exception
            verdict.set(fail);                     // set the verdict and
            stop                                   // stop as result of the exception
        }
    []  MyPort.catch(MyProc3, MyExceptionTwo)      // catch a second exception
        {verdict.set(inconc)}                      // set the verdict and continue after
                                                   // the call as result of the
                                                   // second exception

    [MyCondition] MyPort.catch(MyProc3, MyExceptionThree) {} // catch a third exception which
                                                             // may occur if MyCondition
                                                             // evalutates to true

    []  MyPort.catch(timeout) {}  // timeout exception i.e., the called party
                                  // does not react in time, nothing is done
}
```

22.2.1.3 Handling timeout exceptions to the Call

The **call** operation may optionally include a timeout. This is defined as an explicit value or constant of **float** type and defines the length of time after the **call** operation has started that a **timeout** exception shall be generated by the test system. If no timeout value part is present in the **call** operation no **timeout** exception shall be generated. For example:

```
MyPort.call(MyProc:{5,MyVar}, 20E-3)
{
    []  MyPort.catch(timeout)
        {
            verdict.set(fail);
            stop
        }
}
// This example shows a call with a timeout value of 20ms. This means if the called party does
// not respond with a reply or exception within this time the test system will automatically
// generate a timeout exception. The handling of the timeout is done by means of a catch
// operation. If the procedure completes without a timeout exception, execution will continue
// with the statement following the call operation.
```

Using the keyword **nowait** in the timeout value part of a **call** operation allows calling a procedure without waiting either for a termination, a response, an exception raised by the called procedure or a timeout exception. For example:

```
MyPort.call(MyProc:{5, MyVar}, nowait);
// In this example the test component will continue execution without
```

```
// waiting for the termination of MyProc.
```

In such cases a possible response or exception has to be removed from the queue by using a **getreply** or a **catch** operation in a subsequent **alt** statement.

22.2.2 The Reply operation

The **reply** operation is used to reply to a previously accepted call according to the procedure signature. A **reply** operation shall only be used at a procedure-based (or mixed) port. The type definition of the port shall include the name of the procedure to which the **reply** operation belongs.

The value part of the **reply** operation consists of a signature reference with an associated actual parameter list and (optional) return value. The signature may either be defined in the form of a signature template or it may be defined in-line. All **out** and **inout** parameters of the signature shall have a specific value i.e., the use of matching mechanisms such as *AnyValue* is not allowed. For example:

```
MyPort.reply(MyProc2:{ - ,5});
// Replies to an accepted call of MyProc2. The MyProc2 has no return value but two parameters.
// The first parameter is an in parameter i.e., its value will not be replied and therefore
// needs not to be specified. The second parameter is either an out or an inout parameter. Its
// value is 5.
```

In cases of one-to-many connections the communication partner shall be specified explicitly and shall be unique. This shall be denoted using the **to** keyword. For example:

```
MyPort.reply(MyProc3:{ - ,5}) to MyPartner;
// This example is identical to previous one, but the reply is directed to a component with a
// component reference stored in variable MyPartner
```

If a value is to be returned to the calling party this shall be explicitly stated using the **value** keyword.

```
MyPort.reply(MyProc:{5,MyVar} value 20);
// Replies to an accepted call of MyProc. The return value of MyProc is 20 and it has two
// parameters which are out or inout parameters. Their values are provided by 5 and MyVar.
```

22.2.3 The Raise operation

The **raise** operation is used to raise an exception. An exception shall only be raised at a procedure-based (or mixed) port. An exception is a reaction to an accepted procedure call the result of which leads to an exceptional event. The type of the exception shall be specified in the signature of the called procedure. The type definition of the port shall include in its list of accepted procedure calls the name of the procedure to which the exception belongs.

NOTE: The relation between an accepted call and a **raise** operation cannot always be checked statically. For testing it is allowed to specify a **raise** operation without an associated **getcall** operation.

The value part of the **raise** operation consists of the signature reference followed by the exception value. For example:

```
MyPort.raise(MySignature, MyVariable + YourVariable - 2);
// Raises an exception with a value which is the result of the arithmetic expression
// at MyPort
```

Exceptions are specified as a type. Therefore the exception value may either be derived from a template or be the value resulting from an expression (which of course can be an explicit value). The optional type field in the value specification to the **raise** operation shall be used in cases where it is necessary to avoid any ambiguity of the type of the value being sent. For example:

```
MyPort.raise(MyProc, MyExceptionType:{5, MyVar});
// Raises an exception from the remote procedure defined by Myproc with the value defined
// by template MyExceptionTemplate with the actual parameters 5 and MyVar at port MyPort
```

In cases of one-to-many connections the communication partner shall be specified uniquely. This shall be denoted using the keyword **to**. For example:

```
MyPort.raise(MySignature, "My string") to MyPartner;
// Raises a string exception with the value"My string" at MyPort to to a component with an
// component reference stored in variable MyPartner
```

22.3 Receiving operations

The receiving operations are:

a) **`receive`**: receive an asynchronously sent message;

b) **`trigger`**: trigger on the reception of a specific message;

c) **`getcall`**: accept a procedure call;

d) **`getreply`**: handling the reply to a previously called procedure;

e) **`catch`**: catch an exception which has be raised as a reaction to a call operation; and

f) **`check`**: check the top element of the in-queue of a particular port.

22.3.1 General format of the receiving operations

Receiving operation consists of a *receive* part and an *assignment* part.

The receive part:

a) specifies the port at which the operation shall take place;

b) defines a matching part which specifies the acceptable input which will match the statement;

c) gives an (optional) address expression which uniquely identifies the communication partner (in case of one-to-many connections).

The port name, operation name and value part of all receiving operations shall be present. The identification of the communication partner (denoted by the **`from`** keyword) is optional and need only be specified in cases of one-to-many connections where the receiving entity needs to be explicitly identified.

22.3.1.1 Making assignments on receiving operations

The assignment part in a receiving operation is optional. For message-based ports it is used when it is required to store received messages. In the case of procedure-based ports it is used for storing the **`in`** and **`inout`** parameters of an accepted call or for storing exceptions.

In addition, the assignment part may also be used to assign the **`sender`** address of a message, exception, **`reply`** or **`call`** to a variable. This is useful for one-to-many connections where, for example, the same message or call can be received from different components, but the message, **`reply`** or exception must be sent back to the original sending component.

22.3.2 The Receive operation

The **`receive`** operation is used to receive a value from an incoming message port queue. The value may be specified by referencing a template, a variable, or a constant or can be defined in-line from an expression (which of course can be an explicit value). When defining the value in-line the optional type field shall be used to avoid any ambiguity of the type of the value being received. The **`receive`** operation shall only be used on message-based (or mixed) ports and the type of the value to be received shall be included in the list of incoming types of the port type definition.

The **`receive`** operation removes the top message from the associated incoming port queue if, and only if, that top message satisfies all the matching criteria associated with the **`receive`** operation. No binding of the incoming values to the terms of the expression or to the template shall occur.

If the match is not successful, the top message shall not be removed from the port queue i.e., if the **`receive`** operation is not successful the execution of the test case shall continue with the next alternative.

The matching criteria are related to the type and value of the message to be received. The type and value of the message to be received may either be derived from a template or be the value resulting from an expression (which of course can be an explicit value).

```
MyPort.receive(MyTemplate(5, MyVar));
// Specifies the reception of a value which fulfils the conditions defined by the template
// MyTemplate with actual parameters 5 and MyVar.

MyPort.receive(A<B);
// Specifies the reception of a Boolean value true or false depending on the outcome of A<B
```

An optional type field in the matching criteria to the **receive** operation shall be used to avoid any ambiguity of the type of the value being received. For example:

```
MyPort.receive(integer:MyVar);
// Specifiess the reception of an integer value which has the same value as the variable MyVar
// at MyPort. The (optional) type identifier integer is not strictly necessary because the
// type is already given by the definition of MyVar. However, in complex and long test cases
// such a type identifier may be used to improve readability.

MyPort.receive(MyVar);
// Is an alternative to the previous example.
```

If the match is successful, the value removed from the port queue can be stored in a variable and the address of the component that sent the message, can be retrieved and stored in a variable. This is denoted by the symbol '->' and the keyword **value**. For example:

```
MyPort.receive(MyType:*) from MyPartner -> value MyVar;
// Specifies the reception of an arbitrary value of MyType (from a component with an address
// stored in variable MyPartner) which afterwards is assigned to the variable MyVar. MyVar has
// to be of the type MyType.
```

In the case of one-to-many connections the **receive** operation may be restricted to a certain communication partner. This restriction shall be denoted using the **from** keyword.

```
MyPort.receive(charstring:"Hello")from MyPartner;
// Specifies the reception of the charstring "Hello" from a component with a component reference
// or address stored in the variable MyPartner.
```

It is also possible to retrieve the component reference or address of the sender of a message. This is denoted by the keyword **sender**. For example:

```
MyPort.receive(MyTemplate:{5, MyVarOne}) -> value MyVarTwo sender MyPartner;
// Specifies the reception of a value which fulfils the conditions defined by the template
// MyTemplate with actual parameters 5 and MyVarOne. After reception the value is assigned to
// the variable MyVarTwo. The reference of the sender component is retrieved by call operation
// and assigned to variable MyPartner.

MyPort.receive(A<B) -> sender MyPartner;
// Specifies the reception of a Boolean value of true or false depending on the outcome of A<B.
// The component reference of the sender component is retrieved by call operation and assigned
// to variable MyPartner.
```

22.3.2.1 Receive any message

A **receive** operation with no argument list for the type and value matching criteria of the message to be received shall remove the message on the top of the incoming port queue (if any) if all other matching criteria are fulfilled.

NOTE: This is equivalent to the TTCN-2 OTHERWISE statement.

A message received by *ReceiveAnyMessage* shall not be assigned to a variable.

EXAMPLE:

```
MyPort.receive;
// Removes the top value from MyPort.MyPort.

MyPort.receive from MyPartner;
// Removes the top value from CL1 if it is a message from the component with the
addressreference

MyPort.receive -> sender MySenderVar;
```

```
// Removes the top value from CL1, but remembers the sending instance by storing its reference
// in MySenderVar
```

22.3.2.2 Receive on any port

To **receive** a message on any port use the **any** keyword. For example:

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

22.3.3 The Trigger operation

The **trigger** operation filters messages with certain matching criteria from a stream of received messages on a given incoming port. The **trigger** operation shall only be used on message-based (or mixed) ports and the type of the value to be received shall be included in the list of incoming types of the port type definition. All messages that do not fulfil the matching criteria shall be removed from the queue without any further action i.e., the trigger operation waits for the next message on that queue. If a message meets the matching criteria, the **trigger** operation behaves in the same manner as a **receive** operation. For example:

```
MyPort.trigger(MyType:*);
// Specifies that the operation will trigger on the reception of the first message observed of
// the type MyType with an arbitrary value at port MyPort.
```

The **trigger** operation requires the port name, matching criteria for type and value, an optional **from** restriction (i.e., selection of communication partner) and an optional assignment of the matching message and sender component to variables.

EXAMPLE:

```
MyPort.trigger(MyType:*) from MyPartner;
// Specifies that the operation will trigger on the reception of the first message observed of
// the type MyType with an arbitrary value at port MyCL coming from a component with a reference
// identical to the one stored in the variable MyPartner.

MyPort.trigger(MyType:*) from MyPartner -> value MyRecMessage;
// This example is almost identical to the previous example. The addition is that the message
// which triggers i.e., all matching criteria are met, is stored in the variable MyRecMessage.

MyPort.trigger(MyType:*) -> sender MyPartner;
// Specifies that the operation will trigger on the reception of the first message observed of
// the type MyType with an arbitrary value at MyPort. The reference of the sender component
// of this message will be stored in the variable MyPartner.

MyPort.trigger(integer:*) -> value MyVar sender MyPartner;
// Specifies that the operation will trigger on the reception of an arbitrary integer value
// which afterwards is stored in the variable MyVar and the reference of the sender component of
// this message will be stored in the variable MyPartner.
```

22.3.3.1 Trigger on any message

A **trigger** operation with no argument list shall trigger on the receipt of any message. Thus, its meaning is identical to the meaning of receive any message. A message received by *TriggerOnAnyMessage* shall not be assigned to a variable.

EXAMPLE:

```
MyPort.trigger;

MyPort.trigger from MyPartner;

MyPort.trigger -> sender MySenderVar;
```

22.3.3.2 Trigger on any port

To **trigger** on a message at any port use the **any** keyword. For example:

```
any port.trigger
```

22.3.4 The Getcall operation

The **getcall** operation is used to specify that a test component accepts a call from the SUT, or another test component. The **getcall** operation shall only be used on procedure-based (or mixed) ports and the signature of the procedure call to be accepted shall be included in the list of allowed incoming procedures of the port type definition.

```
MyPort.getcall(MyProc(5, MyVar));
// Will accept a call of MyProc at MyCL with the in or inout parameters 5 and value of MyVar.
```

The **getcall** operation shall remove the top call from the incoming port queue, if, and only if, the matching criteria associated to the **getcall** operation are fulfilled. These matching criteria are related to the signature of the call to be processed and the communication partner. The matching criteria for the signature may either be specified in-line or be derived from a signature template.

A **getcall** operation may be restricted to a certain communication partner in case of one-to-many connections. This restriction shall be denoted using the **from** keyword.

```
MyPort.getcall(MyProc:{5, MyVar}) from MyPartner;
// Will accept a call of MyProc at MyCL (with the in or inout parameters 5 and value of MyVar)
// from a peer entity with the address or component reference stored in variable MyPartner.
```

The assignment part of the **getcall** operation comprises the optional assignment of **in** and **inout** parameter values to variables and the retrieval and assignment of the address of the calling component to a variable.

The keyword **param** is used to retrieve the parameter values of a call. For example:

```
MyPort.getcall(MyProc:{5, MyVar}) from MyPartner -> param (MyPar1Var, MyPar2Var);
// Both parmeters of MyProc are inout parameters and that their values are assigned
// to MyPar1Var and MyPar2Var. The identification of parameters defined in the procedure
// signature and the names in the list of variable names following the param keyword in the
// accept operation above is done by the order in the list
```

The keyword **sender** is used when it is required to retrieve the address of the sender (e.g., for addressing a **reply** or exception to the calling party in a one-to-many configuration).

```
MyPort.getcall(MyProc:{5, MyVar}) -> sender MySenderVar;
    // Will accept a call of MyProc at MyCL with the in or inout parameters 5 and MyVar. The calling
    // party is retrieved by the accept operation and stored in MySenderVar. This allows to handle
    // call of the same procedure from several components at the same port in the same manner.
    // MySenderVar can be used to reply or raise an exception to the calling component.
```

The signature argument of the **getcall** operation shall not be used to pass in variable names for **in** and **inout** parameters. The assignment of **in** and **inout** parameter values to variables shall be made in the assignment part of the **getcall** operation. This allows the use of signature templates in **getcall** operations in the same manner as templates are used for types.

The following **getcall** operations show the possibilities to use matching attributes and omit optional parts, which may be of no importance for the test specification.

EXAMPLE:

```
MyPort.getcall(MyProc:{5, MyVar}) -> param(MyPar1Var, MyPar2Var) sender MySenderVar;

MyPort.getcall(MyProc:{5, *}) -> param(MyPar1Var, MyPar2Var);

MyPort.getcall(MyProc:{*, MyVar}) -> param( - , MyPar2Var);
// Value of the first inout parameter is not important or not used

// The following examples shall explain the possibilities to assign in and inout parameter
// values to variables. The following signature is assumed for the procedure to be called

signature MyProc2(in integer A, integer B, integer C, out integer D, integer E, inout integer F);
// MyProc2 has no return value and no exceptions

MyPort.getcall(MyProc2:{*, *, 3, - , - , *}) ->
                                    param(MyVarIn1, MyVarIn2, MyVarIn3, - , - ,MyVarInout1);
// The in parameters A, B and C are assigned to the variables MyVarIn1, MyVarIn2 and MyVarIn3
// the inout paramameter F is assigned to variable MyVarInout1. The out parameters D and E need
// not to be considered in the assignment part of the accept operation.
```

```
MyPort.getcall(MyProc2:{*, *, *, - , - , *}) -> param(MyVarIn1:=A, MyVarIn2:=B, MyVarIn3:=C,
MyVarInout1:=F);
// Alternative notation for the value assignment of in and inout parameter to variables. Note,
// the names in the assignment list refer to the names used in the signature of MyProc2

MyPort.getcall(MyProc2:{1, 2, 3, - , - ,*}) -> param(MyVarInout1:=F);
// Only the inout parameter value is needed for the further test case execution
```

22.3.4.1 Accepting any call

A **getcall** operation with no argument list for the signature matching criteria will remove the call on the top of the incoming port queue (if any) if all other matching criteria are fulfilled. Parameters of calls accepted by *AcceptAnyCall* shall not be assigned to a variable.

EXAMPLE:

```
MyPort.getcall;
// Removes the top call from MyPort.

MyPort.getcall from MyPartner;
// Removes the top call from CL1 if the calling party is an entity with an address or component
// reference stored in the variable MyPartner.

MyPort.getcall -> sender MySenderVar;
// Removes the top call from CL1, but remembers the calling party by storing its address or
// component reference in MySenderVar
```

22.3.4.2 Getcall on any port

To **getcall** on any port is denoted by the any keyword. For example:

```
any port.getcall(MyProc)
```

22.3.5 The Getreply operation

The **getreply** operation is used to handle replies from a previously called procedure. A **getreply** operation shall only be used at a procedure-based (or mixed) port. For example:

```
MyPort.getreply(MyProc:{5, MyVar} value 20);
// Accepts a reply of procedure MyProc where the returned value is 20 and the values of the two
// out or inout parameters is 5 and the value of MyVar.

MyPort.getreply(MyProc2:{ - , 5});
// Accepts a reply from MyProc2. MyProc2 has no return value but two parameters. The first
// parameter is an in parameter i.e., its value will not be replied and therefore will not be
// considered for matching. The second parameter is either an out or an inout parameter. Its
// value has to be 5.
```

It may either be used in the **getreply** and exception part of a call, for example:

```
MyPort.call (MyProc) to MyPeer
{
    [ ] MyPort.getreply(MyProc:*) {}
    [ ] MyPort.catch {}
}
```

or within an **alt** statement, for example:

```
MyPort.call (MyProc, nowait) to MyPeer;
:
alt
{
    [ ] MyPort.getreply(MyProc:*) {}
    :
}
```

If used in an **alt** statement the **getcall** should cover cases where the response of a previously called procedure arrives too late i.e., a timeout exception has been raised.

As with other receiving operations matching mechanisms are allowed in the **getreply** operation in order to distinguish between replies from a previously called procedure which either differ in the returned value and/or the value of **out** and **inout** parameters.

```
MyPort.getreply(MyProc1:{*, MyVar});
// In this example there is no restriction on the returned value and the value of the
// first parameter.


MyPort.getreply(MyProc1:{*, *});
// The getreply operation will match with any reply from MyProc1 with any returned value. The
// stars are inline template definitions for MyProc1 and the return type of MyProc1.
```

In cases of one-to-many connections the **getreply** operation allows to distinguish between different communication partners by using a **from** clause.

```
MyPort.getreply(MyProc2:{ - ,5}) from MyPartner;
// The reply is only accepeted if it is from a component with the reference specified in the
// variable MyPartner
```

The optional assignment part of the **getreply** operation allows to assign values of **out** and **inout** parameters and returned values to variables.

EXAMPLE:

```
MyPort.getreply(MyProc1:{*, *} value *) -> value MyReturnValue param(MyPar1,MyPar2);
// After acceptance, the returned value is assigned to variable MyReturnValue and the value
// of the two out or inout parameters is assigned to the variables MyPar1 and MyPar2.

MyPort.getreply(MyProc1:{*, *} value *) -> value MyReturnValue param( - , MyPar2) sender MySender;
// The value of the first parameter is not considered for the further test execution and
// the address or component reference of the entity  from which the response has been received
// is stored in the variable MySender.

// The following examples describe some possibilities to assign out and inout parameter values
// to variables. The following signature is assumed for the procedure which has been called
signature MyProc2(in integer A, integer B, integer C, out integer D, integer E, inout integer F);
// Note: MyProc2 has no return value and no exceptions

MyPort.getreply(MyProc2:*) -> param( - , - , - , MyVarOut1, MyVarOut2, - , MyVarInout1);
// The in parameters D and E are assigned to the variables MyVarOut1 and MyVarOut2 the inout
// paramameter F is assigned to variable MyVarInout1.

MyPort.getreply(MyProc2:*) -> param(MyVarOut1:=D, MyVarOut2:=E, MyVarInout1:=F);
// Alternative notation for the value assignment of in and inout parameter to variables. Note,
// the names in the assignment list refer to the names used in the signature of MyProc2

MyPort.getreply(MyProc2:{ - , - , - , 3, *, *}) -> param(MyVarInout1:=F);
// Only the inout parameter value is needed for the further test case execution
```

22.3.5.1 Get any reply from any call

A **getreply** operation with no argument list for the signature matching criteria shall remove a **reply** message on the top of the incoming port queue (if any) if all other matching criteria are fulfilled. Parameters or return values of responses accepted by *GetAnyReply* shall not be assigned to a variable.

EXAMPLE:

```
MyPort.getreply;
// Removes the top response from MyPort.

MyPort.getreply from MyPartner;
// Removes the top response from CL1 if the responding party is an entity with the address
// or component reference stored in variable MyPartner.

MyPort.getreply -> sender MySenderVar;
// Removes the top response from CL1, but remembers the responding party by storing it
// in the variable MySenderVar
```

22.3.5.2 Get a reply on any port

To get a reply on any port use the **any** keyword. For example:

```
any port.getreply(Myproc)
```

22.3.6 The Catch operation

The **catch** operation is used to catch exceptions raised by a peer entity as a reaction to a procedure call. The **catch** operation shall only be used at procedure-based (or mixed) ports. The type of the caught exception shall be specified in the signature for the procedure which raised the exception.

```
MySyncPort.catch(MySignature, integer: MyVar);
// Specifiess the catch of an exception raised by a procedure with a signature Mysignature at
// port MySyncPort. The exception is an integer value which has the same value as the variable
// MyVar. The (optional) type identifier integer is not strictly necessary because the type is
// already given by the definition of MyVar. However, in complex and long test cases such a type
// identifier may be used to improve readability.

MySyncPort.catch(MySignature, MyVar);
// Is an alternative to the previous example.

MySyncPort.catch(MySignature, A<B);
 // Catches a Boolean exception of true or false depending on the outcome of A<B raised by a
// procedure with a signature MySignature at port MySyncPort.
```

The **catch** operation may be part of the accepting part of a call or be used to determine an alternative in an **alt** statement. If the **catch** operation is used in the accepting part of a **call** operation, the information about port name and signature reference to indicate the procedure which rose the exception is redundant, because this information follows from the **call** operation. However, for readability reasons (e.g., in case of complex **call** statements) this information shall be repeated.

Exceptions are specified as types and thus can be treated like messages e.g., templates can be used to distinguish between different values of the same exception type.

```
MySyncPort.catch(MySignature, MyTemplate:{5, MyVar});
// Catches an exception raised by a procedure with a signature Mysignature at port MySyncPort
// which fulfils the conditions defined by the template MyTemplate with actual parameters 5
// and MyVar.
```

The **catch** operation requires the port name, matching criteria for type and value, an optional **from** restriction (i.e., selection of communication partner) and an optional assignment of the matching exception and **sender** component to variables. For example:

```
MySyncPort.catch(MySignature, charstring:"Hello")from MyPartner;
 // Catches the IA5 string "Hello" raised by a procedure with a signature Mysignature at port
 // MySyncPort from an entity with an address or component reference stored in MyPartner.

MySyncPort.catch(MySignature, MyType:*) from MyPartner -> value MyVar;
 // Catches an exception with an arbitrary value of MyType (raised by a procedure with a
 // signature Mysignature at port MySyncPort from a component with a reference stored in
 // the variable MyPartner) which afterwards is assigned to the variable MyVar. MyVar has to be
 // of the type MyType.

MySyncPort.catch(MySignature, MyTemplate (5, MyVarOne)) -> value MyVarTwo sender MyPartner;
// Catches an exception raised by a procedure with a signature Mysignature with a value which
// fulfils the conditions defined by the template MyTemplate with actual parameters 5 and
// MyVarOne. Afterwards the exception is assigned to MyVarTwo. The address or reference of the
// sender entity is retrieved by the catch operation and assigned to MyPartner.
```

22.3.6.1 The Timeout exception

There is one special **timeout** exception which is caught by the **catch** operation. The **timeout** exception is an emergency exit for cases where a called procedure neither replies nor raises an exception within a predetermined time. For example:

```
MyPort.catch(timeout); // Catches a timeout exception.
```

Catching **timeout** exceptions shall be restricted to the exception handling part of a call. No further matching criteria (including a **from** part) and no assignment part is allowed for a **catch** operation that handles a **timeout** exception.

22.3.6.2 Catch any exception

A **catch** operation with no argument list allows any valid exception to be caught. The most general case is without using the **from** keyword and without an assignment part. This statement will also catch the **timeout** exception. For example:

```
MyPort.catch;

MyPort.catch from MyPartner;

MyPort.catch -> sender MySenderVar;
```

22.3.6.3 Catch on any port

To **catch** an exception on any port use the **any** keyword. For example:

```
any port.catch(timeout)
```

22.3.7 The Check operation

The **check** operation is a generic operation that allows read access to the top element of message-based and procedure-based *incoming* port queues without removing the top element from the queue. The **check** operation has to handle values of a certain type at message-based ports and to distinguish between calls to be accepted, exceptions to be caught and replies from previous calls at procedure-based ports.

The receiving operations **receive**, **getcall**, **getreply** and **catch** together with their matching and assignment parts, are used by the **check** operation to define the condition which has to be checked and to extract the value or values of its parameters if required.

```
MyAsyncPort.check(receive(integer: 5));
// Will check for an integer value of 5 as top message in the asynchronous port MyAsyncPort.

MyPort.check(getcall(MyProc:{5, MyVar}) from MyPartner);
// Will check for a a call of MyProc at MyCL (with the in or inout parameters 5 and MyVar) from
// a peer entity with the address or component reference stored in the variable MyPartner.

MyPort.check(getreply(MyProc:{5, MyVar} value 20));
// Checks for a reply from procedure MyProc at MyPort where the returned value is 20 and
// the values of the two out or inout parameters is 5 and the value of MyVar.

MySyncPort.check(catch(MySignature, MyTemplate (5, MyVar)));
// Checks for an exception raised by a procedure with a signature Mysignature at port MySyncPort
// which fulfils the conditions defined by the template MyTemplate with actual parameters 5
// and MyVar.
```

It is the *top* element of an incoming port queue that shall be checked (it is not possible to look *into* the queue). If the queue is empty the **check** operation fails. If the queue is not empty, a copy of the top element is made and the receiving operation specified in the **check** operation is performed on the copy. The **check** operation fails if the receiving function fails i.e., the matching criteria are not fulfilled. In this case the *copy* of the top element of the queue is discarded and test execution continues in the normal manner, i.e., the next alternative to the check operation is evaluated. The **check** operation is successful if the receiving function is successful.

Using the **check** operation in a wrong manner, e.g., check for an exception at a message-based port shall cause a test case error.

NOTE: In most cases the correct usage of the check operation can be checked statically, i.e., before compilation.

EXAMPLE:

```
MyPort.check(getreply(MyProc1:{*, MyVar} value *) -> value MyReturnValue param(MyPar1));
// In this example the returned value is assigned to variable MyReturnValue and the value of
// the first out or inout parameter is assigned to variable MyPar1.

MyPort.check(getcall(MyProc:{5, MyVar}) from MyPartner -> param (MyPar1Var, MyPar2Var));
// In this example both parameters of MyProc are considered to be inout parameters and that
// their values are assigned to MyPar1Var and MyPar2Var.

MyPort.check(getcall(MyProc:{5, MyVar}) -> sender MySenderVar);
// Will accept a call of MyProc at MyCL with the in or inout parameters 5 and MyVar. The calling
```

```
// party is retrieved and stored in MySenderVar.
```

22.3.7.1 The Check any operation

A **check** operation with no argument list allows to check whether something waits for processing in an incoming port queue. The *CheckAny* operation allows to distinguish between different senders (in case of one-to-many connections) by using a **from** clause and to retrieve the sender by using a shorthand assignment part with a sender clause.

EXAMPLE:

```
MyPort.check;

MyPort.check(from MyPartner);

MyPort.check(-> sender MySenderVar);
```

22.4 Controlling communication ports

TTCN-3 operations for controlling message-based, procedure-based and mixed ports are:

- **clear**: remove the contents of an incoming port queue;
- **start**: start listening at and give access to a port;
- **stop**: stop listening and disallow sending operations at a port.

22.4.1 The Clear port operation

The **clear** operation removes the contents of the *incoming* queue of the named port. If the port queue is already empty then this operation shall have no action.

```
MyPort.clear; // clears port MyPort
```

22.4.2 The Start port operation

If a port is defined as allowing receiving operations such as **receive**, **getcall** etc., the **start** operation clears the incoming queue of the named port and starts listening for traffic over the port. If the port is defined to allow sending operations then the operations such as **send**, **call**, **raise** etc., are also allowed to be performed at that port. For example:

```
MyPort.start; // starts MyPort
```

By default, all ports of a component shall be started when a component starts execution.

22.4.3 The Stop port operation

If a port is defined as allowing receiving operations such as **receive**, **getcall** the **start** operation **stop** operation causes listening at the named port to cease. If the port is defined to allow sending operations then **stop** port disallows the operations such as **send**, **call**, **raise** etc., to be performed. For example:

```
MyPort.stop;  // stops MyPort
```

22.5 Use of any and all with ports

The keywords **any** and **all** may be used with configuration operations as indicated in table 16.

Table 16: Any and All with ports

Operation	Allowed		Example
	any	all	
Receive communication operations (**receive, trigger, getcall, getreply, catch, check**)	yes		**any port.receive**
connect / map			
Start		yes	**all port.start**
Stop		yes	**all port.stop**
Clear		yes	**all port.clear**

23 Timer operations

TTCN-3 supports a number of timer operations. These operations may be used in test cases, functions and in module control.

Table 17: Overview of TTCN-3 timer operations

Timer operations	
Statement	**Associated keyword or symbol**
Start timer	**Start**
Stop timer	**Stop**
Read elapsed time	**Read**
Check if timer running	**running**
Timeout event	**timeout**

23.1 The Start timer operation

The **start** timer operation is used to indicate that a timer should start running. Timer values shall be of type **float**. For example:

```
MyTimer1.start;       // MyTimer1 is started with the default duration
MyTimer2.start(20E-3);// MyTimer2 is started with a duration of 20ms
```

The optional timer value parameter shall be used if no default duration is given, or if it is desired to override the default value specified in the timer declaration. When a timer duration is overridden, the new value applies only to the current instance of the timer, any later **start** operations for this timer, which do not specify a duration, shall use the default duration. The timer clock runs from the float value zero (0.0) up to maximum stated by the duration parameter.

23.2 The Stop timer operation

The **stop** operation is used to stop a running timer and to remove it from the list of running timers. A stopped timer becomes inactive and its elapsed time is set to the float value zero (0.0). If the timer name on the **stop** operation is **all**, then all running (i.e., active) timers are stopped. For example:

```
MyTimer1.stop;     // stops MyTimer1
all timer.stop;    // stops all running timers
```

Stopping an inactive timer is a valid operation, although it does not have any effect.

23.3 The Read timer operation

The **read** operation is used to retrieve the time that has elapsed since the specified timer was started and to store it into the specified variable. This variable shall be of type **float**. For example:

```
var float Myvar;
MyVar := MyTimer1.read; // assign to MyVar the time that has elapsed since MyTimer1 was started
```

Applying the **read** operation on an inactive timer will return the value zero.

23.4 The Running timer operation

The **running** operation is used to check whether or not a timer is running (i.e., that it has been started and has neither timed out nor been cancelled). The operation returns the value **true** if the timer is running, **false** otherwise. For example:

```
if (MyTimer1.running) { … }
```

23.5 The Timeout event

The **timeout** operation denotes the timeout of a previously started timer. The **timeout** operation can be used in alternatives together with **receive** and **getcall**, **getreply**, **catch** and **other timeout** operations.

EXAMPLE:

```
MyTimer1.timeout; // checks for the timeout of the previously started timer MyTimer1
```

The **any** keyword is used to indicate the **timeout** of any timer (rather than an explicitly named timer) started within the scope of the timeout. For example:

```
any timer.timeout; // checks for the timeout of any previously started timer
```

23.6 Use of any and all with timers

The keywords **any** and **all** may be used with timer operations as indicated in table 18.

Table 18: Any and All with Timers

Operation	Allowed		Example
	any	all	
start			
stop		yes	**All timer.stop**
read			
running	yes		**if (any timer.running) {…}**
timeout	yes		**Any timer.timeout**

24 Test verdict operations

Verdict operations allow to set and retrieve verdicts using the **get** and **set** operations respectively. These operations shall only be used in test cases and functions.

Table 19: Overview of TTCN-3 test verdict operations

Test verdict operations	
Statement	**Associated keyword or symbol**
Set local verdict	**Verdict.set**
Get local verdict	**Verdict.get**

Each test component of the active configuration shall maintain it's own local verdict. The local verdict is an object which is created for each test component at the time of its instantiation. It is used to track the individual verdict in each test component (i.e., in the MTC and in each and every PTC).

NOTE: Unlike TTCN-2 assigning a final verdict does not halt execution of the test component in which the behaviour is executing. If required, this shall be explicitly done using the **stop** statement.

24.1 Test case verdict

Additionally there is a global verdict that is updated when each test component (i.e., the MTC and each and every PTC) terminates execution. This verdict is not accessible to the **get** and **set** operations. The value of this verdict shall be returned by the test case when it terminates execution. If the returned verdict is not explicitly saved in the control part (e.g., assigned to a variable) then it is lost.

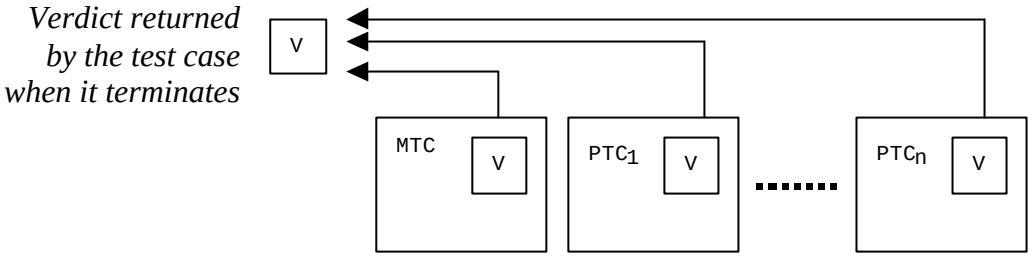


Figure 10: Illustration of the relationship between verdicts

NOTE: TTCN-3 does not specify the actual mechanisms that perform the updating of the local and test case verdicts. These mechanisms are implementation dependent.

24.2 Verdict values and overwriting rules

The verdict can have five different values: **pass**, **fail**, **inconc**, **none** and **error** i.e., the distinguished values of the **verdicttype** (see clause 6.1).

NOTE: **inconc** means an inconclusive verdict.

The **set** operation shall only be used with the values **pass**, **fail**, **inconc** and **none**. For example:

```
verdict.set(pass);
verdict.set(inconc);
```

The value of the local verdict may be retrieved using the **get** operation. For example:

```
MyResult := verdict.get; // Where MyResult is a variable of type verdicttype
```

When a test component is instantiated, its local verdict object is created and set to the value **none**.

When changing the value of the verdict (i.e., using the **set** operation) the effect of this change shall follow the overwriting rules listed in table 20. The test case verdict is implicitly set on the termination of a test component. The effect of this implicit operation shall also follow the overwriting rules listed in table 20.

Table 20: Overwriting rules for the verdict

Current Value of Verdict	New verdict assignment value			
	pass	inconc	fail	none
none	pass	inconc	fail	none
pass	pass	inconc	fail	pass
inconc	inconc	inconc	fail	inconc
fail	fail	Fail	fail	fail

EXAMPLE:

```
:
verdict.set(pass);// the local verdict is set to pass
:
verdict.set(fail);// until this line is executed which will result in the value
:                 // of the local verdict being overwritten to fail
:                 // When the ptc terminates the test case verdict is set to fail
```

24.2.1 Error verdict

The **error** verdict is special in that it is set by the test system to indicate that a test case (i.e., run-time) error has occurred. It shall not be set by the **set** operation. No other verdict value can override an **error** verdict. This means that an **error** verdict can only be a result of an **execute** test case operation.

25 SUT operations

In some testing situations where there may be no explicit interface to the SUT and it may be necessary that the SUT should be made to initiate certain actions (e.g., send a message to the test system).

This action may defined as a string, for example:

```
sut.action("Send MyTemplate on lower PCO");// Informal description of the SUT action
```

or as a reference to a template which specifies the structure of the message to be sent by the SUT, for example:

```
sut.action(MyTemplate); // This is equivalent to the TTCN-2 IMPLICIT SEND statement.
```

In both cases there is no specification of what is done to or by the SUT to trigger this action, only an informal specification of the required reaction itself.

SUT actions can be specified in test cases, functions, named alternatives and module control.

26 Module control part

Test cases are defined in the module definitions and executed in the module control. All variables, timers etc. (if any) defined in the control part of a module shall be passed into the test case by parameterization if they are to be used in the behaviour definition of that test case i.e., TTCN-3 does not support global variables of any kind.

At the start of each test case the test configuration shall be reset. This means that all **create**, **connect,** etc. operations that may have been performed in a previous test case are not 'visible' to the new test case.

26.1 Execution of test cases

A test case is called using an **execute** statement. As the result of the execution of a test case a test verdict of either **none**, **pass**, **inconclusive**, **fail** or **error** shall be returned and may be assigned to a variable for further processing.

Optionally, the **execute** statement allows supervision of a test case by means of a timer duration. If the test case does not end within this duration, the result of the test case execution shall be an error verdict and the test system shall terminate the test case.

EXAMPLE:

```
execute(MyTestCase1());                  // executes MyTestCase1, without storing the
                                         // returned test verdict and without time
                                         // supervision

MyVerdict := execute(MyTestCase2());     // executes MyTestCase2 and stores the resulting
                                         // verdict in variable MyVerdict
```

```
MyVerdict := execute(MyTestCase3(),5E-3);// executes MyTestCase3 and stores the resulting
                                         // verdict in variable MyVerdict. If the test case
                                         // does not terminate within 5ms, MyVerdict will
                                         // get the value 'error'
```

26.2 Termination of test cases

A test case terminates with the termination of the MTC. After the termination of the MTC all running parallel test components shall be terminated by the means of testing (i.e., test system).

NOTE 1: The concrete mechanism for stopping all PTCs is tool specific and therefore outside the scope of the present document.

The final verdict of a test case is calculated based on the final local verdicts of the different test components according to the rules defined in clause 24. The actual local verdict of a test component becomes its final local verdict when the test component terminates itself or is stopped by the means of testing (i.e., test system).

NOTE 2: To avoid race conditions for the calculation of test verdicts due to the delayed stopping of PTCs, the MTC should ensure that all PTCs have stopped (by means of the **done** statement) before it stops itself.

26.3 Controlling execution of test cases

Program statements, limited to those defined in table 11, may be used in the control part of a module to specify such things as the order in which the tests are to be executed or the number of times a test case may be run. For example:

```
module MyTestSuite
{   :
    control
    {   :
        // Do this test 10 times
        count:=0;
        while (count < 10)
        {   execute (MySimpleTestCase1());
            count := count+1;
        }
    }
}
```

If no programming statements are used then, by default, the test cases are executed in the sequential order in which they appear in the module control.

NOTE: This does not preclude the possibility that certain tools may wish to override this default ordering to allow a user or tool to select a different execution order.

Test cases return a single value of type **verdicttype** so it is possible to control the order of execution depending on the outcome of a test case. For example:

```
if (MySimpleTestCase() == pass) { log("Success!") }
```

26.4 Test case selection

Boolean expressions may be used to select and deselect which test cases are to be executed. This includes, of course, the use of functions that return a **boolean** value.

NOTE: This is equivalent to the TTCN-2 named test selection expressions.

EXAMPLE:

```
module MyTestSuite
{   :
    control
    {   :
        if (MySelectionExpression1())
        {   execute(MySimpleTestCase1());
            execute(MySimpleTestCase2());
            execute(MySimpleTestCase3());
```

```
        }
        if (MySelectionExpression2())
        {   execute(MySimpleTestCase4());
            execute(MySimpleTestCase5());
            execute(MySimpleTestCase6());
        }
        :
    }
}
```

Another way to execute test cases as a group is to collect them in a function and execute that function from the module control. For example:

```
:
function MyTestCaseGroup1()
{   execute(MySimpleTestCase1());
    execute(MySimpleTestCase2());
    execute(MySimpleTestCase3());
}
function MyTestCaseGroup2()
{   execute(MySimpleTestCase4());
    execute(MySimpleTestCase5());
    execute(MySimpleTestCase6());
}
:
control
{   if (MySelectionExpression1()) { MyTestCaseGroup1(); }
    if (MySelectionExpression1()) { MyTestCaseGroup2(); }
    :
}
:
```

26.5 Use of timers in control

Timers may be used to control execution of test cases. This may be done using an explicit timeout in the execute statement. For example:

```
MyReturnVal := execute (MyTestCase(), 7E-3); // variable of verdicttype
// Where the return verdict will be error if the TestCase does not complete execution
// within 7ms
```

The timer operations may also be used. For example:

```
// Example of the use of the running timer operation
while (T1.running or x<10)// Where T1 is a previously started timer
{   execute(MyTestCase());
    x := x+1;
}

// Example of the use of the start and timeout operations

timer T1 := 1;
:
execute(MyTestCase1());
T1.start;
T1.timeout;// Pause before executing the next test case
execute(MyTestCase2());
```

27 Specifying attributes

Attributes can be associated with TTCN-3 language elements by means of the **with** statement. The syntax for the argument of the **with** statement (i.e., the actual attributes) is simply defined as a free text string.

There are three kinds of attributes:

a) **display**: allows the specification of display attributes related to specific presentation formats;

b) **encode**: allows references to specific encoding rules;

c) **extension**: allows the specification of user-defined attributes.

27.1 Display attributes

All TTCN-3 language elements can have **display** attributes to specify how particular language elements should be displayed in, for example, a graphical format.

Special attribute strings related to the display attributes for the tabular (conformance) presentation format can be found in draft new Recommendation Z.141 [1].

Special attribute strings related to the display attributes for the graphical presentation format can be found in draft new Recommendation Z.142 [2].

Other **display** attributes may be defined by the user.

NOTE: Because user-defined attributes are not standardised the interpretation of these attributes between tools supplied by different vendors may differ or even not be supported.

27.2 Encoding attributes

Encoding rules define how a particular value, template etc. is encoded and transmitted, usually as a bit stream, over a communication **port**. TTCN-3 does not have a default encoding mechanism. This means that encoding rules or encoding directives are defined in some external manner to TTCN-3.

The **encode** attribute allows the association of some referenced encoding rule or encoding directive to be made to a TTCN-3 type definitions (and to a type definitions only).

Special attribute strings related to ASN.1 encoding attributes can be found in annex E.

The manner in which the actual encoding rules are defined (e.g., prose, functions etc.) is outside the scope of the present document. If no specific rules are referenced then encoding shall be a matter for individual implementation.

In most cases encoding attributes will be used in a hierarchical manner. The top-level is the entire module, the next level is a group of types and the lowest is an individual type:

a) **module**: encoding applies to all types defined in the module, including TTCN-3 base types;

b) **group**: encoding applies to a group of user-defined type definitions;

c) **type**: encoding applies to a single user-defined type;

d) **field:** encoding applies to a field in a **record** or **set** type;

EXAMPLE:

```
module MyTTCNmodule
{   :
    import type MyRecord from MySecondModule with {encode "MyRule 1"}
           // All instances of MyRecord will be encoded according to MyRule 1
    :
    type charstring MyType;// Normally encoded according to the global rule
    :
```

```
    group MyRecords
    {   :
        type record MyPDU1
        {
            integer    field1,    // field1 will be encoded according to Rule 3
            boolean    field2,    // field2 will be encoded according to Rule 3
            Mytype     field3     // field3 will be encoded according to Rule 2
        }
        with {encode (field1, field2) "Rule 3"}
        :
    }
    with {encode "Rule 2"}

}
with {encode "Global encoding rule"}
```

27.2.1 Invalid encodings

If it is desired to specify invalid encoding rules then these shall be specified in a referenceable source external to the module in the same way that valid encoding rules are referenced.

27.3 Extension attributes

All TTCN-3 language elements can have **extension** attributes specified by the user.

NOTE: Because user-defined attributes are not standardized the interpretation of these attributes between tools supplied by different vendors may differ or even not be supported.

27.4 Scope of attributes

A **with** statement always associates attributes to single language elements. It is also possible to associate attributes to a number of language elements by associating a **with** statement to the surrounding scope unit or **group** of language elements.

The **with** statement follows the scoping rules as defined in clause 5.4, i.e., a **with** statement that is placed inside the scope of another **with** statement shall override the outermost **with**. This shall also apply to the use of the **with** statement with groups. Care should be taken when the overwriting scheme is used in combination with references to single definitions. The general rule is that attributes shall be assigned and overwritten according to the order of their occurrence.

EXAMPLE:

```
// MyPDU1 will be displayed as PDU
type record MyPDU1 { … } with { display "PDU"}

// MyPDU2 will be displayed as PDU with the application specific extension attribute MyRule
type record MyPDU2 { … }
with
{
    display "PDU";
    extension "MyRule"
}

// The following group definition …
group MyPDUs {
    type record MyPDU3 { … }
    type record MyPDU4 { … }
}
with {display "PDU"}  // All types of group MyPDUs will be displayed as PDU

// is identical to
group MyPDUs {
    type record MyPDU3 { … } with { display "PDU"}
    type record MyPDU4 { … } with { display "PDU"}
}

// Example of the use of the overwriting scheme of the with statement
group MyPDUs
{
```

```
        type record MyPDU1 { … }
        type record MyPDU2 { … }

        group MySpecialPDUs
        {
            type record MyPDU3 { … }
            type record MyPDU4 { … }
        }
        with {extension "MySpecialRule"} // MyPDU3 and MyPDU4 will have the application
                                         // specific extension attribute MySpecialRule
    }
    with
    {
        display "PDU";     // All types of group MPDUs will be displayed as PDU and
        extension "MyRule";// (if not overwritten) have the extension attribute MyRule
    }

    // is identical to …
    group MyPDUs
    {
        type record MyPDU1 { … } with {display "PDU"; extension "MyRule" }
        type record MyPDU2 { … } with {display "PDU"; extension "MyRule" }
        group MySpecialPDUs {
            type record MyPDU3 { … } with {display "PDU"; extension "MySpecialRule" }
            type record MyPDU4 { … } with {display "PDU"; extension "MySpecialRule" }
        }
    }
```

27.5 Overwriting rules for attributes

An attribute definition in a lower scope unit will override a general attribute definition in a higher scope. For example:

```
type record MyRecordA
{
    :
} with {encode "RuleA”}

// In the following, MyRecordA is encoded according to RuleA and not according to RuleB
type record MyRecordB
{   :
    field  MyRecordA
} with {encode "RuleB”}
```

An attribute definition in a lower scope can be overwritten in a higher scope by using the **`override`** directive. For example:

```
type record MyRecordA
{
    :
} with {encode "RuleA”}

// In the following, MyRecordA is encoded according to RuleB
type record MyRecordB
{   :
    fieldA MyRecordA
} with {encode override "RuleB”}
```

The override directive forces all contained types at all lower scopes to be forced to the specified attribute.

27.6 Changing attributes of imported language elements

In general, a language element is imported together with its attributes. In some cases these attributes may have to be changed when importing the language element e.g., a type may be displayed in one module as ASP, then it is imported by another module where it should be displayed as PDU. For such cases it is allowed to change attributes on the import statement.

EXAMPLE:

```
Import type MyType from MyModule with {display "ASP"} // MyType will be displayed as ASP

Import group MyGroup from MyModule with
{
    display "ASP";               // By default all types will be displayed as ASP.
    extension "MyRule"
```

```
}
```

Annex A (normative):
BNF and static semantics

A.1 TTCN-3 BNF

This annex defines the syntax of TTCN-3 using extended BNF (henceforth just called BNF).

A.1.1 Conventions for the syntax description

Table A.1 defines the metanotation used to specify the extended BNF grammar for TTCN-3:

Table A.1: The Syntactic Metanotation

::=	is defined to be
abc xyz	abc followed by xyz
|	alternative
[abc]	0 or 1 instances of abc
{abc}	0 or more instances of abc
{abc}+	1 or more instances of abc
(...)	textual grouping
Abc	the non-terminal symbol abc
abc	a terminal symbol abc
"abc"	a terminal symbol abc

A.1.2 Statement terminator symbols

In general all TTCN-3 language constructs (i.e., definitions, declarations, statements and operations) are terminated with a semi-colon (;). The semi-colon is optional if the language construct ends with a right-hand curly brace (}) or the following symbol is a right-hand curly brace (}), i.e., the language construct is the last statement in a statement block.

A.1.3 Identifiers

TTCN-3 identifiers are case sensitive and may only contain lowercase letters (a-z) uppercase letters (A-Z) and numeric digits (0-9). Use of the underscore (_) symbol is also allowed. An identifier shall begin with a letter (i.e., not a number and not an underscore).

A.1.4 Comments

Comments written in free text may appear anywhere in a TTCN-3 specification.

Block comments shall be opened by the symbol pair /* and closed by the symbol pair */. For example:

```
/* This is a block comment
spread over two lines */
```

Block comments shall not be nested.

```
/* This is not /* a legal */ comment */
```

Line comments shall be opened by the symbol pair // and closed by a <*newline*>. For example:

```
// This is a line comment
// spread over two lines
```

Line comments may follow TTCN-3 program statements but they shall not be embedded in a statement. For example:

```
// The following is not legal
const // This is MyConst integer MyConst := 1;

// The following is legal
const integer MyConst := 1; // This is MyConst
```

A.1.5 TTCN-3 terminals

TTCN-3 terminal symbols and reserved words are listed in table A.2 and table A.3.

Table A.2: List of TTCN-3 special terminal symbols

Begin/end block symbols	**{ }**
Begin/end list symbols	**( )**
Alternative symbols	**[ ]**
To symbol (in a range)	**..**
Line comments and Block comments	**/* */ //**
Line/statement terminator symbol	**;**
Arithmetic operator symbols	**+ / -**
String concatenation operator symbol	**&**
Equivalence operator symbols	**!= == >= <=**
String enclosure symbols	**" '**
Wildcard/matching symbols	**? ***
Assignment symbol	**:=**
Communication operation assignment	**->**
Bitstring, hexstring and Octetstring values	**B H O**
Float exponent	**E**

The following lists the special identifiers reserved for the predefined functions defined in annex D:

int2char, char2int, int2unichar, unichar2int, bit2int, hex2int, int2bit, int2hex, int2oct, int2str, oct2int, str2int, lengthof, sizeof, ischosen, ispresent

Table A.3: List of TTCN-3 terminals which are reserved words

action	**fail**	**named**	**self**
activate	**false**	**none**	**send**
address	**float**	**nonrecursive**	**sender**
all	**for**	**not**	**set**
alt	**from**	**not4b**	**signature**
and	**function**	**nowait**	**start**
and4b		**null**	**stop**
any	**get**		**sut**
	getcall	**objid**	**system**
bitstring	**getreply**	**octetstring**	
boolean	**goto**	**of**	**template**
	group	**omit**	**testcase**
call		**on**	**timeout**
catch	**hexstring**	**optional**	**timer**
char		**or**	**to**
charstring	**if**	**or4b**	**trigger**
check	**ifpresent**	**out**	**true**
clear	**import**	**override**	**type**
complement	**in**		
component	**inconc**	**param**	**union**
connect	**infinity**	**pass**	**universal**
const	**inout**	**pattern**	**unmap**
control	**integer**	**port**	
create	**interleave**	**procedure**	**value**
			valueof
deactivate	**label**	**raise**	**var**
disconnect	**language**	**read**	**verdict**
display	**length**	**receive**	**verdicttype**
do	**log**	**record**	
done		**rem**	**while**
	map	**repeat**	**with**
else	**match**	**reply**	
encode	**message**	**return**	**xor**
enumerated	**mixed**	**running**	**xor4b**
error	**mod**	**runs**	
exception	**modifies**		
execute	**module**		
expand	**mtc**		
extension			
external			

The TTCN-3 terminals listed in table A.3 shall not be used as identifiers in a TTCN-3 module. These terminals shall be written in all lowercase letters.

A.1.6 TTCN-3 syntax BNF productions

A.1.6.1 TTCN Module

```
1. TTCN3Module ::= TTCN3ModuleKeyword TTCN3ModuleId [ModuleParList]
                   BeginChar
                   [ModuleDefinitionsPart]
                   [ModuleControlPart]
                   EndChar
                   [WithStatement] [SemiColon]
2. TTCN3ModuleKeyword ::= "module"
3. TTCN3ModuleId ::= ModuleIdentifier [DefinitiveIdentifier]
4. ModuleIdentifier ::= Identifier
5. DefinitiveIdentifier ::= Dot ObjectIdentifierKeyword "{" DefinitiveObjIdComponentList "}"
6. DefinitiveObjIdComponentList ::= {DefinitiveObjIdComponent}+
7. DefinitiveObjIdComponent ::= NameForm |
                   DefinitiveNumberForm |
                   DefinitiveNameAndNumberForm
8. DefinitiveNumberForm ::= Number
9. DefinitiveNameAndNumberForm ::= Identifier  "(" DefinitiveNumberForm ")"
```

```
10. ModuleParList ::= "(" ModulePar {"," ModulePar} ")"
11. ModulePar ::= [InParKeyword] ModuleParType ModuleParIdentifier
                  [AssignmentChar ConstantExpression]
/* STATIC SEMANTICS - The Value of the ConstantExpression shall be of the same type as the stated
type for the Parameter */
12. ModuleParType ::= Type
13. ModuleParIdentifier ::= Identifier
```

A.1.6.2 Module Definitions Part

```
14. ModuleDefinitionsPart ::= ModuleDefinitionsList
15. ModuleDefinitionsList ::= {ModuleDefinition [SemiColon]}+
16. ModuleDefinition ::= (TypeDef |
                  ConstDef |
                  TemplateDef |
                  FunctionDef |
                  SignatureDef |
                  TestcaseDef |
                  NamedAltDef |
                  ImportDef |
                  GroupDef |
                  ExtFunctionDef |
                  ExtConstDef) [WithStatement]
```

A.1.6.2.1 Typedef Definitions

```
17. TypeDef ::= TypeDefKeyword TypeDefBody
18. TypeDefBody ::= StructuredTypeDef | SubTypeDef
19. TypeDefKeyword ::= "type"
20. StructuredTypeDef ::= RecordDef | UnionDef | SetDef | RecordOfDef | SetOfDef | EnumDef |
                  PortDef | ComponentDef
21. RecordDef ::= RecordKeyword StructDefBody
22. RecordKeyword ::= "record"
23. StructDefBody ::= (StructTypeIdentifier  [StructDefFormalParList] | AddressKeyword)
                  BeginChar
                  [StructFieldDef {"," StructFieldDef}]
                  EndChar
24. StructTypeIdentifier ::= Identifier
25. StructDefFormalParList ::= "(" StructDefFormalPar {"," StructDefFormalPar} ")"
26. StructDefFormalPar ::= FormalValuePar | FormalTypePar
/* STATIC SEMANTICS - FormalValuePar shall resolve to an in parameter */
27. StructFieldDef ::= Type StructFieldIdentifier [ArrayDef] [SubTypeSpec] [OptionalKeyword]
28. StructFieldIdentifier ::= Identifier
29. OptionalKeyword ::= "optional"
30. UnionDef ::= UnionKeyword UnionDefBody
31. UnionKeyword ::= "union"
32. UnionDefBody ::= (StructTypeIdentifier [StructDefFormalParList] | AddressKeyword)
                  BeginChar
                  UnionFieldDef {"," UnionFieldDef}
                  EndChar
33. UnionFieldDef ::= Type StructFieldIdentifier [ArrayDef] [SubTypeSpec]
34. SetDef ::= SetKeyword StructDefBody
35. SetKeyword ::= "set"
36. RecordOfDef ::= RecordKeyword OfKeyword [StringLength]  StructOfDefBody
37. OfKeyword ::= "of"
38. StructOfDefBody ::= Type (StructTypeIdentifier | AddressKeyword) [SubTypeSpec]
39. SetOfDef ::= SetKeyword OfKeyword [StringLength]  StructOfDefBody
40. EnumDef ::= EnumKeyword (EnumTypeIdentifier | AddressKeyword)
                  BeginChar
                  NamedValueList
                  EndChar
41. EnumKeyword ::= "enumerated"
42. EnumTypeIdentifier ::= Identifier
43. NamedValueList ::= NamedValue {"," NamedValue}
44. NamedValue ::= NamedValueIdentifier ["(" Number ")"]
45. NamedValueIdentifier ::= Identifier
46. SubTypeDef ::= Type (SubTypeIdentifier | AddressKeyword)  [ArrayDef] [SubTypeSpec]
47. SubTypeIdentifier ::= Identifier
48. SubTypeSpec ::= AllowedValues | StringLength
/* STATIC SEMANTICS - The values shall be of the same type as the field being subtyped */
49. AllowedValues ::= "(" ValueOrRange {"," ValueOrRange} ")"
50. ValueOrRange ::= IntegerRangeDef | SingleConstExpression
/* STATIC SEMANTICS - IntergerRangeDef production shall only be used with integer based types */
51. IntegerRangeDef ::= LowerBound ".." UpperBound
52. StringLength ::= LengthKeyword "(" SingleConstExpression [".." UpperBound] ")"
/* STATIC SEMANTICS - StringLength shall only be used with String types or to limit set of and
record of */
53. LengthKeyword ::= "length"
```

```
54. PortType ::= [GlobalModuleId Dot] PortTypeIdentifier
55. PortDef ::= PortKeyword PortDefBody
56. PortDefBody ::= PortTypeIdentifier PortDefAttribs
57. PortKeyword ::= "port"
58. PortTypeIdentifier ::= Identifier
59. PortDefAttribs ::= MessageAttribs | ProcedureAttribs | MixedAttribs
60. MessageAttribs ::= MessageKeyword
                BeginChar
                {MessageList  [SemiColon]}+
                EndChar
61. MessageList ::= Direction AllOrTypeList
62. Direction ::= InParKeyword | OutParKeyword | InOutParKeyword
63. MessageKeyword ::= "message"
64. AllOrTypeList ::= AllKeyword | TypeList
65. AllKeyword ::= "all"
66. TypeList ::= Type {"," Type}
67. ProcedureAttribs ::= ProcedureKeyword
                BeginChar
                {ProcedureList  [SemiColon]}+
                EndChar
68. ProcedureKeyword ::= "procedure"
69. ProcedureList ::= Direction AllOrSignatureList
70. AllOrSignatureList ::= AllKeyword | SignatureList
71. SignatureList ::= Signature {"," Signature}
72. MixedAttribs ::= MixedKeyword
                BeginChar
                {MixedList  [SemiColon]}+
                EndChar
73. MixedKeyword ::= "mixed"
74. MixedList ::= Direction ProcOrTypeList
75. ProcOrTypeList ::= AllKeyword | (ProcOrType {"," ProcOrType})
76. ProcOrType ::= Signature | Type
77. ComponentDef ::= ComponentKeyword ComponentTypeIdentifier
                BeginChar
                [ComponentDefList]
                EndChar
78. ComponentKeyword ::= "component"
79. ComponentType ::= [GlobalModuleId Dot] ComponentTypeIdentifier
80. ComponentTypeIdentifier ::= Identifier
81. ComponentDefList ::= {ComponentElementDef [SemiColon]}+
82. ComponentElementDef ::= PortInstance | VarInstance | TimerInstance | ConstDef
83. PortInstance ::= PortKeyword PortType PortElement {"," PortElement}
84. PortElement ::= PortIdentifier [ArrayDef]
85. PortIdentifier ::= Identifier
```

A.1.6.2.2 Constant Definitions

```
86. ConstDef ::= ConstKeyword Type ConstList
87. ConstList ::= SingleConstDef  {"," SingleConstDef}
88. SingleConstDef ::= ConstIdentifier [ArrayDef] AssignmentChar ConstantExpression
/* STATIC SEMANTICS - The Value of the ConstantExpression shall be of the same type as the stated
type for the constant */
89. ConstKeyword ::= "const"
90. ConstIdentifier ::= Identifier
```

A.1.6.2.3 Template Definitions

```
91. TemplateDef ::= TemplateKeyword BaseTemplate  [DerivedDef]
                AssignmentChar TemplateBody
92. BaseTemplate ::= (Type | Signature) TemplateIdentifier  ["(" TemplateFormalParList ")"]
93. TemplateKeyword ::= "template"
94. TemplateIdentifier ::= Identifier
95. DerivedDef ::= ModifiesKeyword TemplateRef
96. ModifiesKeyword ::= "modifies"
97. TemplateFormalParList ::= TemplateFormalPar {"," TemplateFormalPar}
98. TemplateFormalPar ::= FormalValuePar |
                FormalTemplatePar
/* STATIC SEMANTICS - FormalValuePar shall resolve to an in parameter */
99. TemplateBody ::= SimpleSpec | FieldSpecList |
                ArrayValueOrAttrib
100. SimpleSpec ::= SingleValueOrAttrib
/* STATIC SEMANTICS - SimpleSpec shall not be used for constructed types */
101. FieldSpecList ::= "{"[FieldSpec {"," FieldSpec}] "}"
102. FieldSpec ::= FieldReference AssignmentChar TemplateBody
103. FieldReference ::= RecordRef | ArrayOrBitRef | ParRef
104. RecordRef ::= StructFieldIdentifier
105. ParRef ::= SignatureParIdentifier
```

```
/* OPERATIONAL SEMANTICS - SignatureParIdentifier shall be a formal parameter Identifier from the
associated signature definition */
106. SignatureParIdentifier ::= ValueParIdentifier
107. ArrayOrBitRef ::= "[" FieldOrBitNumber "]"
/* STATIC SEMANTICS - ArrayRef shall be optionally used for array types and ASN.1 SET OF and
SEQUENCE OF and TTCN record, record of, set and set Identifier of. The same notation can be used for
a Bit reference inside an ASN.1 or TTCN bitstring type */
108. FieldOrBitNumber ::= SingleExpression
/* STATIC SEMANTICS - SingleExpression will resolve to a value of integer type */
109. SingleValueOrAttrib ::= MatchingSymbol [ExtraMatchingAttributes] |
               SingleExpression [ExtraMatchingAttributes] |
               TemplateRefWithParList
/* STATIC SEMANTIC - VariableIdentifier (accessed via singleExpression) may only be used in inline
template definitions to reference variables in the current scope */
110. ArrayValueOrAttrib ::= "{" ArrayElementSpecList "}"
111. ArrayElementSpecList ::= ArrayElementSpec {"," ArrayElementSpec}
112. ArrayElementSpec ::= NotUsedSymbol | TemplateBody
113. NotUsedSymbol ::= Dash
114. MatchingSymbol ::= Complement | Omit | AnyValue | AnyOrOmit | ValueList |
               IntegerRange | BitStringMatch | HexStringMatch |
               OctetStringMatch | CharStringMatch
115. ExtraMatchingAttributes ::= LengthMatch | IfPresentMatch
116. BitStringMatch ::= "'" {BinOrMatch} "'" B
117. BinOrMatch ::= Bin | AnyValue | AnyOrOmit
118. HexStringMatch ::= "'" {HexOrMatch} "'" H
119. HexOrMatch ::= Hex | AnyValue | AnyOrOmit
120. OctetStringMatch ::= "'" {OctOrMatch} "'" O
121. OctOrMatch ::= Oct | AnyValue | AnyOrOmit
122. CharStringMatch ::= PatternKeyword CharStringPattern {StringOp  CharStringPattern}
/* STATIC SEMANTICS - all CharStringPatterns shall resolve to the same character or character string
type */
123. CharStringPattern ::= CharStringValue | TemplateRefWithParList
124. PatternKeyword ::= "pattern"
125. Complement ::= ComplementKeyword (SingleConstExpression | ValueList)
126. ComplementKeyword ::= "complement"
127. Omit ::= OmitKeyword
128. OmitKeyword ::= "omit"
129. AnyValue ::= "?"
130. AnyOrOmit ::= "*"
131. ValueList ::= "(" SingleConstExpression {"," SingleConstExpression}+ ")"
132. LengthMatch ::= StringLength
133. IfPresentMatch ::= IfPresentKeyword
134. IfPresentKeyword ::= "ifpresent"
135. IntegerRange ::= "(" LowerBound ".." UpperBound ")"
136. LowerBound ::= SingleConstExpression | Minus InfinityKeyword
137. UpperBound ::= SingleConstExpression | InfinityKeyword
138. InfinityKeyword ::= "infinity"
139. TemplateInstance ::= InLineTemplate
140. TemplateRefWithParList ::= [GlobalModuleId Dot] TemplateIdentifier [TemplateActualParList] |
               TemplateParIdentifier
141. TemplateRef ::= [GlobalModuleId Dot] TemplateIdentifier | TemplateParIdentifier
142. InLineTemplate ::= [(Type | Signature) Colon] [DerivedDef AssignmentChar] TemplateBody
/* STATIC SEMANTICS - The type field may only be omitted when the type is implicitly unambigous */
143. TemplateActualParList ::= "(" TemplateActualPar {"," TemplateActualPar} ")"
144. TemplateActualPar ::= TemplateInstance
/* STATIC SEMANTICS - When the corresponding formal parameter is not of template type the
TemplateInstance production shall resolve to one or more SingleExpressions */
145. TemplateOps ::= MatchOp | ValueofOp
146. MatchOp ::= MatchKeyword "(" Expression "," TemplateInstance")"
/* STATIC SEMANTICS - The type of the value returned by the expression shall be the same as the
template type and each field of the template shall resolve to a single value */
147. MatchKeyword ::= "match"
148. ValueofOp ::= ValueofKeyword "(" TemplateInstance")"
149. ValueofKeyword ::= "valueof"
```

A.1.6.2.4 Function Definitions

```
150. FunctionDef ::= FunctionKeyword FunctionIdentifier
               "("[FunctionFormalParList] ")" [RunsOnSpec] [ReturnType]
               BeginChar
               FunctionBody
               EndChar
151. FunctionKeyword ::= "function"
152. FunctionIdentifier ::= Identifier
153. FunctionFormalParList ::= FunctionFormalPar {"," FunctionFormalPar}
154. FunctionFormalPar ::= FormalValuePar |
               FormalTimerPar |
               FormalTemplatePar |
```

```
              FormalPortPar
155. ReturnType ::= ReturnKeyword Type
156. ReturnKeyword ::= "return"
157. RunsOnSpec ::= RunsKeyword OnKeyword (ComponentType | MTCKeyword)
158. RunsKeyword ::= "runs"
159. OnKeyword ::= "on"
160. MTCKeyword ::= "mtc"
161. FunctionBody ::= [FunctionStatementOrDefList]
162. FunctionStatementOrDefList ::= {FunctionStatementOrDef [SemiColon]}+
163. FunctionStatementOrDef ::= FunctionLocalDef |
              FunctionLocalInst |
              FunctionStatement
164. FunctionLocalInst ::= VarInstance |
              TimerInstance
165. FunctionLocalDef ::= ConstDef
166. FunctionStatement ::= ConfigurationStatements |
              TimerStatements |
              CommunicationStatements |
              BasicStatements |
              BehaviourStatements |
              VerdictStatements |
              SUTStatements
167. FunctionInstance ::= FunctionRef "(" [FunctionActualParList] ")"
168. FunctionRef ::= [GlobalModuleId Dot] FunctionIdentifier
169. FunctionActualParList ::= FunctionActualPar {"," FunctionActualPar}
170. FunctionActualPar ::= TimerRef |
              TemplateInstance |
              Port |
              ComponentRef
/* STATIC SEMANTICS - When the corresponding formal parameter is not of template type the
TemplateInstance production shall resolve to one or more SingleExpressions i.e. eqivelent to the
Expression production */
```

A.1.6.2.5 Signature Definitions

```
171. SignatureDef ::= SignatureKeyword SignatureIdentifier
              "("[SignatureFormalParList] ")" [ReturnType]
              [ExceptionSpec]
172. SignatureKeyword ::= "signature"
173. SignatureIdentifier ::= Identifier
174. SignatureFormalParList ::= SignatureFormalPar {"," SignatureFormalPar}
175. SignatureFormalPar ::= FormalValuePar
176. ExceptionSpec ::= ExceptionKeyword "(" ExceptionTypeList ")"
177. ExceptionKeyword ::= "exception"
178. ExceptionTypeList ::= Type {"," Type}
179. Signature ::= [GlobalModuleId Dot] SignatureIdentifier
```

A.1.6.2.6 Testcase Definitions

```
180. TestcaseDef ::= TestcaseKeyword TestcaseIdentifier
              "("[TestcaseFormalParList] ")" ConfigSpec
              BeginChar
              FunctionBody
              EndChar
181. TestcaseKeyword ::= "testcase"
182. TestcaseIdentifier ::= Identifier
183. TestcaseFormalParList ::= TestcaseFormalPar {"," TestcaseFormalPar}
184. TestcaseFormalPar ::= FormalValuePar |
              FormalTemplatePar
185. ConfigSpec ::= RunsOnSpec [SystemSpec]
186. SystemSpec ::= SystemKeyword ComponentType
187. SystemKeyword ::= "system"
188. TestcaseInstance ::= ExecuteKeyword "(" TestcaseRef "(" [TestcaseActualParList] ")" ["," 
TimerValue]                 ")"
189. ExecuteKeyword ::= "execute"
190. TestcaseRef ::= [GlobalModuleId Dot] TestcaseIdentifier
191. TestcaseActualParList ::= TestcaseActualPar {"," TestcaseActualPar}
192. TestcaseActualPar ::=
              TemplateInstance
/* STATIC SEMANTICS - When the corresponding formal parameter is not of template type the
TemplateInstance production shall resolve to one or more SingleExpressions i.e. equivalent to the
Expression production */
```

A.1.6.2.7 NamedAlt Definitions

```
193. NamedAltDef ::= NamedKeyword AltKeyword NamedAltIdentifier
              "("  [NamedAltFormalParList] ")"
              BeginChar
```

```
              AltGuardList                    EndChar
194. NamedKeyword ::= "named"
195. NamedAltIdentifier ::= Identifier
196. NamedAltFormalParList ::= NamedAltFormalPar {"," NamedAltFormalPar}
197. NamedAltFormalPar ::= FormalValuePar |
              FormalTimerPar |
              FormalTemplatePar |
              FormalPortPar
198. NamedAltInstance ::= NamedAltRef  "(" [NamedAltActualParList]")"
199. NamedAltRef ::= [GlobalModuleId Dot] NamedAltIdentifier
200. NamedAltActualParList ::= NamedAltActualPar {"," NamedAltActualPar}
201. NamedAltActualPar ::=
              TimerRef |
              TemplateInstance |
              Port |
              ComponentRef
/* STATIC SEMANTICS - When the corresponding formal parameter is not of template type the
TemplateInstance production shall resolve to one or more SingleExpressions i.e. equivalent to the
Expression production */
```

A.1.6.2.8 Import Definitions

```
202. ImportDef ::= ImportKeyword ImportSpec
203. ImportKeyword ::= "import"
204. ImportSpec ::= ImportAllSpec |
              ImportGroupSpec |
              ImportTypeDefSpec |
              ImportTemplateSpec |
              ImportConstSpec |
              ImportTestcaseSpec |
              ImportNamedAltSpec |
              ImportFunctionSpec |
              ImportSignatureSpec
205. ImportAllSpec ::= AllKeyword [DefKeyword] ImportFromSpec
206. ImportFromSpec ::= FromKeyword ModuleId [NonRecursiveKeyword]
207. ModuleId ::= GlobalModuleId [LanguageSpec]
/* STATIC SEMANTICS - LanguageSpec may only be omitted if the referenced module contains TTCN-3
notation */
208. LanguageKeyword ::= "language"
209. LanguageSpec ::= LanguageKeyword  FreeText
210. GlobalModuleId ::= ModuleIdentifier [Dot ObjectIdentifierValue]
211. DefKeyword ::= TypeDefKeyword |
              ConstKeyword |
              TemplateKeyword |
              TestcaseKeyword |
              FunctionKeyword |
              SignatureKeyword |
              NamedKeyword AltKeyword
212. NonRecursiveKeyword ::= "nonrecursive"
213. ImportGroupSpec ::= GroupKeyword GroupIdentifier {"," GroupIdentifier} ImportFromSpec
214. ImportTypeDefSpec ::= TypeDefKeyword TypeDefIdentifier {"," TypeDefIdentifier} ImportFromSpec
215. TypeDefIdentifier ::= StructTypeIdentifier |
              EnumTypeIdentifier |
              PortTypeIdentifier |
              ComponentTypeIdentifier |
              SubTypeIdentifier
216. ImportTemplateSpec ::= TemplateKeyword TemplateIdentifier {"," TemplateIdentifier}
ImportFromSpec
217. ImportConstSpec ::= ConstKeyword ConstIdentifier {"," ConstIdentifier} ImportFromSpec
218. ImportTestcaseSpec ::= TestcaseKeyword TestcaseIdentifier {"," TestcaseIdentifier}
ImportFromSpec
219. ImportFunctionSpec ::= FunctionKeyword FunctionIdentifier {"," FunctionIdentifier}
ImportFromSpec
220. ImportSignatureSpec ::= SignatureKeyword SignatureIdentifier {"," SignatureIdentifier}
ImportFromSpec
221. ImportNamedAltSpec ::= NamedKeyword AltKeyword NamedAltIdentifier {"," NamedAltIdentifier}
ImportFromSpec
```

A.1.6.2.9 Group Definitions

```
222. GroupDef ::= GroupKeyword GroupIdentifier
              BeginChar
              [ModuleDefinitionsPart]
              EndGroupChar
223. GroupKeyword ::= "group"
224. EndGroupChar ::= "}"
225. GroupIdentifier ::= Identifier
```

A.1.6.2.10 External Function Definitions

```
226. ExtFunctionDef ::= ExtKeyword FunctionKeyword ExtFunctionIdentifier
              "("[FunctionFormalParList] ")" [ReturnType]
227. ExtKeyword ::= "external"
228. ExtFunctionIdentifier ::= Identifier
```

A.1.6.2.11 External Constant Definitions

```
229. ExtConstDef ::= ExtKeyword ConstKeyword Type ExtConstIdentifier
230. ExtConstIdentifier ::= Identifier
```

A.1.6.3 Control Part

```
231. ModuleControlPart ::= ControlKeyword
              BeginChar
              ModuleControlBody
              EndChar
              [WithStatement] [SemiColon]
232. ControlKeyword ::= "control"
233. ModuleControlBody ::= [ControlStatementOrDefList]
234. ControlStatementOrDefList ::= {ControlStatementOrDef [SemiColon]}+
235. ControlStatementOrDef ::= FunctionLocalInst |
              ControlStatement |
              FunctionLocalDef
236. ControlStatement ::= TimerStatements |
              BasicStatements |
              BehaviourStatements |
              SUTStatements
```

A.1.6.3.1 Variable Instantiation

```
237. VarInstance ::= VarKeyword Type VarList
238. VarList ::= SingleVarInstance {"," SingleVarInstance}
239. SingleVarInstance ::= VarIdentifier [ArrayDef] [AssignmentChar VarInitialValue]
240. VarInitialValue ::= Expression
241. VarKeyword ::= "var"
242. VarIdentifier ::= Identifier
243. VariableRef ::= (VarIdentifier | ValueParIdentifier) [ExtendedFieldReference]
```

A.1.6.3.2 Timer Instantiation

```
244. TimerInstance ::= TimerKeyword TimerIdentifier [ArrayDef]
              [AssignmentChar TimerValue]
245. TimerKeyword ::= "timer"
246. TimerIdentifier ::= Identifier
247. TimerValue ::= SingleExpression
/* STATIC SEMANTICS - SingleExpression shall resolve to a value of type float */
248. TimerRef ::= TimerIdentifier [ArrayOrBitRef]|
              TimerParIdentifier [ArrayOrBitRef]
```

A.1.6.3.3 Component Operations

```
249. ConfigurationStatements ::= ConnectStatement |
              MapStatement |
              DisconnectStatement |
              UnmapStatement |
              DoneStatement |
              StartTCStatement |
              StopTCStatement
250. ConfigurationOps ::= CreateOp | SelfOp | SystemOp | MTCOp | RunningOp
251. CreateOp ::= ComponentType Dot CreateKeyword
252. SystemOp ::= "system"
253. SelfOp ::= "self"
254. MTCOp ::= MTCKeyword
255. DoneStatement ::= ComponentId Dot DoneKeyword
256. ComponentId ::= ComponentIdentifier | (AnyKeyword | AllKeyword) ComponentKeyword
257. DoneKeyword ::= "done"
258. RunningOp ::= ComponentId Dot RunningKeyword
259. RunningKeyword ::= "running"
260. CreateKeyword ::= "create"
261. ConnectStatement ::= ConnectKeyword PortSpec
262. ConnectKeyword ::= "connect"
263. PortSpec ::= "(" PortRef "," PortRef ")"
264. PortRef ::= ComponentRef Colon Port
265. ComponentRef ::= ComponentIdentifier | SystemOp | SelfOp | MTCOp
266. DisconnectStatement ::= DisconnectKeyword PortSpec
```

```
267. DisconnectKeyword ::= "disconnect"
268. MapStatement ::= MapKeyword PortSpec
269. MapKeyword ::= "map"
270. UnmapStatement ::= UnmapKeyword PortSpec
271. UnmapKeyword ::= "unmap"
272. StartTCStatement ::= ComponentIdentifier Dot StartKeyword "(" FunctionInstance ")"
/* STATIC SEMANTICS - The Function instance may only have in parameters */
273. StartKeyword ::= "start"
274. StopTCStatement ::= StopKeyword
275. ComponentIdentifier ::= VariableRef | FunctionInstance
/* STATIC SEMANTICS - The variable associated with VariableRef or the  Return type associated with
FunctionInstance shall be of component type */
```

A.1.6.3.4 Port Operations

```
276. Port ::= (PortIdentifier | PortParIdentifier) [ArrayOrBitRef]
277. CommunicationStatements ::= SendStatement | CallStatement | ReplyStatement | RaiseStatement |
               ReceiveStatement | TriggerStatement | GetCallStatement |
               GetReplyStatement | CatchStatement | CheckStatement |
               ClearStatement | StartStatement | StopStatement
278. SendStatement ::= Port Dot PortSendOp
279. PortSendOp ::= SendOpKeyword "(" SendParameter ")" [ToClause]
280. SendOpKeyword ::= "send"
281. SendParameter ::= TemplateInstance
282. ToClause ::= ToKeyword AddressRef
283. ToKeyword ::= "to"
284. AddressRef ::= VariableRef | FunctionInstance
/* STATIC SEMANTICS - VariableRef and FunctionInstance return shall be of address or component type
*/
285. CallStatement ::= Port Dot PortCallOp  [PortCallBody]
286. PortCallOp ::= CallOpKeyword "(" CallParameters ")" [ToClause]
287. CallOpKeyword ::= "call"
288. CallParameters ::= TemplateInstance ["," CallTimerValue]
/* STATIC SEMANTICS - only out parameters may be omited or specified with a matching attribute */
289. CallTimerValue ::= TimerValue | NowaitKeyword
/* STATIC SEMANTICS - Value shall be of type float */
290. NowaitKeyword ::= "nowait"
291. PortCallBody ::= BeginChar
               CallBodyStatementList
               EndChar
292. CallBodyStatementList ::= {CallBodyStatement [SemiColon]}+
293. CallBodyStatement ::= CallBodyGuard  StatementBlock
294. CallBodyGuard ::= AltGuardChar CallBodyOps
295. CallBodyOps ::= GetReplyStatement | CatchStatement
296. ReplyStatement ::= Port Dot PortReplyOp
297. PortReplyOp ::= ReplyKeyword "(" TemplateInstance [ReplyValue]")" [ToClause]
298. ReplyKeyword ::= "reply"
299. ReplyValue ::= ValueKeyword Expression
300. RaiseStatement ::= Port Dot PortRaiseOp
301. PortRaiseOp ::= RaiseKeyword "(" Signature "," TemplateInstance ")" [ToClause]
302. RaiseKeyword ::= "raise"
303. ReceiveStatement ::= PortOrAny Dot PortReceiveOp
304. PortOrAny ::= Port | AnyKeyword PortKeyword
305. PortReceiveOp ::= ReceiveOpKeyword ["(" ReceiveParameter ")"] [FromClause] [PortRedirect]
/* STATIC SEMANTICS - The PortRedirect option may only be present if the ReceiveParameter option is
also present  */
306. ReceiveOpKeyword ::= "receive"
307. ReceiveParameter ::= TemplateInstance
308. FromClause ::= FromKeyword AddressRef
309. FromKeyword ::= "from"
310. PortRedirect ::= PortRedirectSymbol (ValueSpec [SenderSpec] | SenderSpec)
311. PortRedirectSymbol ::= "->"
312. ValueSpec ::= ValueKeyword VariableRef
313. ValueKeyword ::= "value"
314. SenderSpec ::= SenderKeyword VariableRef
/* STATIC SEMANTICS - Variable ref shall be of address or component type */
315. SenderKeyword ::= "sender"
316. TriggerStatement ::= PortOrAny Dot PortTriggerOp
317. PortTriggerOp ::= TriggerOpKeyword ["(" ReceiveParameter ")"] [FromClause] [PortRedirect]
/* STATIC SEMANTICS - The PortRedirect option may only be present if the ReceiveParameter option is
also present */
318. TriggerOpKeyword ::= "trigger"
319. GetCallStatement ::= PortOrAny  Dot PortGetCallOp
320. PortGetCallOp ::= GetCallOpKeyword ["(" ReceiveParameter ")"] [FromClause]
[PortRedirectWithParam]
/* STATIC SEMANTICS - The PortRedirectWithParam option may only be present if the ReceiveParameter
option is also present */
321. GetCallOpKeyword ::= "getcall"
```

```
322. PortRedirectWithParam ::= PortRedirectSymbol RedirectSpec
323. RedirectSpec ::= ValueSpec  [ParaSpec] [SenderSpec] |
               ParaSpec [SenderSpec] |
               SenderSpec
324. ParaSpec ::= ParaKeyword ParaAssignmentList
325. ParaKeyword ::= "param"
326. ParaAssignmentList ::= "(" (AssignmentList | VariableList) ")"
327. AssignmentList ::= VariableAssignment {"," VariableAssignment}
328. VariableAssignment ::= VariableRef AssignmentChar ParameterIdentifier
/* STATIC SEMANTICS - The parameterIdentifiers shall be from the corresponding signature definition
*/
329. ParameterIdentifier ::= ValueParIdentifier |
               TimerParIdentifier |
               TemplateParIdentifier |
               PortParIdentifier
330. VariableList ::= VariableEntry  {"," VariableEntry}
331. VariableEntry ::= VariableRef | NotUsedSymbol
332. GetReplyStatement ::= PortOrAny Dot PortGetReplyOp
333. PortGetReplyOp ::= GetReplyOpKeyword ["(" ReceiveParameter [ValueMatchSpec] ")"]
               [FromClause] [PortRedirectWithParam]
/* STATIC SEMANTICS - The PortRedirectWithParam option may only be present if the ReceiveParameter
option is also present */
334. GetReplyOpKeyword ::= "getreply"
335. ValueMatchSpec ::= ValueKeyword TemplateInstance
336. CheckStatement ::= PortOrAny Dot PortCheckOp
337. PortCheckOp ::= CheckOpKeyword ["(" CheckParameter ")"]
338. CheckOpKeyword ::= "check"
339. CheckParameter ::= PortReceiveOp | PortGetCallOp | PortGetReplyOp | PortCatchOp |
               [FromClause] [PortRedirectSymbol SenderSpec]
340. CatchStatement ::= PortOrAny Dot PortCatchOp
341. PortCatchOp ::= CatchOpKeyword ["("CatchOpParameter ")"] [FromClause] [PortRedirect]
/* STATIC SEMANTICS - The PortRedirect option may only be present if the CatchOpParameter option is
also present */
342. CatchOpKeyword ::= "catch"
343. CatchOpParameter ::= Signature "," TemplateInstance | TimeoutKeyword
344. ClearStatement ::= PortOrAll Dot PortClearOp
345. PortOrAll ::= Port | AllKeyword PortKeyword
346. PortClearOp ::= ClearOpKeyword
347. ClearOpKeyword ::= "clear"
348. StartStatement ::= PortOrAll Dot PortStartOp
349. PortStartOp ::= StartKeyword
350. StopStatement ::= PortOrAll Dot PortStopOp
351. PortStopOp ::= StopKeyword
352. StopKeyword ::= "stop"
353. AnyKeyword ::= "any"
```

A.1.6.3.5 Timer Operations

```
354. TimerStatements ::= StartTimerStatement | StopTimerStatement | TimeoutStatement
355. TimerOps ::= ReadTimerOp | RunningTimerOp
356. StartTimerStatement ::= TimerRef Dot StartKeyword ["(" TimerValue ")"]
357. StopTimerStatement ::= TimerRefOrAll Dot StopKeyword
358. TimerRefOrAll ::= TimerRef | AllKeyword TimerKeyword
359. ReadTimerOp ::= TimerRef Dot ReadKeyword
360. ReadKeyword ::= "read"
361. RunningTimerOp ::= TimerRefOrAny  Dot RunningKeyword
362. TimeoutStatement ::= TimerRefOrAny  Dot TimeoutKeyword
363. TimerRefOrAny ::= TimerRef | AnyKeyword TimerKeyword
364. TimeoutKeyword ::= "timeout"
```

A.1.6.4 Type

```
365. Type ::= PredefinedType | ReferencedType
366. PredefinedType ::= BitStringKeyword |
               BooleanKeyword |
               CharStringKeyword |
               UniversalCharString |
               CharKeyword |
               UniversalChar |
               IntegerKeyword |
               OctetStringKeyword |
               ObjectIdentifierKeyword |
               HexStringKeyword |
               VerdictKeyword |
               FloatKeyword |
               AddressKeyword
367. BitStringKeyword ::= "bitstring"
368. BooleanKeyword ::= "boolean"
```

```
369. IntegerKeyword ::= "integer"
370. OctetStringKeyword ::= "octetstring"
371. ObjectIdentifierKeyword ::= "objid"
372. HexStringKeyword ::= "hexstring"
373. VerdictKeyword ::= "verdict"
374. FloatKeyword ::= "float"
375. AddressKeyword ::= "address"
376. CharStringKeyword ::= "charstring"
377. UniversalCharString ::= UniversalKeyword CharStringKeyword
378. UniversalKeyword ::= "universal"
379. CharKeyword ::= "char"
380. UniversalChar ::= UniversalKeyword CharKeyword
381. ReferencedType ::= [GlobalModuleId Dot] TypeReference [ExtendedFieldReference]
382. TypeReference ::= StructTypeIdentifier[TypeActualParList] |
               EnumTypeIdentifier |
               SubTypeIdentifier |
               TypeParIdentifier |
               ComponentTypeIdentifier
383. TypeActualParList ::= "(" TypeActualPar {"," TypeActualPar} ")"
384. TypeActualPar ::= SingleConstExpression | Type
```

A.1.6.4.1 Array Types

```
385. ArrayDef ::= {"[" ArrayBounds [".." ArrayBounds] "]"}+
386. ArrayBounds ::= SingleConstExpression
/* STATIC SEMANTICS - ArrayBounds will resolve to a non negative value of integer type */
```

A.1.6.5 Value

```
387. Value ::= PredefinedValue | ReferencedValue
388. PredefinedValue ::= BitStringValue |
               BooleanValue |
               CharStringValue |
               IntegerValue |
               OctetStringValue |
               ObjectIdentifierValue |
               HexStringValue |
               VerdictValue |
               EnumeratedValue |
               FloatValue |
               AddressValue
389. BitStringValue ::= Bstring
390. BooleanValue ::= "true" | false
391. IntegerValue ::= Number
392. OctetStringValue ::= Ostring
393. ObjectIdentifierValue ::= ObjectIdentifierKeyword  "{" ObjIdComponentList "}"
/* STATIC SEMANTICS - ReferencedValue shall be of type object Identifer */
394. ObjIdComponentList ::= {ObjIdComponent}+
395. ObjIdComponent ::= NameForm |
               NumberForm |
               NameAndNumberForm
396. NumberForm ::= Number | ReferencedValue
/* STATIC SEMANTICS - referencedValue shall be of type integer and have a non negative Value */
397. NameAndNumberForm ::= Identifier  NumberForm
398. NameForm ::= Identifier
399. HexStringValue ::= Hstring
400. VerdictValue ::= "pass" | fail | inconc | none | error
401. EnumeratedValue ::= NamedValueIdentifier
402. CharStringValue ::= Cstring | Quadruple | ReferencedValue
/* STATIC SEMANTICS - ReferencedValue shall resolve to a string type */
403. Quadruple ::= "(" Group "," Plane "," Row "," Cell ")"
404. Group ::= Number
405. Plane ::= Number
406. Row ::= Number
407. Cell ::= Number
408. FloatValue ::= FloatDotNotation | FloatENotation
409. FloatDotNotation ::= Number Dot DecimalNumber
410. FloatENotation ::= Number [Dot DecimalNumber] Exponential [Minus] Number
411. Exponential ::= E
412. ReferencedValue ::= ValueReference [ExtendedFieldReference]
413. ValueReference ::= [GlobalModuleId Dot] ConstIdentifier |
               ExtConstIdentifier |
               ValueParIdentifier |
               ModuleParIdentifier |
               VarIdentifier
414. Number ::= (NonZeroNum {Num}) | 0
415. NonZeroNum ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
416. DecimalNumber ::= {Num}
```

```
417. Num ::= 0 | NonZeroNum
418. Bstring ::= "'" {Bin} "'" B
419. Bin ::= 0 | 1
420. Hstring ::= "'" {Hex} "'" H
421. Hex ::= Num | A | B | C | D | E | F| a | b | c | d | e | f
422. Ostring ::= "'" {Oct} "'" O
423. Oct ::= Hex Hex
424. Cstring ::= """ {Char} """
425. Char ::= /* REFERENCE - A character defined by the relevant CharacterString type */
426. Identifier ::= Alpha{AlphaNum | Underscore}
427. Alpha ::= UpperAlpha | LowerAlpha
428. AlphaNum ::= Alpha | Num
429. UpperAlpha ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T |
U | V | W | X | Y | Z
430. LowerAlpha ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t |
u | v | w | x | y | z
431. ExtendedAlphaNum ::= /* REFERENCE - A character from any character set defined in ISO/IEC 10646
*/
432. FreeText ::= """ {ExtendedAlphaNum} """
433. AddressValue ::= "null"
```

A.1.6.6 Parameterisation

```
434. InParKeyword ::= "in"
435. OutParKeyword ::= "out"
436. InOutParKeyword ::= "inout"
437. FormalValuePar ::= [(InParKeyword | InOutParKeyword | OutParKeyword)] Type ValueParIdentifier
438. ValueParIdentifier ::= Identifier
439. FormalTypePar ::= [InParKeyword] TypeParIdentifier
440. TypeParIdentifier ::= Identifier
441. FormalPortPar ::= [InOutParKeyword] PortTypeIdentifier PortParIdentifier
442. PortParIdentifier ::= Identifier
443. FormalTimerPar ::= [InOutParKeyword] TimerKeyword TimerParIdentifier
444. TimerParIdentifier ::= Identifier
445. FormalTemplatePar ::= [InParKeyword] TemplateKeyword Type TemplateParIdentifier
446. TemplateParIdentifier ::= Identifier
```

A.1.6.7 With Statement

```
447. WithStatement ::= WithKeyword WithAttribList
448. WithKeyword ::= "with"
449. WithAttribList ::= "{" MultiWithAttrib "}"
450. MultiWithAttrib ::= {SingleWithAttrib [SemiColon]}+
451. SingleWithAttrib ::= AttribKeyword  [OverrideKeyword] [AttribQualifier] AttribSpec
452. AttribKeyword ::= EncodeKeyword |
                       DisplayKeyword |
                       ExtensionKeyword
453. EncodeKeyword ::= "encode"
454. DisplayKeyword ::= "display"
455. ExtensionKeyword ::= "extension"
456. OverrideKeyword ::= "override"
457. AttribQualifier ::= "(" DefOrFieldRefList ")"
458. DefOrFieldRefList ::= DefOrFieldRef {"," DefOrFieldRef}
459. DefOrFieldRef ::= DefinitionRef | FieldReference
460. DefinitionRef ::= StructTypeIdentifier |
                       EnumTypeIdentifier |
                       PortTypeIdentifier |
                       ComponentTypeIdentifier |
                       SubTypeIdentifier |
                       ConstIdentifier |
                       TemplateIdentifier |
                       NamedAltIdentifier |
                       TestcaseIdentifier |
                       FunctionIdentifier |
                       SignatureIdentifier
461. AttribSpec ::= FreeText
```

A.1.6.8 Behaviour Statements

```
462. BehaviourStatements ::= TestcaseInstance |
                             FunctionInstance |
                             ReturnStatement |
                             AltConstruct |
                             InterleavedConstruct |
                             LabelStatement |
                             GotoStatement |
                             ActivateStatement |
```

```
              DeactivateStatement |
              NamedAltInstance
/* STATIC SEMANTICS - TestcaseInstance shall not be called from within an existing executing
testcase or function chain called from a testcase i.e. testcases can only be instantiated from the
control part or from functions directly called from the control part */
463. VerdictStatements ::= SetLocalVerdict
464. VerdictOps ::= GetLocalVerdict
465. SetLocalVerdict ::= SetVerdictKeyword "(" SingleExpression ")"
/* STATIC SEMANTICS - SingleExpression shall resolve to a value of type verdict */
/* STATIC SEMANTICS - The SetLocalVerdict shall not be used to assign the Value error */
466. SetVerdictKeyword ::= VerdictKeyword Dot SetKeyword
467. GetLocalVerdict ::= VerdictKeyword Dot GetKeyword
468. GetKeyword ::= "get"
469. SUTStatements ::= SUTAction "(" (FreeText | TemplateRefWithParList)  ")"
470. SUTAction ::= SUTKeyword Dot ActionKeyword
471. SUTKeyword ::= "sut"
472. ActionKeyword ::= "action"
473. ReturnStatement ::= ReturnKeyword  [Expression]
474. AltConstruct ::= AltKeyword BeginChar AltGuardList EndChar
475. AltKeyword ::= "alt"
476. AltGuardList ::= {AltGuardElement [SemiColon]}+  [ElseStatement [SemiColon]]
477. AltGuardElement ::= GuardStatement | ExpandStatement
478. GuardStatement ::= AltGuardChar GuardOp  StatementBlock
479. ExpandStatement ::= "["ExpandKeyword "]" NamedAltInstance
480. ElseStatement ::= "["ElseKeyword "]" StatementBlock
481. ExpandKeyword ::= "expand"
482. AltGuardChar ::= "[" [BooleanExpression] "]"
483. GuardOp ::= TimeoutStatement | ReceiveStatement | TriggerStatement | GetCallStatement |
              CatchStatement | CheckStatement | GetReplyStatement | DoneStatement
/* STATIC SEMANTICS - GuardOp used within the module control part. Shall only contain the
timeoutStatement */
484. StatementBlock ::= BeginChar [FunctionStatementOrDefList] EndChar
485. InterleavedConstruct ::= InterleavedKeyword BeginChar InterleavedGuardList EndChar
486. InterleavedKeyword ::= "interleave"
487. InterleavedGuardList ::= {InterleavedGuardElement [SemiColon]}+
488. InterleavedGuardElement ::= InterleavedGuard InterleavedAction
489. InterleavedGuard ::= "[" "]" GuardOp
490. InterleavedAction ::= StatementBlock
/* STATIC SEMANTICS - The StatementBlock may not contain loop statements, goto, activate,
deactivate, stop, return or calls to functions */
491. LabelStatement ::= LabelKeyword LabelIdentifier
492. LabelKeyword ::= "label"
493. LabelIdentifier ::= Identifier
494. GotoStatement ::= GotoKeyword (LabelIdentifier | AltKeyword)
/* STATIC SEMANTICS - The AltKeyword option may only be used within an ALT construct */
495. GotoKeyword ::= "goto"
496. ActivateStatement ::= ActivateKeyword "(" NamedAltList ")"
497. ActivateKeyword ::= "activate"
498. NamedAltList ::= NamedAltInstance {"," NamedAltInstance}
499. DeactivateStatement ::= DeactivateKeyword ["(" NamedAltRefList ")"]
500. DeactivateKeyword ::= "deactivate"
501. NamedAltRefList ::= NamedAltRef {"," NamedAltRef}
```

A.1.6.9 Basic Statements

```
502. BasicStatements ::= Assignment | LogStatement | LoopConstruct | ConditionalConstruct
503. Expression ::= SingleExpression | CompoundExpression
/* STATIC SEMANTICS - Expression shall not contain Configuration or verdict operations within the
module control part */
504. CompoundExpression ::= FieldExpressionList | ArrayExpression
505. FieldExpressionList ::= "{" FieldExpressionSpec {"," FieldExpressionSpec} "}"
506. FieldExpressionSpec ::= FieldReference AssignmentChar Expression
507. ArrayExpression ::= "{" [ArrayElementExpressionList] "}"
508. ArrayElementExpressionList ::= NotUsedOrExpression  {"," NotUsedOrExpression}
509. NotUsedOrExpression ::= Expression | NotUsedSymbol
510. ConstantExpression ::= SingleConstExpression | CompoundConstExpression
511. SingleConstExpression ::= SingleExpression
/* STATIC SEMANTICS - SingleConstExpression shall not contain Variables or Module parameters and
shall resolve to a constant Value at compile time */
512. BooleanExpression ::= SingleExpression
/* STATIC SEMANTICS - BooleanExpression shall resolve to a Value of type Boolean */
513. CompoundConstExpression ::= FieldConstExpressionList | ArrayConstExpression
514. FieldConstExpressionList ::= "{" FieldConstExpressionSpec {"," FieldConstExpressionSpec} "}"
515. FieldConstExpressionSpec ::= FieldReference AssignmentChar ConstantExpression
516. ArrayConstExpression ::= "{" [ArrayElementConstExpressionList] "}"
517. ArrayElementConstExpressionList ::= ConstantExpression {"," ConstantExpression}
518. Assignment ::= VariableRef  ":=" Expression
```

```
/* OPERATIONAL SEMANTICS - The Expression on the RHS of Assignment shall evaluate to an explicit
Value of the type of the LHS. */
519. SingleExpression ::= SimpleExpression {BitOp SimpleExpression}
/* OPERATIONAL SEMANTICS - If both SimpleExpressions and the BitOp exist then the SimpleExpressions
shall evaluate to specific values of compatible types */
520. SimpleExpression ::= SubExpression [RelOp SubExpression]
/* OPERATIONAL SEMANTICS - If both SubExpressions and the RelOp exist then the SubExpressions shall
evaluate to specific values of compatible types. */
/* OPERATIONAL SEMANTICS - If RelOp is "<" | ">" | ">=" | "<=" then each SubExpression shall
evaluate to a specific integer, Enumerated or float Value (these values can beTTCN or ASN.1 values)
*/
521. SubExpression ::= Product [ShiftOp Product]
/* OPERATIONAL SEMANTICS - Each Product shall resolve to a specific Value. If more than one Product
exists the right-hand operand shall be of type integer and if the shift op is '<<' or '>>' then the
left-hand operand shall resolve to either bitstring, hexstring, octetstring or integer type. If the
shift op is '<@' or '@>' then the left-hand operand shall be of type bitstring, hexstring,
charstring or universal charstring */
522. Product ::= Term {AddOp Term}
/* OPERATIONAL SEMANTICS - Each Term shall resolve to a specific Value. If more than one Term exists
then the Terms shall resolve to type integer or float.  */
523. Term ::= Factor {MultiplyOp Factor}
/* OPERATIONAL SEMANTICS - Each Factor shall resolve to a specific Value. If more than one Factor
exists then the Factors shall resolve to type integer or float. */
524. Factor ::= [UnaryOp] Primary
/* OPERATIONAL SEMANTICS - The Primary shall resolve to a specific Value. If UnaryOp exists and is
"not" then Primary shall resolve to type BOOLEAN if the UnaryOp is "+" or "-" then Primary shall
resolve to type integer or float. If the UnaryOp resolves to not4b then the Primary shall resolve to
the type bitstring, hexstring or octetstring. */
525. Primary ::= OpCall | Value | "(" SingleExpression ")"
526. ExtendedFieldReference ::= {(Dot StructFieldIdentifier | ArrayOrBitRef)}+
527. OpCall ::= ConfigurationOps | VerdictOps | TimerOps | TestcaseInstance | FunctionInstance |
TemplateOps
528. AddOp ::= "+" | "-"
/* OPERATIONAL SEMANTICS - Operands of the "+" or "-" operators shall be of type integer or
float(i.e., TTCN or ASN.1 predefined) or derivations of integer or float (i.e., subrange) */
529. MultiplyOp ::= "*" | "/" | mod | rem
/* OPERATIONAL SEMANTICS - Operands of the "*", "/", rem or mod operators shall be of type integer
or float(i.e., TTCN or ASN.1 predefined) or derivations of integer or float (i.e., subrange). */
530. UnaryOp ::= "+" | "-" | not | not4b
/* OPERATIONAL SEMANTICS - Operands of the "+" or "-" operators shall be of type integer or
float(i.e., TTCN or ASN.1 predefined) or derivations of integer or float (i.e., subrange). Operands
of the not operator shall be of type boolean (TTCN or ASN.1) or derivatives of type Boolean.
Operands of the not4b operator will be of type bitstring, octetstring or hexstring. */
531. RelOp ::= "==" | "<" | ">" | "!=" | ">=" | "<="
/* OPERATIONAL SEMANTICS - The precedence of the operators is defined in table 7 */
532. BitOp ::= "and4b" | xor4b |or4b | and | xor | or | StringOp
/* OPERATIONAL SEMANTICS - Operands of the and, or or xor operators shall be of type boolean (TTCN
or ASN.1) or derivatives of type Boolean. Operands of the and4b, or4b or xor4b operator shall be of
type bitstring, hexstring or octetstring (TTCN or ASN.1) or derivatives of these types. */
/* OPERATIONAL SEMANTICS - The precedence of the operators is defined in table 7 */
533. StringOp ::= "&"
/* OPERATIONAL SEMANTICS - Operands of the string operator shall be bitstring, hexstring,
octetstring or character string */
534. ShiftOp ::= "<<" | ">>" | "<@" | "@>"
535. LogStatement ::= LogKeyword "(" [FreeText] ")"
536. LogKeyword ::= "log"
537. LoopConstruct ::= ForStatement |
                      WhileStatement |
                      DoWhileStatement
538. ForStatement ::= ForKeyword "(" Initial [SemiColon] Final [SemiColon] Step ")"
                      StatementBlock
539. ForKeyword ::= "for"
540. Initial ::= VarInstance | Assignment
541. Final ::= BooleanExpression
542. Step ::= Assignment
543. WhileStatement ::= WhileKeyword "(" BooleanExpression ")"
                      StatementBlock
544. WhileKeyword ::= "while"
545. DoWhileStatement ::= DoKeyword StatementBlock
                      WhileKeyword "(" BooleanExpression ")"
546. DoKeyword ::= "do"
547. ConditionalConstruct ::= IfKeyword "(" BooleanExpression ")"
                      StatementBlock
                      {ElseIfClause}   [ElseClause]
548. IfKeyword ::= "if"
549. ElseIfClause ::= ElseKeyword  IfKeyword "(" BooleanExpression ")"  StatementBlock
550. ElseKeyword ::= "else"
551. ElseClause ::= ElseKeyword  StatementBlock
```

A.1.6.10 Miscellaneous productions

```
552. Dot ::= "."
553. Dash ::= "-"
554. Minus ::= Dash
555. SemiColon ::= ";"
556. Colon ::= ":"
557. Underscore ::= "_"
558. BeginChar ::= "{"
559. EndChar ::= "}"
560. AssignmentChar ::= ":="
```

Annex B (normative): Operational semantics

This annex defines the meaning of a 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 by (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, 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 several TTCN-3 constructs is explained by replacing them with other language constructs. For example, named alternatives are macros and their meaning is completely explained by replacing all macro references by the corresponding macro definitions. This includes the handling of default behaviour.

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 test case, function or the module control. The mapping of TTCN-3 behaviour descriptions onto flow graphs is straightforward.

B.1 Structure of this annex

This annex is structured into two parts:

1) The first part (see clause B.2) defines the meaning of TTCN-3 shorthand and macro notations by their replacement by 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.

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

B.2 Replacement of shorthand notations and macro calls

Shorthand notations have to be expanded and macro references have to be replaced by the corresponding definitions on a textual level before this operational semantics can be used for the explanation of TTCN-3 behaviour.

TTCN-3 shorthand notations are:

- stand-alone receiving operations;
- **`trigger`** operations;
- usages of the keyword **`any`** in timer and receiving operations;
- usages of the keyword **`all`** in timer and port operations;
- missing **`return`** and **`stop`** statements at the end of function and test case definitions.

TTCN-3 macros are named alternatives, i.e., **named alt** definitions. They are called:

- explicitly instead of an **alt** statement, i.e., they appear like a function call;
- explicitly in **alt** statements by using an **expand** keyword;
- implicitly in case they are referenced as default behaviour in **activate** and **deactivate** statements.

In addition to shorthand notations and macro calls, 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.

B.2.1 Order of replacement steps

The textual replacements of shorthand notations, macro calls, global constants and module parameters have to be done in the following order:

1) adding **stop** and **return** statements in module control, functions and test cases;
2) replacement of global constants and module parameters by concrete values;
3) embedding stand-alone receiving operations into **alt** statements;
4) macro expansion of pure *macro calls*, this means:
   - explicit expansions of **alt** statements which include the **expand** keyword (and refers to a **named alt** definition);
   - explicit expansion of calls of **named alt**-definitions.
5) expansion of **interleave** statements;
6) expansion of default behaviour;
7) replacement of all **trigger** operations by equivalent **receive** operations and **goto** statements;
8) replacement of all usages of the keywords **any** and **all** in timer and port operations.

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

B.2.2 Adding stop and return operations in behaviour descriptions

TTCN-3 allows leaving module control, test cases and functions that do not return any value without specifying an explicit **stop** or **return** operation. For the operational semantics it is assumed that missing **return** and **stop** operations are added, i.e., **stop** operations are added in module control and test cases and **return** operations are added in functions.

EXAMPLE:

```
// Function and test case definition without explicit return and stop statements at
// the end of their behaviour description

    function MyFunction(inout integer MyPar) {
                                    // MyFunction doesn't return a value but changes the value
        MyPar := 10 * MyPar1;       // of MyPar which is passed in by refefernce
        if (MyPar == 999) stop;     // Stops execution if MyPar has the value 999

        // IMPLICIT return if MyPar != 999
    }

    testcase MyTestCase() runs on MyMTCtype {
        MyMTCbehaviour();           // Function that defines MTC behavior

        // IMPLICIT stop after return of MyMTCbehaviour
    }


// MyFunction and MyTestCase after adding explicit return and stop operations

    function MyFunction(inout integer MyPar) {
                                    // MyFunction doesn't return a value but changes the value
        MyPar := 10 * MyPar1;       // of MyPar which is passed in by refefernce
        if (MyPar == 999) stop;     // Stops execution if MyPar has the value 999

        return;                     // EXPLICIT return
    }

    testcase MyTestCase() runs on MyMTCtype {

        MyMTCbehaviour();           // Function that defines MTC behavior

        stop;                       // EXPLICIT stop
    }
```

B.2.3 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.

B.2.4 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, a named alternative or a test case. In such a case the receiving operation as considered to be shorthand for an **alt** statement with only one alternative defined by the receiving operation. For the operational semantics an **alt** statement in which the receiving statement is embedded shall replace all stand-alone occurrences of receiving operations.

EXAMPLE:

```
// The stand-alone occurrence of
     :
    MyCL.trigger(MyType:*);
     :

// shall be replaced by
     :
    alt {
        []  MyCL.trigger (MyType:*);
    }
     :

// or
     :
    MyPTC.done;
     :

// shall be replaced by
     :
    alt {
        []  MyPTC.done;
    }
     :
```

B.2.5 Macro expansion

The macro expansion in TTCN-3 is related to the usage of named alternatives (**named alt** definitions) in **alt** statements or instead of **alt** statements, i.e., the named **alt** definition is referenced similar to a function call in a sequence of statements.

B.2.5.1 Expansion of named alternatives in alternative statements

The expansion of named alternatives in **alt** statements is related to the alternative branches indicated by the **expand** keyword in square brackets followed by a reference to a **named alt** definition (as only statement of that branch). In such a case the alternative branches of the referenced named alternative replace the branch with the **expand** keyword. For the operational semantics it is assumed that this replacement is done on a syntactical level. An example of this expansion can be found in the main part of the present document.

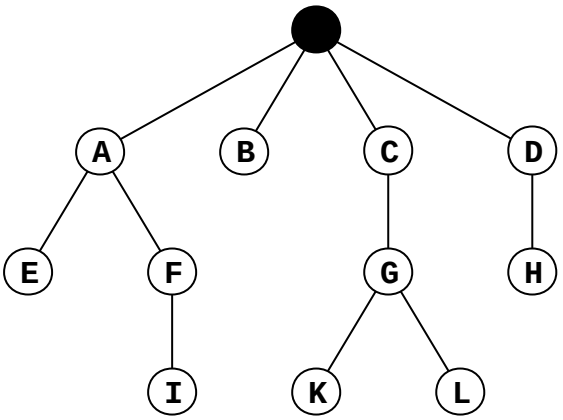
B.2.5.2 Explicit call of a named alternative

Named alternatives can also be referenced similar to a function call in a sequence of statements. In this case the reference shall be expanded by the corresponding **named alt** definition. An example of this expansion can be found in the main part of the present document.

B.2.6 Replacement of the interleave construct

The meaning of the **interleave** statement is defined by its replacement by a series of nested **alt** statements that have 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.

A series of nested **alt** statements can be described by means of a tree. Tree nodes represent the statements in the **alt** statements. A branching denotes an **alt** statement and statements in the same branch describe statements in the same alternative. This is schematically shown in figure B.1. Figure B.1a) presents a tree and figure B.1b) shows the corresponding representation in form of a series of nested **alt** statements.



(a) Tree

```
alt {
    [] A {
            alt {
                [] E ;
                [] F {
                        I
                    }
            }
        }
    [] B;
    [] C {
            G;
            alt {
                [] K;
                [] L;
            }
        }
    [] D{
            H
        }
}
```

(b) TTCN-3 like representation of (a)

Figure B.1: Nested alt statements and a corresponding tree representation

In the following the construction of a tree representation of an **interleave** statement is presented. The transformation of the tree into the series of nested **alt** statements is straightforward and needs no further explanation.

An **interleave** statement can be seen as a partial ordered set ***POS*** of allowed TTCN-3 statements. Formally:

- ***POS*** = (***S***, <) where:

  S is the set of allowed TTCN-3 statements; and

  $< \subseteq (S \times S)$ describes the reflexive and transitive order relation.

The term *allowed TTCN-3 statements* refers to the fact that the control transfer statements **for**, **while**, **do-while**, **goto**, **activate**, **deactivate**, **stop**, **return** and calls of user-defined functions which include communication operations are not allowed to be used in **interleave** statements. In addition, it is also not allowed to guard branches of an **interleave** statement with Boolean expressions, to expand **interleave** statements with named alternatives or to specify **else** branches.

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

- The *DISCARD* function deletes an element ***s*** from a partially ordered set ***POS*** and returns the resulting partially ordered set ***POS'***:

  $DISCARD(s, POS) = POS'$ where:

  $POS' = (S', <')$; and

  $S' = S \backslash \{s\}$; and

  $<' = < \cap (S \backslash \{s\} \times S \backslash \{s\})$.

- The *ENABLED* function takes a partially ordered set ***POS*** = (***S***, <) and returns all elements which have no predecessors in ***POS***:

  $ENABLED(POS) = \{ s \mid s \in S \wedge (< \cap (S \times \{s\}) = \emptyset)\}$

- The *RECEIVING* function takes a set of TTCN-3 statements ***S*** and returns all receiving statements from this set.

$$RECEIVING(\boldsymbol{S}) = \{\, \boldsymbol{s} \mid \boldsymbol{s} \in \boldsymbol{S} \wedge kind(\boldsymbol{s}) \in \{\textbf{receive}, \textbf{trigger}, \textbf{getcall}, \textbf{getreply}, \textbf{catch}, \textbf{check}, \textbf{done}, \textbf{timeout}\}\}$$

- The *SELECT* function selects randomly an element ***s*** from a given set ***S*** and returns ***s***.

$$SELECT(\boldsymbol{S}) = \boldsymbol{s} \qquad \text{where } \boldsymbol{s} \in \boldsymbol{S}$$

NOTE: The *kind* function in the *RECEIVING* function above is not defined formally. *kind* (or type) returns the kind of a given TTCN-3 statement.

The construction algorithm 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 (treeNode *predecessor, partiallyOrderedSet POS) {
    // treeNode refers to the node type of the tree to be constructed
    // partiallyOrderedSet denotes type for a partially ordered set of TTCN-3 statements

    var statement myStmt;                                  // for the storage of a TTCN-3 statement
    var treeNode *newSonNode;                              // for the handling of new tree nodes

// RETRIEVING SETS OF TTCN-3 STATEMENTS THAT HAVE NO PREDECESSORS IN 'POS'
    var statementSet enabStmts := ENABLED(POS);            // all statements without predecessor
    var statementSet enabRecStmts := RECEIVING(enabStmts);// receiving statements in 'enabStmts'
    var statementSet enabNonRecStmts := enabStmts\enabRecStmts;
                                                  // non receiving statements in 'enabStmts'

    if (POS == ∅)
        return;                                           // TERMINATION CRITERION OF RECURSION
    else {
        if (enabNonRecStmts != ∅) {       // Handling of non receiving statements in 'enabStmts'
            myStmt := SELECT(enabNonRecStmts);
            newSonNode := create(myStmt, predecessor);
                          // Creation of a new tree node representing 'myStmt' in the tree
                          // and update of pointers in 'newSonNode' and 'predecessor'.
            CONSTRUCT-SUCCESSORS(newSonNode, DISCARD(myStmt, POS));    // NEXT RECURSION STEP
        }
        else { // Handling of receiving events, the tree will branch
            for each (myStmt in enabRecStmts) {
                newSonNode := create(myStmt, predecessor);// New tree node
                CONSTRUCT-SUCCESSORS(newSonNode, DISCARD(myStmt, POS));// NEXT RECURSION STEP(S)
            }
        }
    }
}
```

Initially, the CONSTRUCT-SUCCESSORS function will be called with a *root node* of an empty tree and the partially ordered set 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.

B.2.7 Expansion of defaults

The TTCN-3 default behaviour mechanism is defined by means of a macro expansion mechanism. The default behaviour has to be provided in form of **named alt** definitions. A **named alt** definition used as default behaviour is referenced in an **activate** statement. The scope of a default is determined by an **activate** statement and corresponding **deactivate** statements or by an **activate** statement and the end of the function or test case in which the **activate** statement is used. Within this scope the alternatives of all **alt** statements are extended by the behaviour specified in the activated **named alt** definitions. The operational semantics assumes that this extension is done on the syntactical level. An example for the extension mechanism can be found in the main part of the present document.

B.2.8 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:

```
// The following trigger operation …

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

// shall be replaced by …

    alt {
        []  MyCL.receive (MyType:*);
        []  MyCL.receive {
                goto alt
            }
    }
```

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

EXAMPLE:

```
// The following alt statement includes a trigger statement …

    alt {
        []  PCO2.receive {
                stop;
            }
        []  MyCL.trigger (MyType:*);
        []  PCO3.catch {
                verdict.set(fail);
                stop;
            }
    }

// which will be replaced by

    alt {
        []  PCO2.receive {
                stop;
            }
        []  MyCL.receive (MyType:*);
        []  MyCL.receive {
                goto alt;
            }
        []  PCO3.catch {
                verdict.set(fail);
                stop;
            }
    }
```

B.2.9 Replacement of the keywords 'any' and 'all'

The usage of the keyword **any** is allowed for:

- the timer operations **running** and **timeout;**
- the receiving operations **receive**, **trigger**, **getcall**, **getreply**, **catch**, **check**.

The usage of the keyword **all** is allowed for:

- the timer operation **stop**;
- the port operations **start**, **stop** and **clear**.

The usage of both keywords is allowed for:

- the **done** and **running** operations for components.

B.2.9.1 Replacement of 'all' in timer and port operations

The application of timer and port operations is related to the scope in which they are used. This means, the keyword **all** addresses all timers and ports known in the scope unit in which **all** (+ operation) is used. The replacement of **all** usages in timer and port operations is straightforward.

A usage of **all port** in a **start**, **stop**, or **clear** operation shall be replaced by a separate **start**, **stop**, or **clear** operation for each known port. A usage of **all timer** in a **stop** operation shall be replaced by a separate **stop** operation for each known timer.

EXAMPLE:

```
// Assume the ports PCO1, PCO2 and the timers T1 and T2 are known

     :
    all port.clear;
     :
     :
    all timer.stop;
     :

// will be replaced by

     :
    PCO1.clear;
    PCO2.clear;
     :
     :
    T1.stop;
    T2.stop;
     :
```

B.2.9.2 Replacement of 'any' in timer and receiving operations

The application of timer and receiving operations is related to the scope in which they are used. This means, the keyword **any** addresses all timers and ports (in case of receiving operations) known in the scope unit in which **any** (+ operation) is used. The replacement of **any** usages in timer and receiving operations is straightforward.

A usage of **any port** in a **receive**, **trigger**, **getcall**, **getreply**, **catch** or **check** operation shall be replaced by separate alternative operations for each known and possible port. Possible means that an **any port.receive** occurrence only is relevant for message based ports.

A usage of **any timer** in a **timeout** operation shall be replaced by separate alternative operations for each known timer in the scope unit.

EXAMPLE:

```
// Assume the ports PCO1, PCO2 and the timers T1 and T2 are known

    alt {
        [] PCO2.receive {
               aTestStep();
           }
        [] any port.receive {
               verdict.set(fail);
               stop;
           }
        [] any timer.timeout {
               verdict.set(fail);
               stop;
           }
    }
```

```
// will be replaced by

    alt {
        [] PCO2.receive {
               stop;
           }
        [] PCO1.receive {
               verdict.set(fail);
               stop;
           }
        [] PCO1.receive {
               verdict.set(fail);
               stop;
           }
        [] T1.receive {
               verdict.set(fail);
               stop;
           }
        [] T2.receive {
               verdict.set(fail);
               stop;
           }
    }
```

A usage of **any timer** in a **running** operation shall be replaced by separate **running** operations for each known timer in the scope unit that are combined by means of **or** operators.

EXAMPLE:

```
// Assume the timers T1 and T2 are known in the scope unit

     :
    if (any timer.running) {
        verdict.set(fail);
        stop;
    }
     :

// will be replaced by

     :
    if (T1.running or T2.running) {
        verdict.set(fail);
        stop;
    }
     :
```

B.2.9.3 The keywords 'any' and 'all' in 'done' and 'running'

The operations **any component.done**, **all component.done**, **any component.running** and **all component.running** can only be executed by the MTC. Due to dynamic test component creation, the MTC may not know all components that have been created during test case execution. Thus, the execution of these operations requires communication with the means of testing. Therefore, **any component.done**, **all component.done**, **any component.running** and **all component.running** are assumed to be system commands, i.e., cannot be replaced by other commands.

B.3 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 B.3.1), the construction of flow graphs representing TTCN-3 module control, test cases, functions and component type definitions is explained (see clause B.3.2), module and component states for the description of the execution states of a TTCN-3 module are defined (see clause B.3.3), the handling of messages, remote procedure calls, replies to remote procedure calls and exceptions is described (see clause B.3.4), the evaluation procedure of module control and test cases is explained (see clause B.3.6) and the meaning of the different TTCN-3 statements is described (see clause B.3.7).

B.3.1 Flow graphs

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

B.3.1.1 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 B.2.

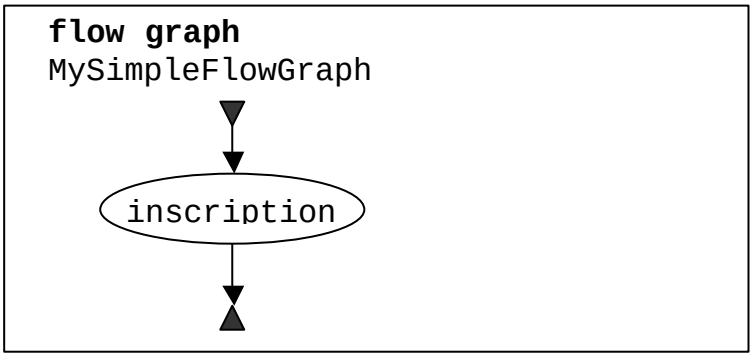


Figure B.2: A simple flow graph

B.3.1.2 Flow graph nodes

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

B.3.1.2.1 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 B.3a).

(a) Flow graph start node **(b) Flow graph end node**

Figure B.3: Start and end nodes

B.3.1.2.2 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 B.3.1.2.3) and reference nodes (see clause B.3.1.2.4) 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 B.3b).

B.3.1.2.3 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. A basic node is shown in figure B.4a).

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 B.4b).

(a) **(b)**

Figure B.4: Basic nodes with attributes

B.3.1.2.4 Reference nodes

Reference nodes refer to flow graph segments (see clause B.3.1.4) 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 B.5a).

(a) Single reference node **(b) OR combination of three reference nodes**

Figure B.5: Reference node

B.3.1.2.4.1 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 (figure B.5b). In the actual flow graph representing the module control, a test case or a function, one alternative is determined by the represented construct.

B.3.1.2.4.2 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 B.6, these operators have been adopted to flow graphs by introducing double-framed reference nodes with associated operator symbols. A single flow line shall replace a reference node, in case of zero occurrences (using a double-framed reference node with '*'-operator).

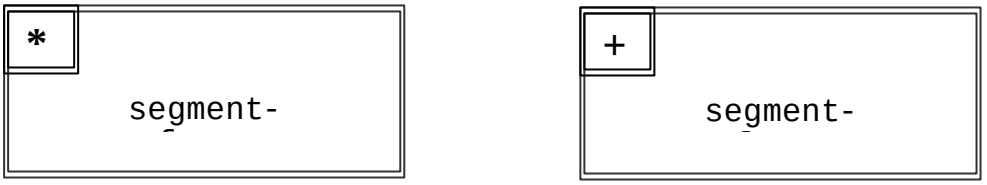


Figure B.6: Repetition of reference nodes

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 B.7 may occur from zero up to 5 times.

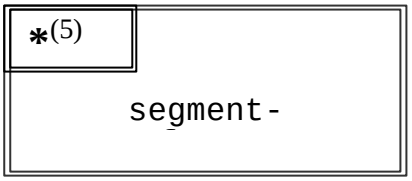
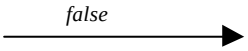
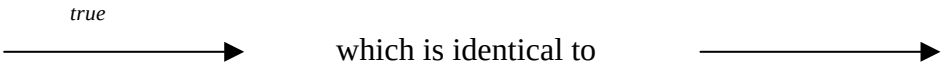


Figure B.7: Restricted repetition of a reference node

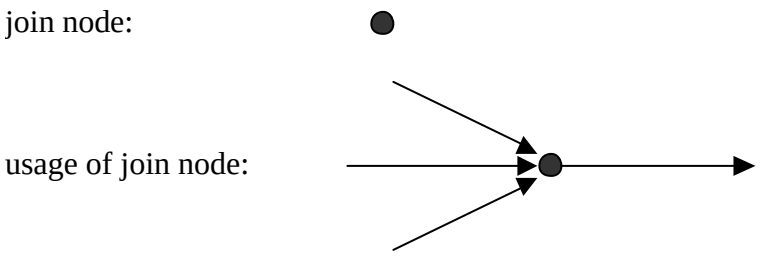
B.3.1.3 Flow lines

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.

Incoming flow line with label: *in-label* →

Outgoing flow line with label: → *out-label*

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.

B.3.1.4 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 B.8, flow graph segments have precise interfaces that consist of incoming and outgoing flow lines. There is only one unlabeled incoming and one or none unlabeled outgoing flow lines. In addition there might exist several labelled incoming and outgoing flow lines. The labelled incoming and outgoing flow lines are needed to describe the meaning of TTCN-3 **`goto`** statements.

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

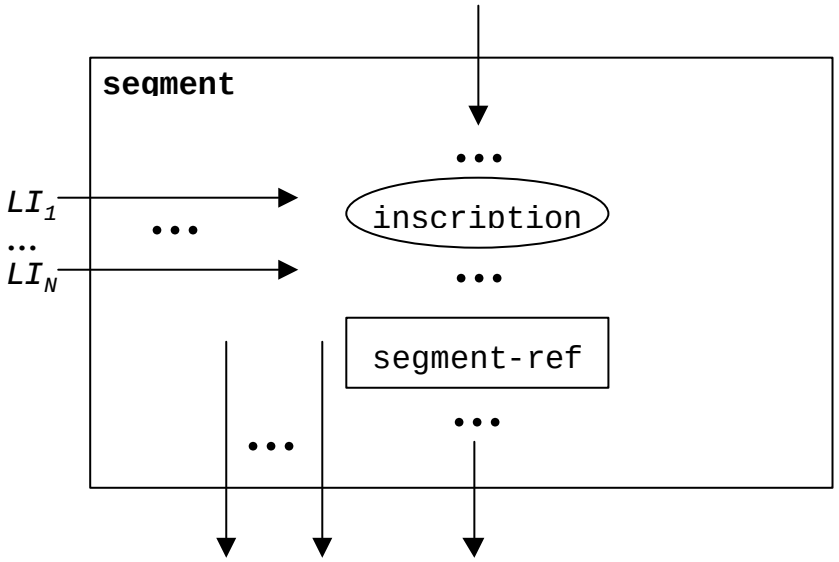
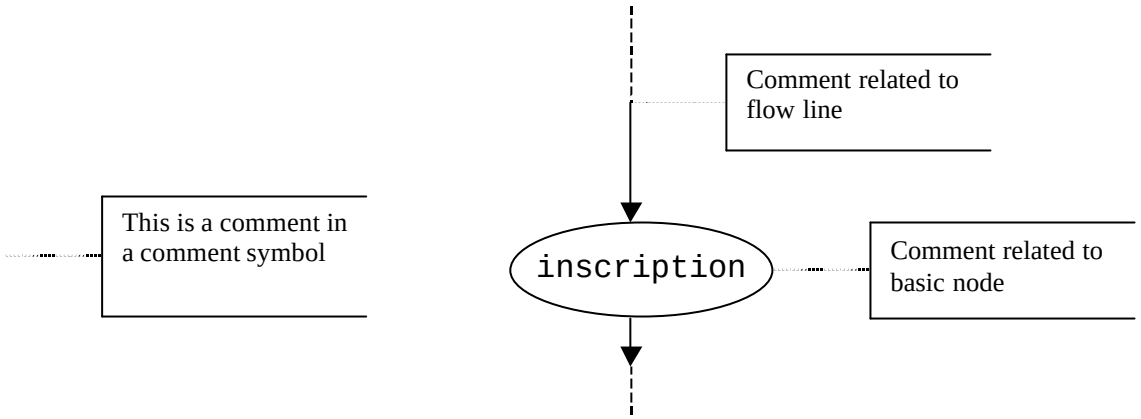


Figure B.8: Schematical flow graph segment description

B.3.1.5 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 B.9.



(a) Comment symbol **(b) Usage of comment symbols**

Figure B.9: Flow graph representation of comments

B.3.1.6 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 are 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 expressing whether a *true* or a *false* successor is returned (see clause B.3.1.3).

B.3.2 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) 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. Function definitions describe behaviour to be executed by the module control or by the test components. 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.

B.3.2.1 The flow graph construction procedure

The flow graphs presented in the figures B.10 and B.11 and the flow graph segments presented in clause B.3.6 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, functions and component type definitions a concrete flow graph segment is constructed.

2) For the module control and for each test case, 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 B.4 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 annex is to provide a correct and complete semantics, not an optimal flow graph representation.

B.3.2.2 Flow graph representation of module control

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

```
module <identifier> (<parameter>) <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 B.3.2.4). 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 B.2.3).

B.3.2.3 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. The information provided by the `<testcase-interface>` is not relevant for the MTC. It will be used by the **`execute`** statement, but needs not to be represented in the flow graph representation of a test case. Thus, for the flow graph representation the following information is relevant only:

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

This is comparable to a function definition and therefore the flow graph representation of a test case is similar to the flow graph representation of a function (see clause B.3.2.4). The semantics will access the flow graphs representing test cases by using the test case names.

B.3.2.4 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 **runs on** and the **return** clauses in the function definition. The information provided by the <function-interface> is not relevant for the behaviour description. It will be used for static semantics checks, but needs not to be represented in the flow graph. Thus, for the flow graph representation the following information is relevant only:

```
function <identifier> (<parameter>) <statement-block>
```

The semantics will access flow graphs representing functions by using the function names.

The scheme of the flow graph representation of a function is shown in figure B.10. The flow graph name <identifier> refers to the name of the represented function (or module control or test case). The nodes of the flow graph have associated comments describing the meaning of the different nodes.

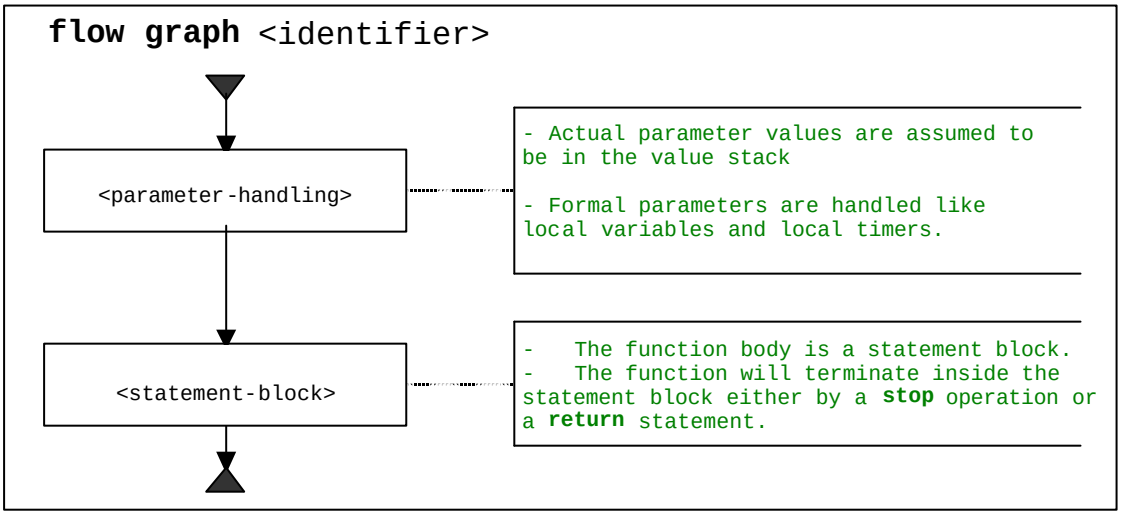


Figure B.10: Flow graph representation of functions

B.3.2.5 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-declartions>
```

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 B.11. The flow graph name `<identifier>` refers to the name of the represented component type.

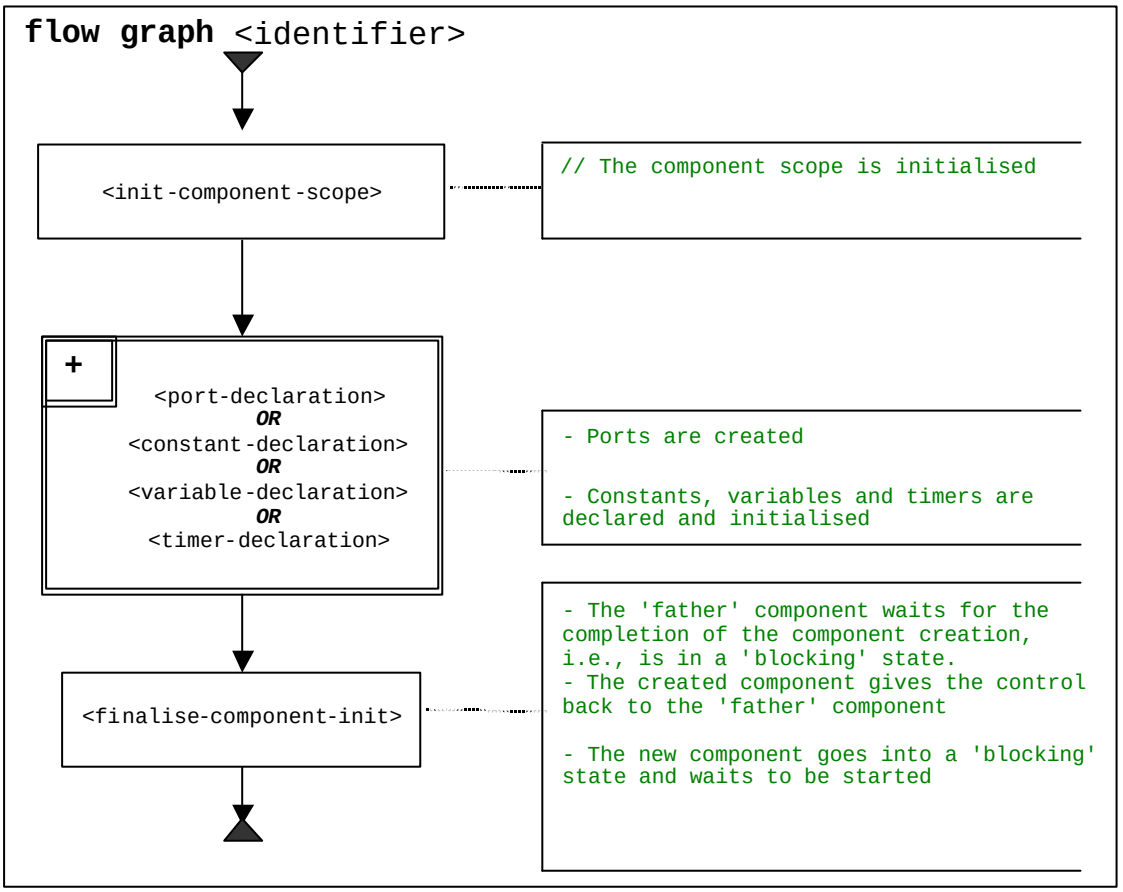


Figure B.11: Flow graph representation of component type definitions

B.3.2.6 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.

B.3.3 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.

B.3.3.1 Module state

As shown in figure B.12 a module state is structured into a *list of entity states*, a *list of port states*, a reference to an *MTC* and a *TC-VERDICT*. The *list of entity states* describes the state of the module control and during the execution of a test case the states of the instantiated test components. The *list of port states*, the *MTC* reference and the *TC-VERDICT* are only relevant during test case execution. The list of port states describes the states of the different ports. *MTC* provides a reference to the MTC, *TC-VERDICT* stores the actual global test verdict of a test case and DONE is a counter that counts the number of updates of *TC-VERDICT*.

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

The behaviour of module control (M-CONTROL in figure B.12) is handled like a normal test component and its state is the first element in the *list of entity states* of a module state.

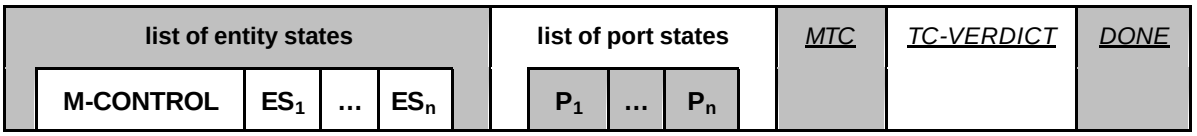


Figure B.12: Structure of a module state

NOTE 2: Port states may be considered to be part of the entity states. However, by **`connect`** and **`map`** ports are made visible for other components and therefore they are handled on the top level of a module state.

B.3.3.1.1 Accessing the module state

The *MTC*, *SYSTEM*, *TC-VERDICT* and *DONE* are parts of a module state are 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 unique identifiers for the evaluation procedure.

For the handling of the *list of entity states* and the *list of port states*, the list operations *append*, *delete*, *first* and *length* can be used.

NOTE 2: The list operations *append*, *delete*, *first* and *length* have the following meaning:

- <*list*>.*append*(<*item*>) appends <*item*> as last element into the list <*list*>;
- <*list*>.*delete*(<*item*>) deletes <*item*> from the list <*list*>;
- <*list*>.*first*() returns the first element of <*list*>;
- <*list*>.*length*() returns the length of <*list*>;
- <*list*>.*next*(<*item*>) returns the element that follows <item> in the list, or **`NULL`** if <*item*> is the last element in the list.

B.3.3.2 Entity states

Entity states are used to describe the actual states of mo dule control and test components. The structure of an entity state is shown in figure B.13.

<*identifier*>	***STATUS***	*CONTROL-STACK*	**Data state**	**timer state**	*VALUE-STACK*	*E-VERDICT*

Figure B.13: 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`** or **`BLOCKED`**. Module control is blocked during the execution of a test case. Test components may be blocked during the creation of other test components, i.e., during the execution of a **`create`** operation.

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 *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 calls. Each list in the list of lists of variable bindings describes the variable bindings in a certain scope unit. Entering or leaving a scope unit corresponds to adding or deleting a list of variable bindings from the *data state*. A more detailed description of the *data state* part of an entity state can be found in clause B.3.3.2.2.

The *timer state* is considered to be a list of lists of timer states. The list of lists structure reflects nested scope units due to nested function calls. Each list in the list of lists of timer states describes the timer bindings (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 states from the *timer state*. A more detailed description of the *timer state* part of an entity state can be found in clause B.3.3.2.3.

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`** function 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.

B.3.3.2.1 Accessing entity states

The *STATUS* and *E-VERDICT* parts of an entity state are handled like global variables, i.e., the values of *STATUS* and *E-VERDICT* can be retrieved or changed by using the 'dot' notation <*identifier*>.*STATUS* and <*identifier*>.*E-VERDICT*. The <*identifier*> in the 'dot' notation refers to the unique identifier of an entity.

The *CONTROL-STACK* and *VALUE-STACK* of an entity state can be addressed by using the 'dot' notation <*identifier*>.*CONTROL-STACK* and <*identifier*>.*VALUE-STACK*.

CONTROL-STACK and *VALUE-STACK* can be accessed and manipulated by using the stack operations *push*, *pop*, *top*, *clear* and *clear-until*.

NOTE: The stack operations *push*, *pop*, *top*, *clear* and *clear-until* have the following meaning:

- <*stack*>.*push*(<*item*>) pushes <*item*> onto <*stack*>;
- <*stack*>.*pop*() pops the top item from <*stack*>;
- <*stack*>.*top*() returns the top element of <*stack*> or **`NULL`** if <*stack*> is empty;
- <*stack*>.*clear*() clears <*stack*>, i.e., pops all items from <*stack*>;

- <*stack*>.*clear-until*(<*item*>) pops items from <*stack*> until <*item*> is found or <*stack*> is empty.

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

NEW-ENTITY (<entity-identifier>, <*flow-graph-node-reference*>)

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

- <entity-identifier> is the unique identifier;
- *STATUS* is set to **ACTIVE**;
- <flow-graph-node-reference> is the only (top) element in *CONTROL-STACK*;
- *data state* and *timer state* are empty lists;
- *VALUE-STACK* is an empty stack;
- *E-VERDICT* is set to **none**.

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:

```
<identifier>.NEXT-CONTROL(boolean <bool>) {
    FlowGraphNodeType successorNode := <identifier>.CONTROL-STACK.NEXT(<bool>).top();
    <identifier>.CONTROL-STACK.pop();
    <identifier>.CONTROL-STACK.push(successorNode).
}
```

B.3.3.2.2 Data state and variable binding

As shown in figure B.14 a *data 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.

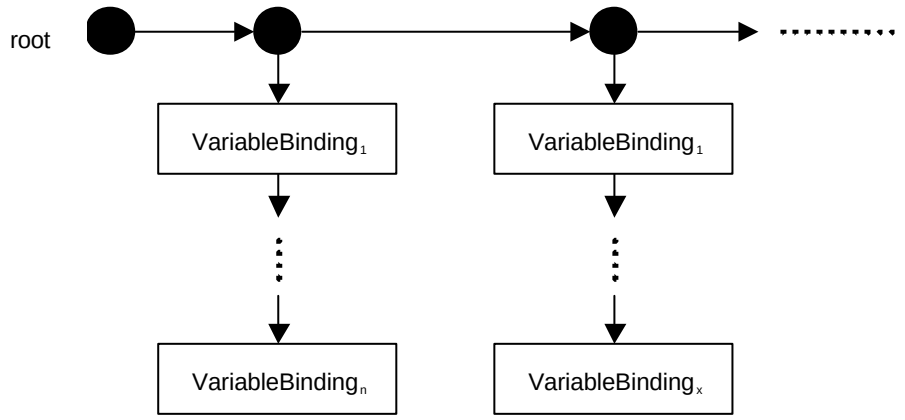


Figure B.14: Structure of the data state part of an entity state

The structure of a variable binding is shown in figure B.15. A variable has a name <*var-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.

Figure B.15: 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.

B.3.3.2.3 Timer state and timer binding

As shown in figure B.16 and figure B.17 a *timer state* and a *data state* in 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.

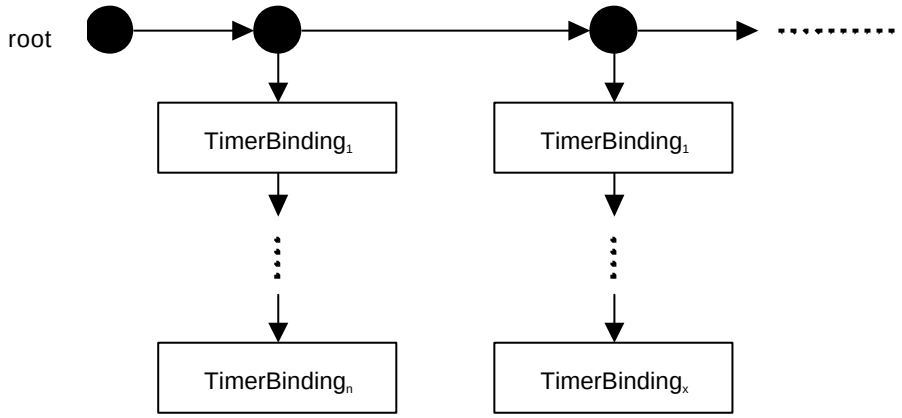


Figure B.16: Structure of the timer state part of an entity state

The structure of a timer binding is shown in figure B.17. The meaning of *<timer-name>* and *location* is similar to the meaning of *<var-name>* and *location* for a variable binding (figure B.15).

<timer-name>	**location**	*STATUS*	*DEF-DURATION*	*ACT-DURATION*	*TIME-LEFT*

Figure B.17: 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 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.

Timer can be only passed by reference into functions, i.e., the mechanism is similar to the mechanism for variables described in clause B.3.3.2.2. This means a new timer binding (with the formal parameter name as <*timer-name*>) is created which gets copies of *location*, *STATUS*, *DEF-DURATION*, *ACT-DURATION* and *TIME-LEFT* from the timer that is passed in by reference. When updating <*timer-name*> all timer bindings with the same *location* are updated at the same time.

B.3.3.2.4 Accessing timer and data states

The *value* of a variable can be retrieved by using the dot notation <*identifier*>.<*var-name*> where <*identifier*> refers to the unique *identifier* of an entity. For changing the value of a variable, the *VAR-SET* function has to be used:

<*identifier*>.*VAR-SET* (<*var-name*>, <*value*>)

sets the value of variable <*var-name*> in the actual scope of an entity with the unique identifier <*identifier*>. In addition, the value of all variables with the same location as variable <*var-name*> will also be set to <*value*>.

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

<*identifier*>.<*timer-name*>.*STATUS*;

<*identifier*>.<*timer-name*>.*DEF-DURATION*;

<*identifier*>.<*timer-name*>.*ACT-DURATION*;

<*identifier*>.<*timer-name*>.*TIME-LEFT.*

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

<*identifier*>.*TIMER-SET*(<*timer-name*>, *STATUS*, <*value*>)

sets the *STATUS* value of timer <*timer-name*> in the actual scope of an entity with the unique identifier <*identifier*> to the value <*value*>. In addition, the *STATUS* of all timers with the same location as timer <*timer-name*> will also be set to <*value*>. The TIMER-SET function can also be used to change the values of *DEF-DURATION*, *ACT-DURATION* and *TIME-LEFT.*

For the handling of variables, timers and scope units the following functions have to be defined:

a) The *INIT-VAR* function: <*identifier*>.*INIT-VAR* (<*var-name*>, <*value*>)

   creates a new variable binding for a variable <*var-name*> with the initial value <*value*> in the actual scope unit of an entity with the unique identifier <*identifier*>. Using the keyword **NONE** as <*value*> means that a variable with undefined initial value is created.

b) The *INIT-TIMER* function: <*identifier*>.*INIT-TIMER* (<*timer-name*>, <*duration*>)

   creates a new timer binding for a timer <*timer-name*> with the default duration <*duration*> in the actual scope of an entity with the unique identifier <*identifier*>. Using the keyword **NONE** as <*duration*> means that a timer without default duration is created.

c) The *GET-VAR-LOC* function: <*identifier*>.*GET-VAR-LOCATION* (<*var-name*>)

   retrieves the location of variable <*var-name*> owned by an entity with the unique identifier <*identifier*>

d) The *GET-TIMER-LOC* function: <*identifier*>.*GET-TIMER-LOCATION* (<*timer-name*>)

   retrieves the location of timer <*timer-name*> owned by an entity with the unique identifier <*identifier*>

e) The *INIT-VAR-LOC* function: <*identifier*>.*INIT-VAR-LOC* (<*var-name*>, <*location*>)

   creates a new variable binding for a variable <*var-name*> with the location <*location*> in the actual scope unit of an entity with the unique identifier <*identifier*>. The variable will be initialized with the value of another variable with the location <*location*>.

NOTE 1: Variables with the same location are a result of parameterization by reference. Due to the handling of reference parameters as described in clause B.3.3.2.2 all variables with the same location will have identical values during their lifetime.

f) The *INIT-TIMER-LOC* function: <*identifier*>.*INIT-TIMER-LOC* (<*timer-name*>, <*location*>)

creates a new timer binding for a timer <*timer-name*> with the location <*location*> in the actual scope unit of an entity with the unique identifier <*identifier*>. 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 2: Timers with the same location are a result of parameterization by reference. Due to the handling of timer reference parameters as described in clause B.3.3.2.3 all timers with the same location will have identical values for *STATUS*, *DEF-DURATION*, *ACT-DURATION* and *TIME-LEFT* during their lifetime.

g) The *INIT-VAR-SCOPE* function: <*identifier*>.*INIT-VAR-SCOPE* ()

initializes a new variable scope in the data state of entity with the unique identifier <identifier>, i.e., an empty list is appended as first list in the list of lists of variable bindings.

h) The *INIT-TIMER-SCOPE* function: <*identifier*>.*INIT-TIMER-SCOPE* ()

initializes a new timer scope in the timer state of entity with the unique identifier <identifier>, i.e., an empty list is appended as first list in the list of lists of timer bindings.

i) The *DEL-VAR-SCOPE* function: <*identifier*>.*DEL-VAR-SCOPE* ()

deletes a variable scope of the data state of entity with the unique identifier <identifier>, i.e., the first list in the list of lists of variable bindings is deleted.

j) The *DEL-TIMER-SCOPE* function: <*identifier*>.*DEL-TIMER-SCOPE* ()

deletes a timer scope of the timer state of entity with the unique identifier <identifier>, i.e., the first list in the list of lists of timer bindings is deleted.

B.3.3.3 Port states

Port states are used to describe the actual states of ports. The structure of a port state is shown in figure B.18. The <*port-name*> refers to the port name that is used by the test component <*owner*> that owns the port to identify 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 <owner> and by the port name <port-name> local to <owner>.

The *list of connections* part of a port state keeps track of the connections between the different ports in the test system. The mechanism is explained in clause B.3.3.2.1.

The *queue of values* part of a port state includes the data items that are received at this port but not yet consumed.

<*port-name*>	<*owner*>	*STATUS*	**list of connections**	**queue of values**

Figure B.18: Structure of a port state

B.3.3.3.1 Handling of connections between 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 B.19, *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 local name used by the <*remote-entity*> to address the queue. 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**, <*port-name*>) in the list of connections of a port it indicates that the port is mapped onto the port <*port-name*> in the test system interface.

Figure B.19: Structure of a connection

B.3.3.3.2 Handling of ports states

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

a) The *NEW-PORT* function: *NEW-PORT*(<*owner*>,<*port-name*>)

creates a new port and returns its reference. The new port is owned by <*owner*> and has the name <*port-name*> to the port identified by the test component <*owner*> and the port name <*port-name*>. The status of the new port is **STARTED** and both, the *list of connections* and the *queue of values* are empty.

b) The *GET-PORT* function: *GET-PORT*(<*owner*>, <*port-name*>)

returns a reference to the port identified by the test component <*owner*> that owns the port and the port name <*port-name*>.

c) The *GET-REMOTE-PORT* function: *GET-REMOTE-PORT*(<*owner*>, <*port-name*>, <*remote-entity*>)

returns the reference to the port that is owned by test component <remote-entity> and connected to a port identified by <*owner*> and <*port-name*>. The symbolic address **SYSTEM** is returned, if the remote port is mapped onto a port in the test system interface.

NOTE 1: *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 <*remote-entity*> 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(<owner>,<port-name>).STATUS

e) The *ADD-CON* function: *ADD-CON*(<*owner*>, <*port-name*>, <*remote-entity*>, <remote-port-name>)

adds a connection (<*remote-entity*>, <remote-port-name>) to the list of connections of port <*port-name*> owned by <owner>.

f) The *DEL-CON* function: *DEL-CON*(<*owner*>, <*port-name*>, <*remote-entity*>, <remote-port-name>)

deletes connection (<*remote-entity*>, <remote-port-name>) from the list of connections of port <*port-name*> owned by <owner>.

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* function references the queue that shall be accessed.

NOTE 2: The queue operations *enqueue*, *dequeue*, *first* and *clear* have the following meaning:

- <*queue*>.*enqueue*(<*item*>) puts <*item*> as last item into <*queue*>;
- <*queue*>.*dequeue*() deletes the first item from <*queue*>;
- <*queue*>.*first*() returns the first item in <*queue*> or **NULL** if <*queue*> is empty;

- <*queue*>.*clear*() removes all elements from <*queue*>.

B.3.3.4 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: 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(<entity-identifier>)

deletes an entity with the unique identifier <*entity-identifier*>. The deletion comprises:

- the deletion of the entity state of <*entity-identifier*>;
- deletion of all ports owned by <*entity-identifier*>;
- deletion of all connections in which <*entity-identifier*> is involved.

b) The EXISTING function: EXISTING(<entity-identifier>)

returns *true* if an entity state for the entity identified by <*entity-identifier*> exists. Otherwise it returns *false*.

c) The UPDATE-REMOTE-REFERENCES function:

UPDATE-REMOTE-REFERENCES (<source-entity>, <target-entity>)

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-entity>.

B.3.4 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 the present document *messages*, *procedure calls*, *replies to procedure calls* and *exceptions* are handled on a conceptual level.

B.3.4.1 Messages

Messages are related to asynchronous communication. Values of all (pre- and user-defined) data types can be exchanged among the entities that communicate. As shown in figure B.20, the operational semantics handles a message as structured object that consist of a *sender* and a *value* part. The *sender* part identifies the sender entity of a message and the *value* part defines the message value.

Figure B.20: 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.

B.3.4.2 Procedure calls and replies

Procedure calls and replies to procedures are related to synchronous calls. 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 message call and the structure of a reply are presented in figure B.21 and figure B.23.

The *sender* and the *procedure reference* part 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 B.21 refers to the **`in`** parameters and **`inout`** parameters and the *parameter part* of the reply in figure B.22 refers to the **`inout`** parameters and **`out`** parameters of the procedure to which call and reply belong. In addition, the reply has a *value* part for the return values in the reply to a procedure.

NOTE 1: As stated in the previous note, the operational semantics only presents a model for the concepts of TTCN-3. Whether and how the information described in figure B.21 and figure B.22 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 B.21. For a reply to a procedure call, **`in`** parameters are of no relevance and are omitted in figure B.22.

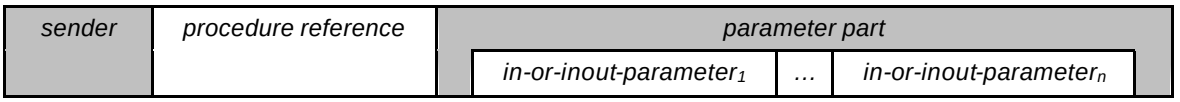


Figure B.21: Structure of a procedure call

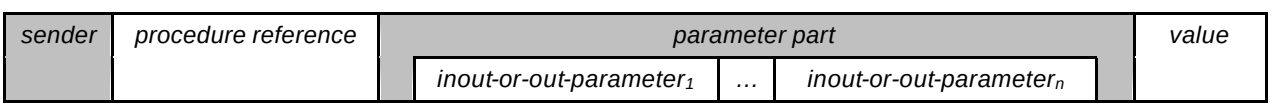


Figure B.22: Structure of a reply to a procedure call

B.3.4.3 Exceptions

Exceptions are also related to synchronous communication. The structure of an exception is shown in figure B.23. It consists of three parts. The *sender* part identifies the sender of the exception; the *procedure reference* part refers to the procedure to which the exception belongs and the *value* part provides the value of the exception. The type of the value of an exception is defined in the signature of the procedure referred to in the procedure reference part. In general it can be of any pre- or user-defined TTCN-3 data type.

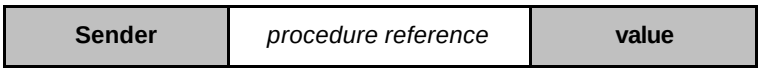


Figure B.23: Structure of an exception

B.3.4.4 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(<sender>, <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*>. The <*sender*> information is also assumed to be part of the item to be sent (figures B.20 to B.23).

B.3.4.5 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(<item-to-check>, <matching-part>, <sender>)*
>
> returns *true* if <*item-to-check*> fulfils the conditions of <*matching-part*> and if <*item-to-check*> has been sent by <*sender*>, otherwise it returns *false*.

B.3.4.6 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 B.3.4.3) 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(<*item-received*>, <*assignment-part*>, <*receiver*>)

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.

B.3.5 Call records for functions and test cases

Functions 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 function or test case definition. The operational semantics handles function calls and calls of test cases by using *call records* as shown in figure B.24. The value of *BEHAVIOUR-ID* is the name of a function or test case, value parameters provide concrete values <$parId_1$> … <$parId_n$> for the formal parameters <*$parId_1$*> … <*$parId_n$*>. Reference parameters provide references to locations of existing variables and timers. Before a function or test case can be executed an appropriate call record has to be constructed.

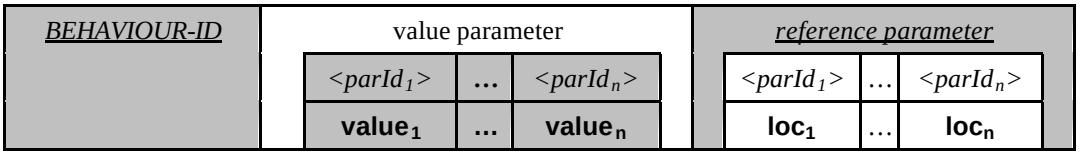


Figure B.24: Structure of a call record

B.3.5.1 Handling of call records

The function or test case name and the actual parameter values can be retrieved by using the dot notation, e.g., <*myRecord*>.<$parId_n$> or <*myRecord*>.*BEHAVIOUR-ID* where <*myRecord*> 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(<behaviour-name>)

creates a new call record for function or test case <behaviour-name> and returns a pointer to the new record. The parameter fields of the new call record have undefined values.

<call-record>.INIT-CALL-RECORD()

creates variables and timers for the handling of value and reference parameters in the actual scope of a function or test case. 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.

B.3.6 The evaluation procedure for a TTCN-3 module

B.3.6.1 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.

B.3.6.1.1 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 B.3.5.1.
- *Entity* := **NULL**; // *Entity* will be used to refer to an entity state. An entity state either // represents module control or a test component.
- *AllEntities*:= **NULL**; // *AllEntities* will be a list of entity states
- *AllPorts*:= **NULL**; // *AllPorts* will be a list of port states
- *MTC*:= **NULL**; // *MTC* will refer to the MTC when a test case is running
- *TC-VERDICT*:= **none**; // *TC-VERDICT* will store the actual test case verdict // when a test case is running
- *DONE*:= 0; // During the execution of a test case *DONE* counts the number // of test components that have terminated.

NOTE: The global variables *AllEntities*, *AllPorts*, *MTC*, *TC-VERDICT* and *DONE* form the module state that is manipulated during the interpretation of a TTCN-3 module.

b) **Creation and initialization of module control**

- *Entity*:= *NEW-ENTITY* (*GET-UNIQUE-ID*(),*GET-FLOW-GRAPH* (<*moduleId*>)); // 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* will be explained in // clause 3.5.1.
- *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
- *AllEntities*.*append*(*Entity*); // The new entity is put into the module state.

B.3.6.1.2 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**;

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: This operational semantics makes no assumptions about time progress and the behaviour of the SUT.

B.3.6.1.3 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**;

b) **Storage:** Store the identifier of the selected entity in the global variable *Entity*.

B.3.6.1.4 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.

B.3.6.2 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*****:

c) *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.

d) *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.

e) *****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 module is meant to be undefined.

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.

B.3.7 Flow graph segment definitions 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 B.3.2.1. 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, 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 make use of the functions defined in the previous parts of clause B.3 and the global variables declared and initialized in the evaluation procedure for TTCN-3 modules (Clause B.3.6). An overall view of all functions and keywords used by the pseudo-code can also be found in clause B.3.7.

B.3.7.1 Alt statement

The flow graph representation of **`alt`** statement in figure B.25 distinguishes between **`alt`** statements that have an **`else`** branch and **`alt`** statements that have no **`else`** branch.

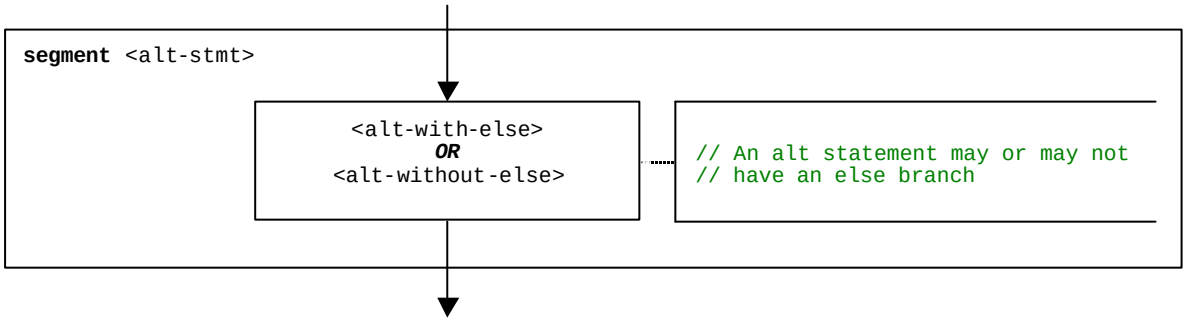


Figure B.25: Flow graph segment <alt-stmt >

The flow graph segments `<alt-with-else>` and `<alt-without-else>` are shown in figure B.26 and figure B.27. The **`else`** branch is a statement block that needs no further explanation. However, both flow graph segments are very similar with the difference that the **`else`** branch provides a definite exit for the **`alt`** statement, whereas an **`alt`** statement without **`else`** branch may loop.

Both flow graph segments have an entry node and beside one incoming flow line, an additional flow line with a label <altId>. This is a symbolic label for the **`alt`** statement. It identifies the target of **`goto alt`** statements and also defines the looping in the `<alt-without-else>` flow graph segment. Both flow graph segments also have a defined exit point by means of the label <altIdExit> and the `alt-exit` node.

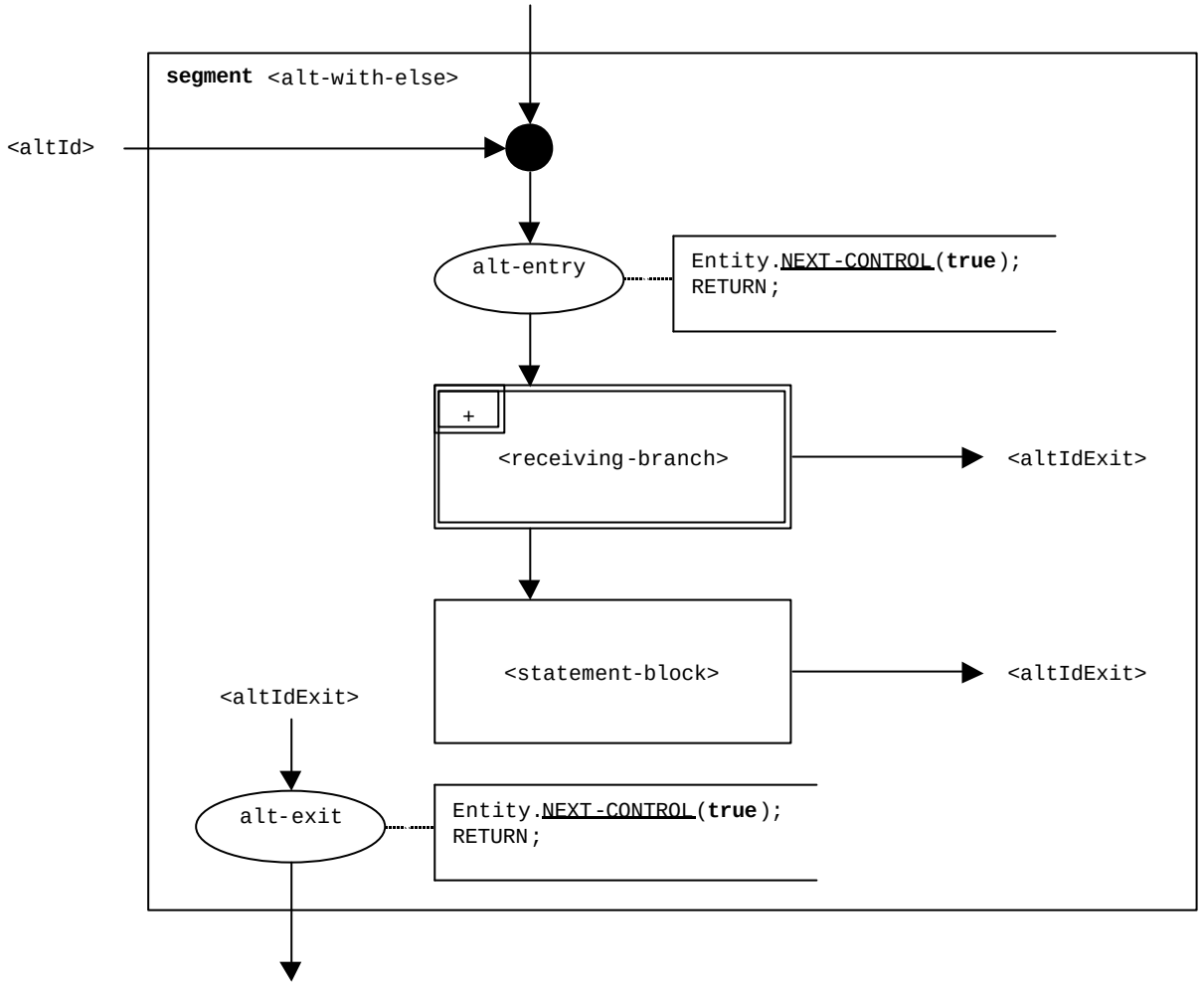


Figure B.26: Flow graph segment <alt-with-else>

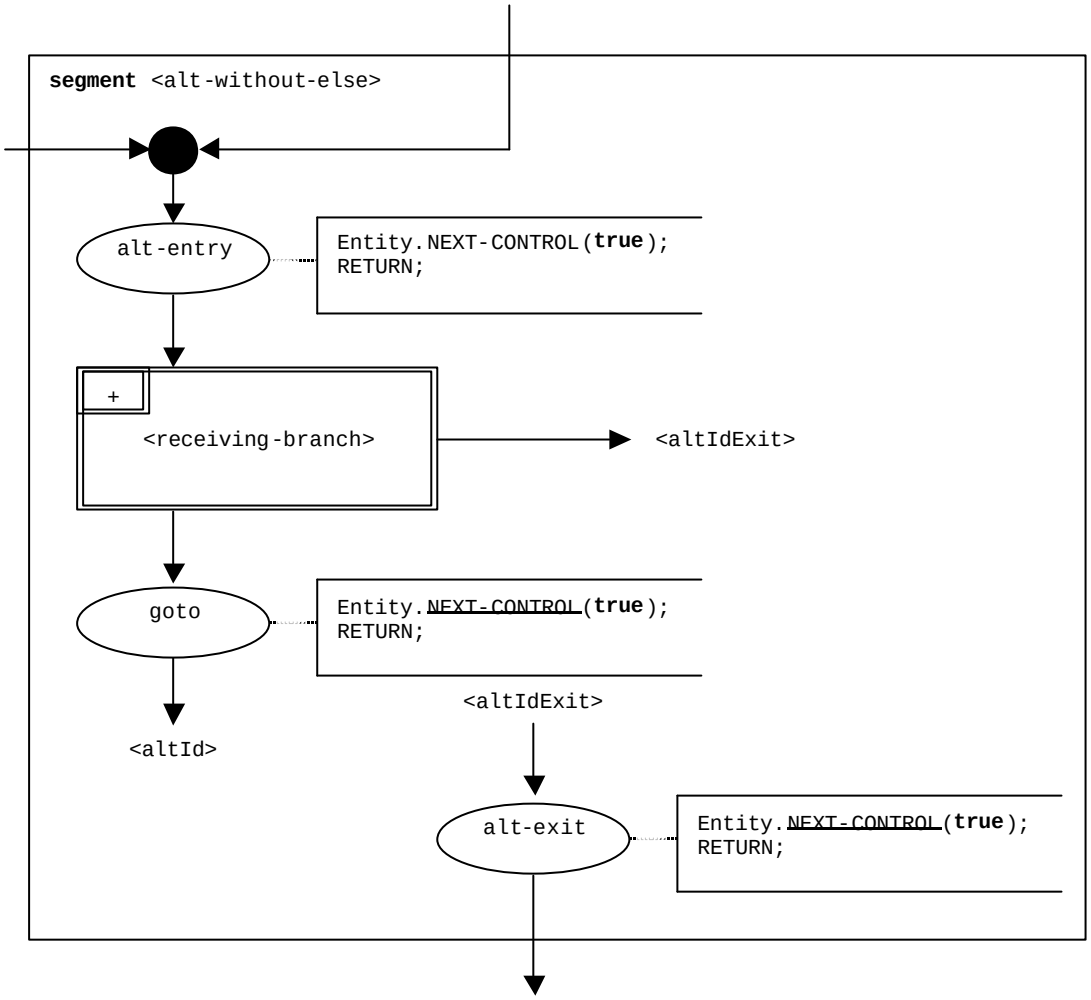


Figure B.27: Flow graph segment <alt-without-else>

B.3.7.1.1 Flow graph segment <receiving-branch>

The execution of the flow graph segment `<receiving-branch>` is shown in figure B.28.

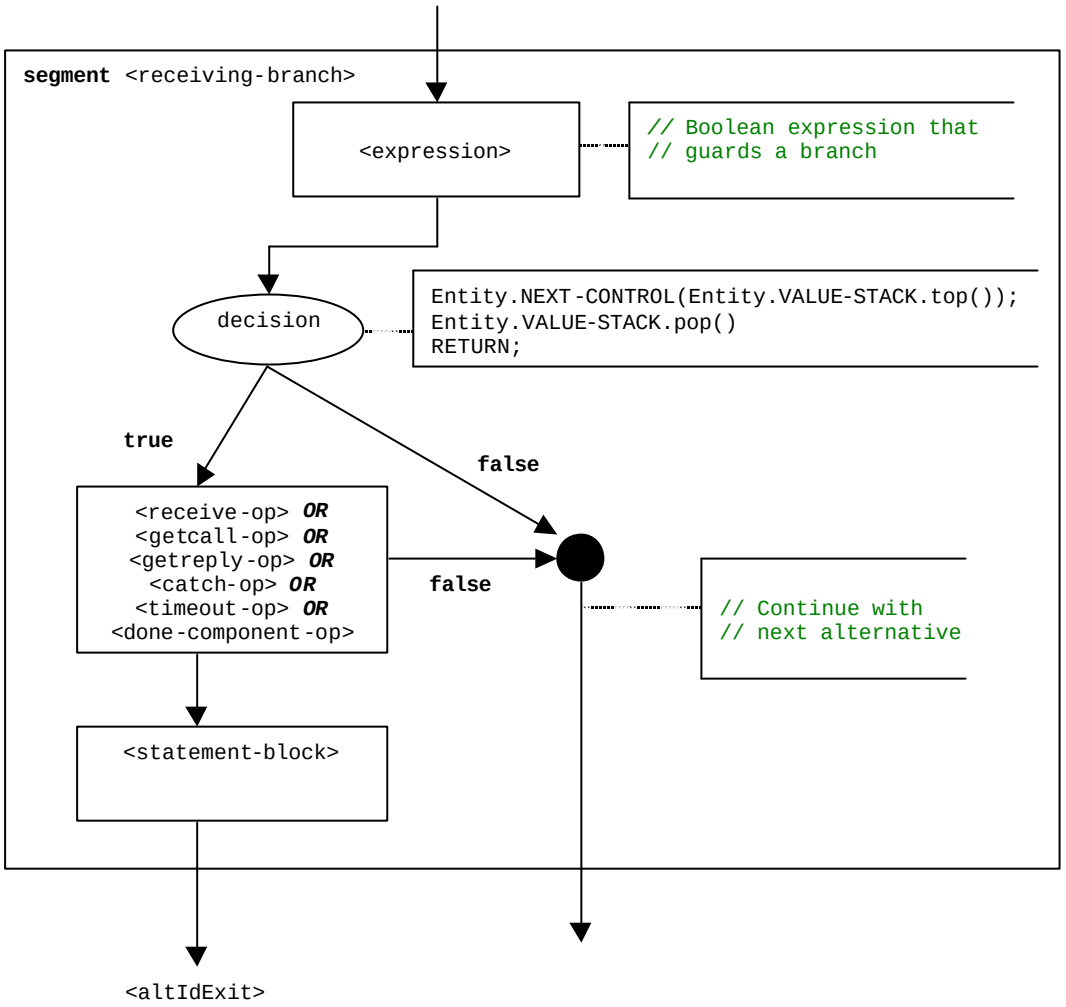


Figure B.28: Flow graph segment <receiving-branch>

B.3.7.2 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 B.29.

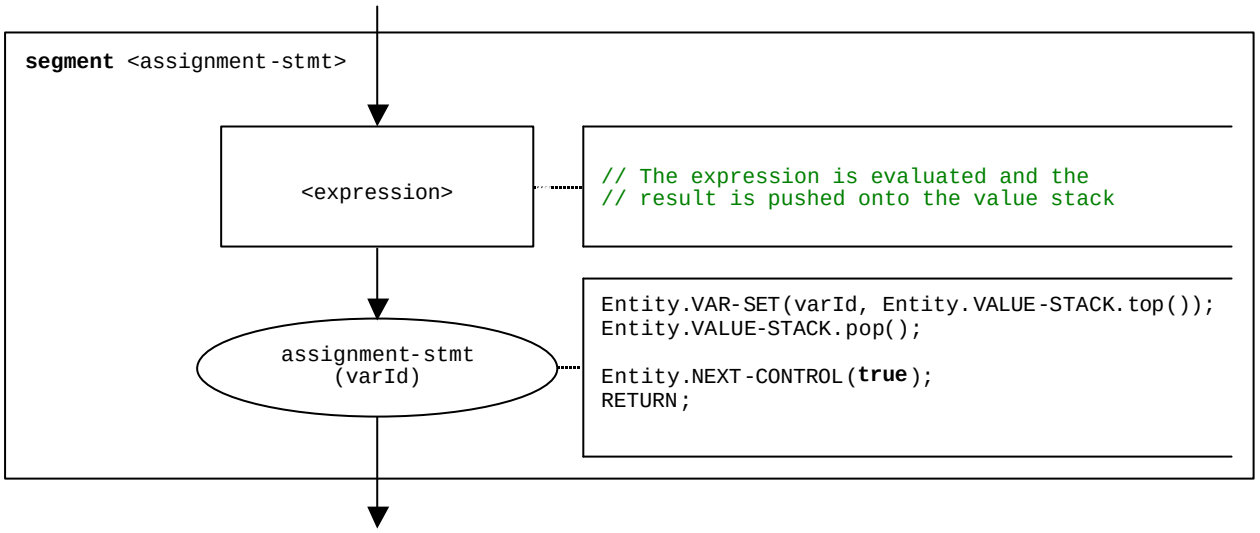


Figure B.29: Flow graph segment <assignment-stmt>

B.3.7.3 Call operation

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

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

The optional <blocking-info> consists of either the keyword **nonblocking** or a duration for a timeout exception. The optional <component_expression> in the **to** clause refers to the receiver entity. It may be provided in form of a variable value or the return value of a function. The optional <call-reception-part> denotes the alternative receptions in case of a blocking **call** operation.

The operational semantics distinguishes between *blocking* and a *non-blocking* **call** operations. A **call** is non-blocking if it expects no replies or if the keyword nonblocking is used. A blocking **call** has a <call-reception-part>.

The flow graph segment <call-op> in figure B.30 defines the execution of a **call** operation. It reflects the distinction between blocking and non-blocking calls.

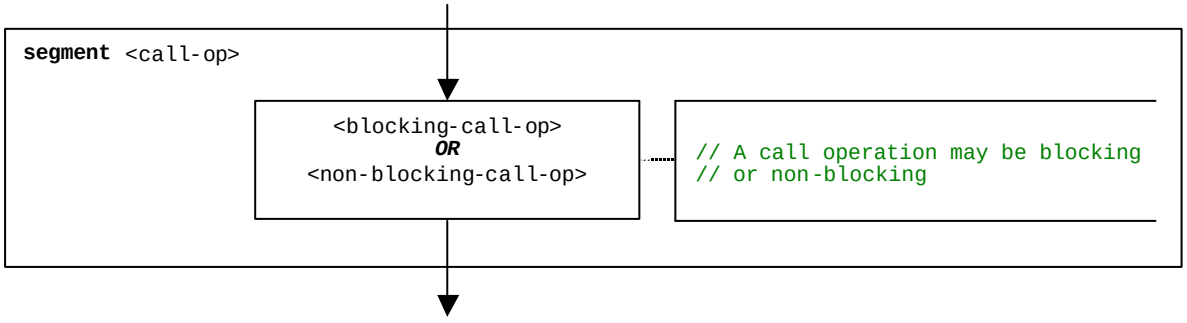


Figure B.30: Flow graph segment <call-op>

For blocking and non-blocking call operations a receiver entity may be specified in form of an expression. The possibilities are shown in figure B.31 and figure B.32.

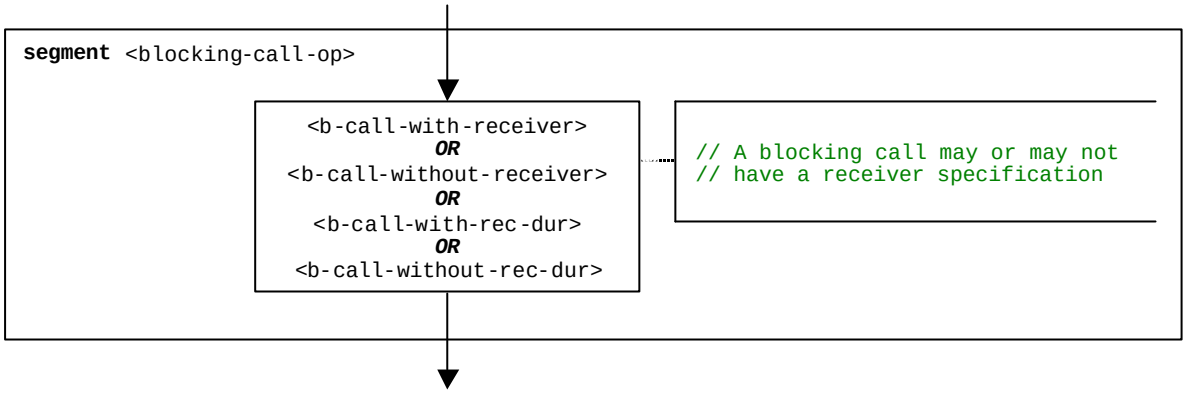


Figure B.31: Flow graph segment <blocking-call-op>

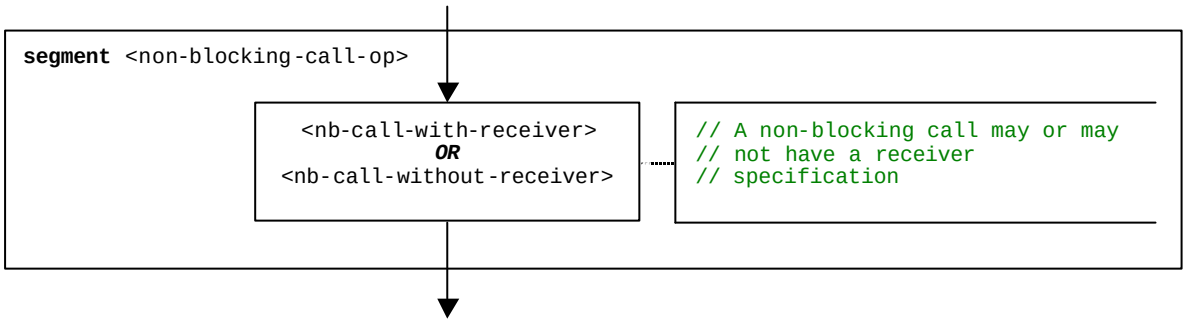


Figure B.32: Flow graph segment <non-blocking-call-op>

B.3.7.3.1 Flow graph segment <nb-call-with-receiver>

The flow graph segment `<nb-call-with-receiver>` in figure B.33 defines the execution of a non-blocking **`call`** operation where the receiver is specified in form of an expression.

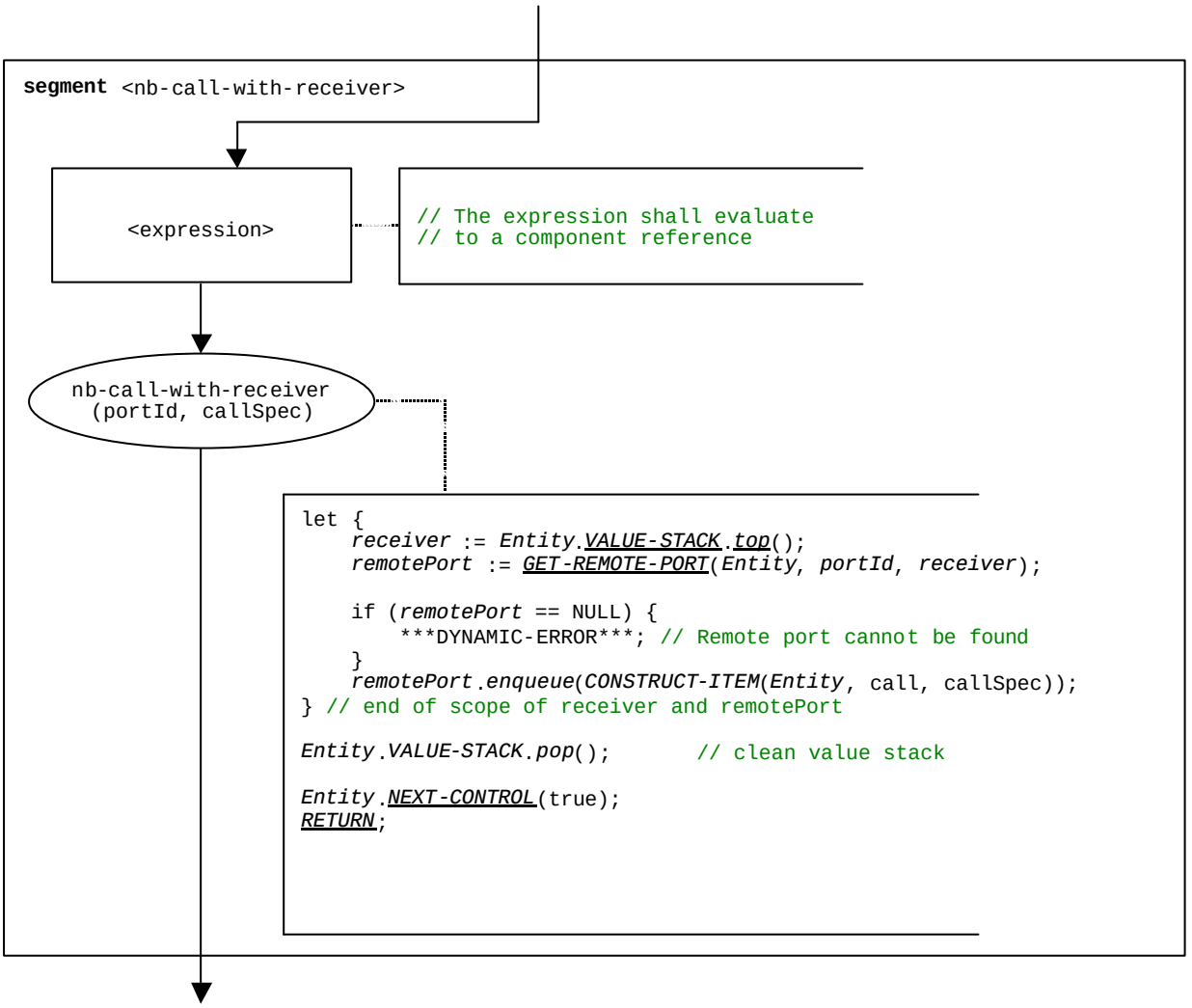


Figure B.33: Flow graph segment <nb-call-with-receiver>

B.3.7.3.2 Flow graph segment <nb-call-without-receiver>

The flow graph segment <nb-call-without-receiver> in figure B.34 defines the execution of a non-blocking **call** operation without a **to**-clause.

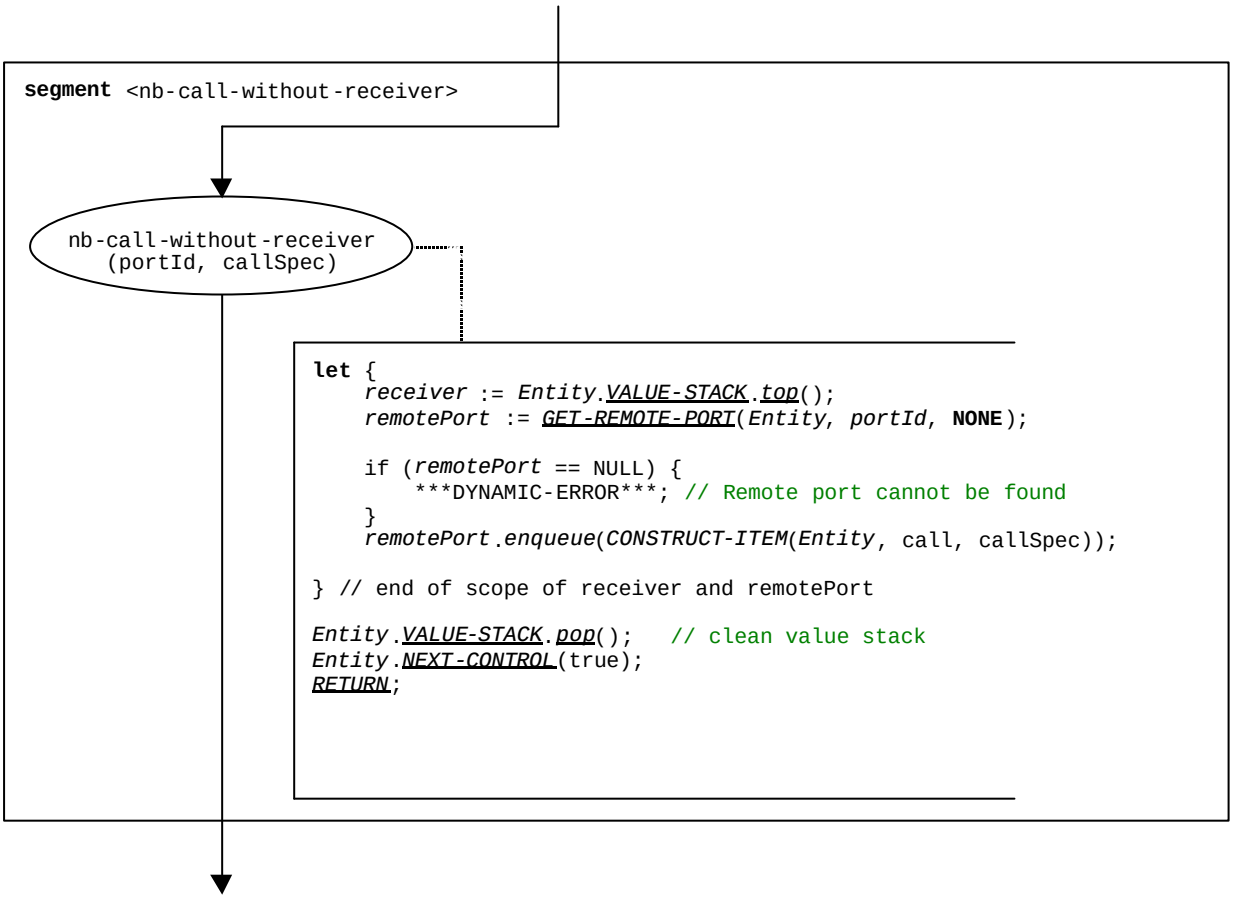


Figure B.34: Flow graph segment <nb-call-without-receiver>

B.3.7.3.3 Flow graph segment <b-call-with-receiver>

Blocking calls are modelled by a non-blocking call followed by an **`alt`** statement. The flow graph segment `<b-call-with-receiver>` describes the execution of a blocking call, without a duration as timer guard, but with a receiver description for the call. The flow graph segment is shown in figure B.35.

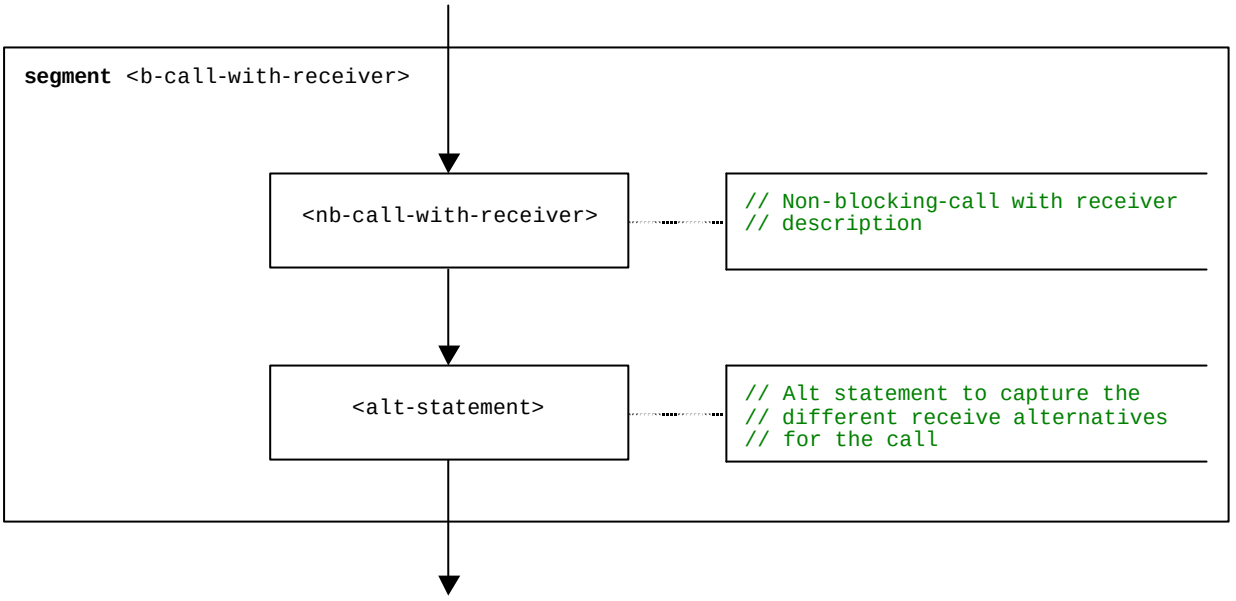


Figure B.35: Flow graph segment <b-call-with-receiver>

B.3.7.3.4 Flow graph segment <b-call-without-receiver>

The flow graph segment `<b-call-without-receiver>` describes the execution of a blocking call, without a duration as timer guard and without a receiver specification for the call. The flow graph segment is shown in figure B.36.

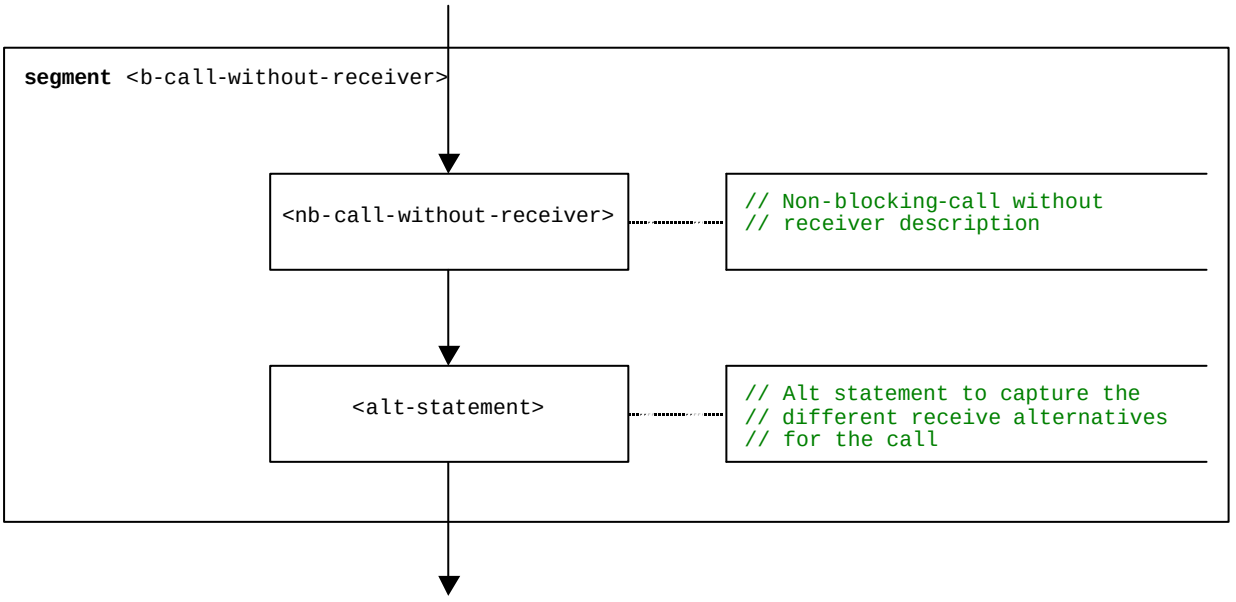


Figure B.36: Flow graph segment <b-call-without-receiver>

B.3.7.3.5 Flow graph segment <b-call-with-rec-dur>

Blocking calls guarded by timers are modelled by a non-blocking call followed by an **alt** statement. For the duration a special system timer SYS-TI is started. The catch timeout branch in the **alt** statement refers to the system timer. The flow graph segment <b-call-with-rec-dur> describes the execution of a blocking call, with a duration as timer guard and a receiver description for the call. The flow graph segment is shown in figure B.37.

NOTE: The handling of the system timer is only handled in an informal manner. The implementation is proprietary to the test equipment.

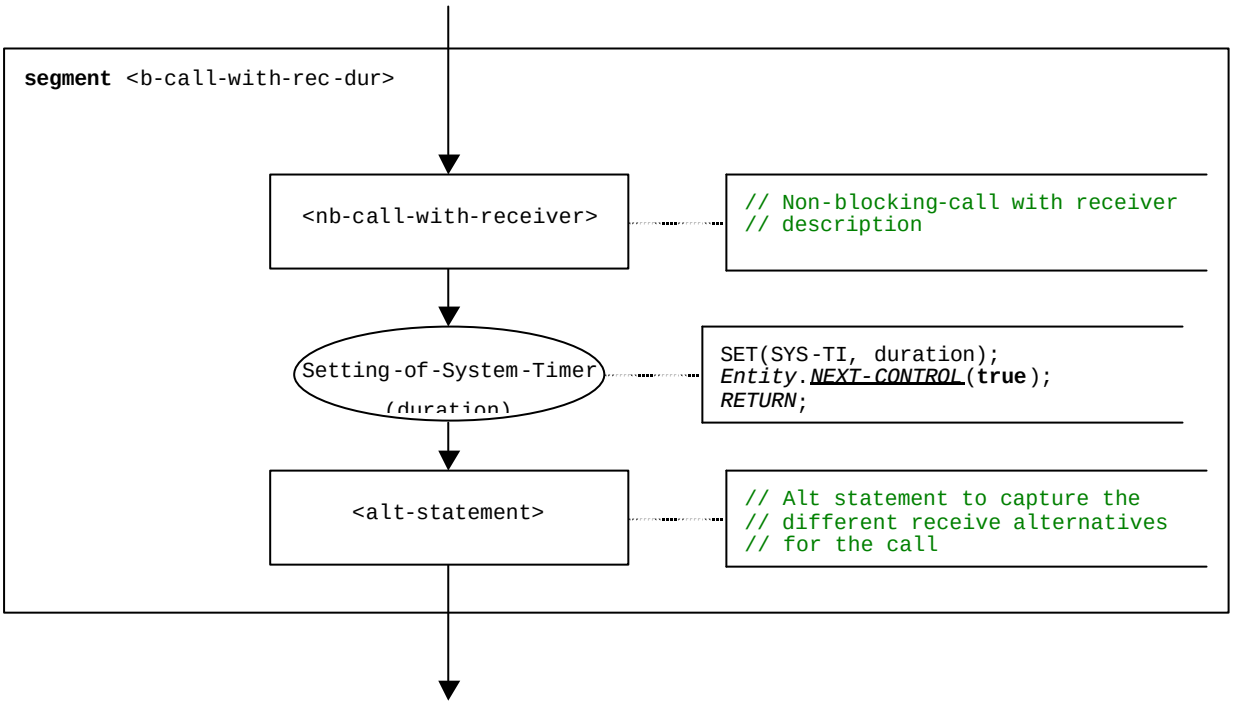


Figure B.37: Flow graph segment <b-call-with-rec-dur>

B.3.7.3.6 Flow graph segment <b-call-without-rec-dur>

The flow graph segment `<b-call-without-rec-dur>` describes the execution of a blocking call, with a duration as timer guard and without a receiver description for the call. The flow graph segment is shown in figure B.38.

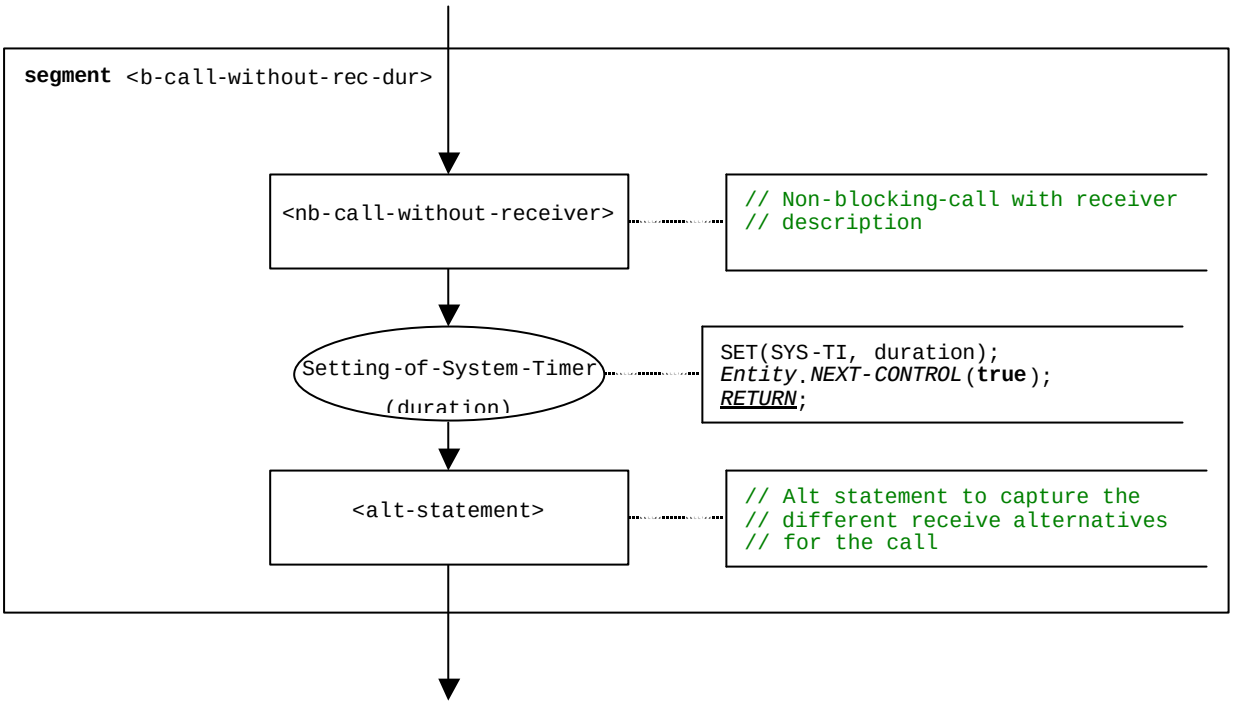


Figure B.38: Flow graph segment <b-call-without-rec-dur>

B.3.7.4 Catch operation

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

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

The optional `<component_expression>` in the **`from`** clause refers to the sender of the exception. 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 catched information if the catched exception matches to the matching specification `<matchingSpec>` and to the (optional) **`from`** clause.

The flow graph segment `<catch-op>` in figure B.39 defines the execution of a **`catch`** operation.

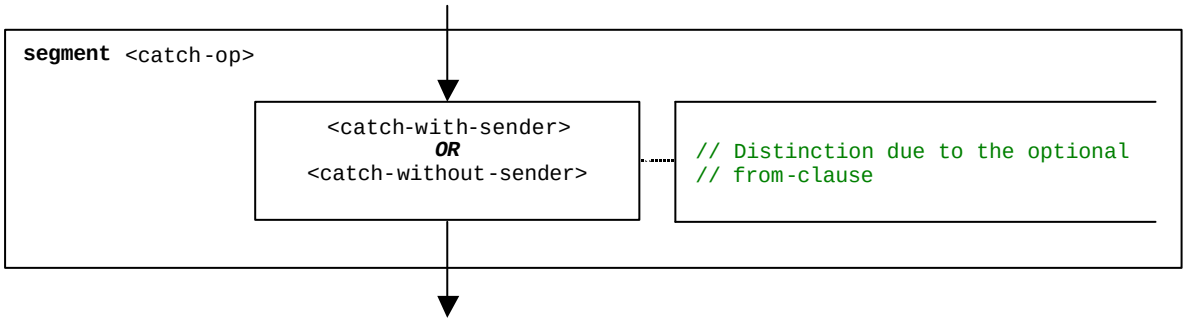


Figure B.39: Flow graph segment <catch-op>

B.3.7.4.1 Flow graph segment <catch-with-sender>

The flow graph segment `<catch-with-sender>` in figure B.40 defines the execution of a **`catch`** operation where the sender is specified in form of an expression.

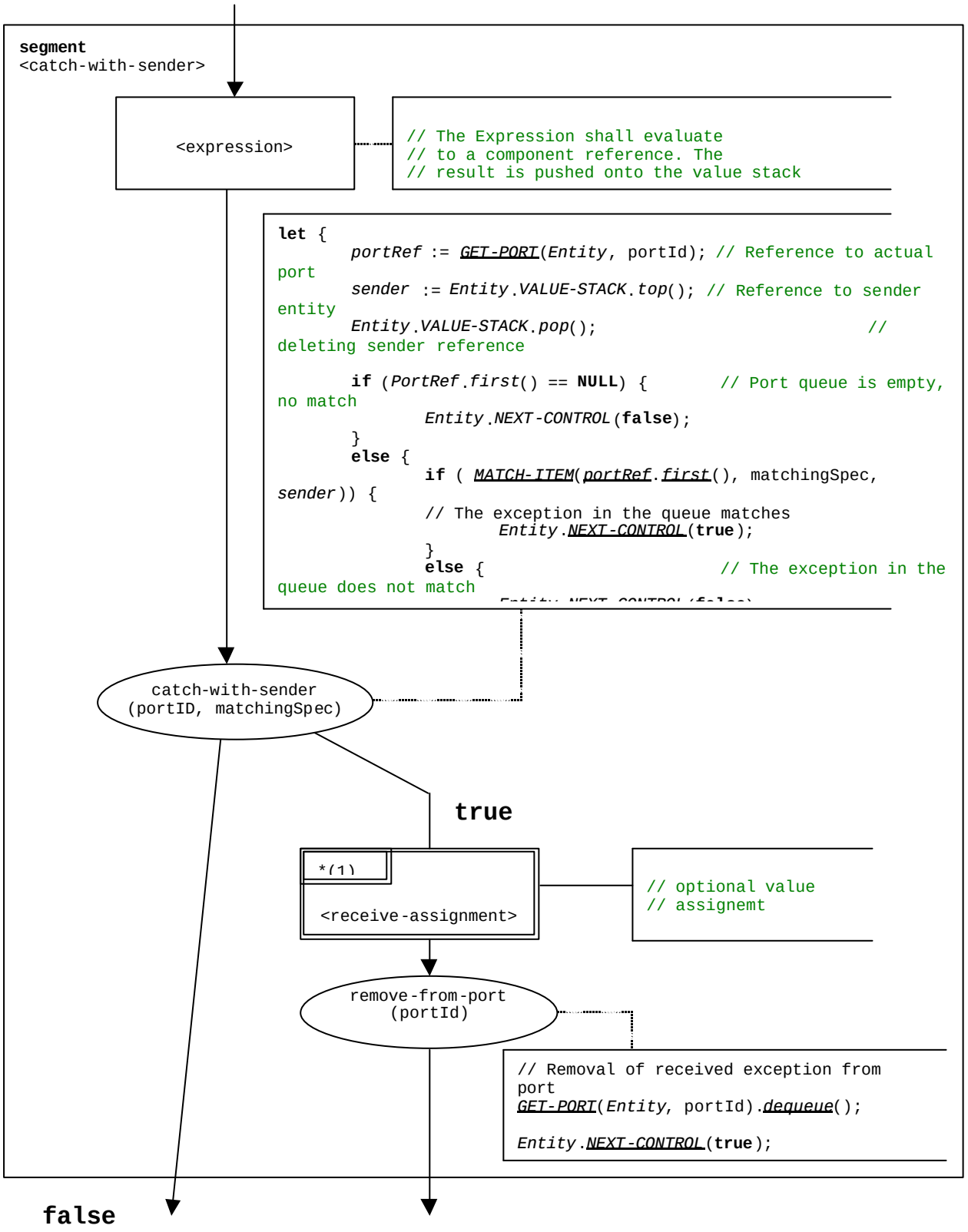


Figure B.40: Flow graph segment <catch-with-sender>

B.3.7.4.2 Flow graph segment <catch-without-sender>

The flow graph segment <catch-with-sender> in figure B.41 defines the execution of a **catch** operation without a **from** clause.

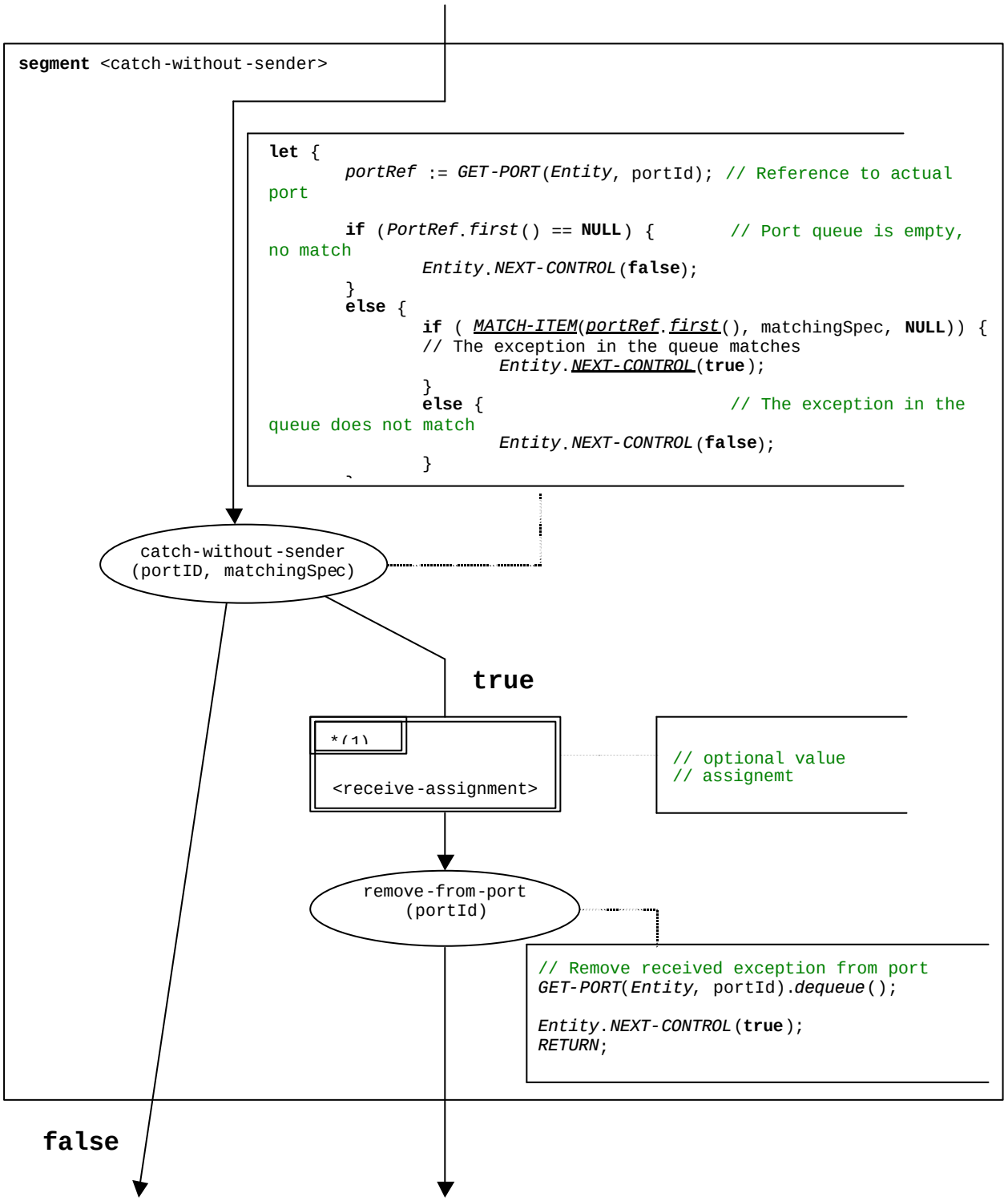


Figure B.41: Flow graph segment <catch-without-sender>

B.3.7.5 Clear port operation

The syntactical structure of the **`clear`** port operation is:

```
<portId>.clear
```

The flow graph segment <clear-port-op> in figure B.42 defines the execution of the **`clear`** port operation.

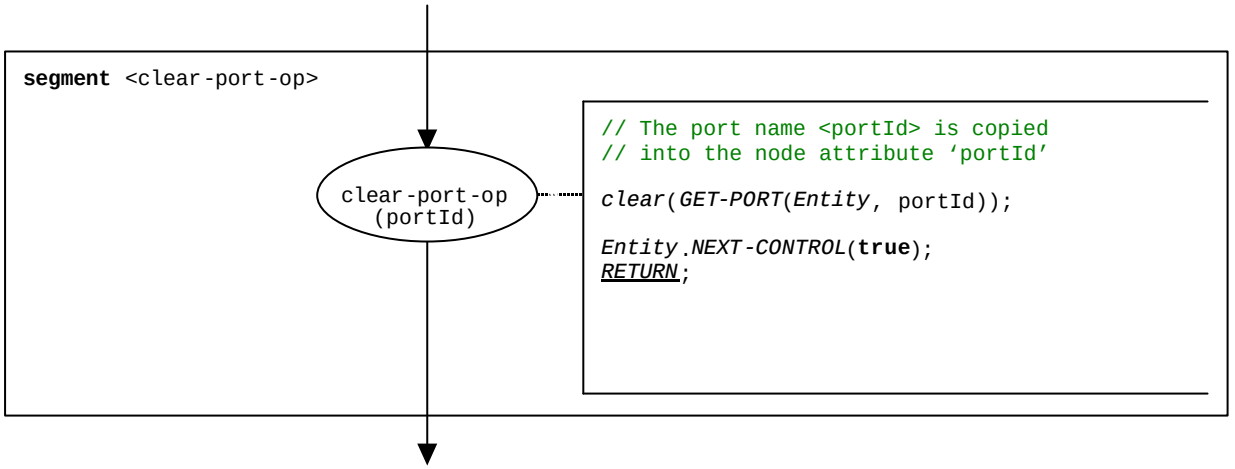


Figure B.42: Flow graph segment <clear-port-op>

B.3.7.6 Connect operation

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

connect(<component_expression$_1$>.<portId1>,<component_expression$_2$>.<portId2>)

The identifiers <portId1> and <portId2> are considered to be port identifiers of the corresponding test components. The components to which the ports belong are referenced by means of the component references <component_expression$_1$> and <component_expression$_2$>. The references may be stored in variables or is returned by a function. For simplicity we consider them as expressions which evaluate to a component reference. Thus, 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 B.43. 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.

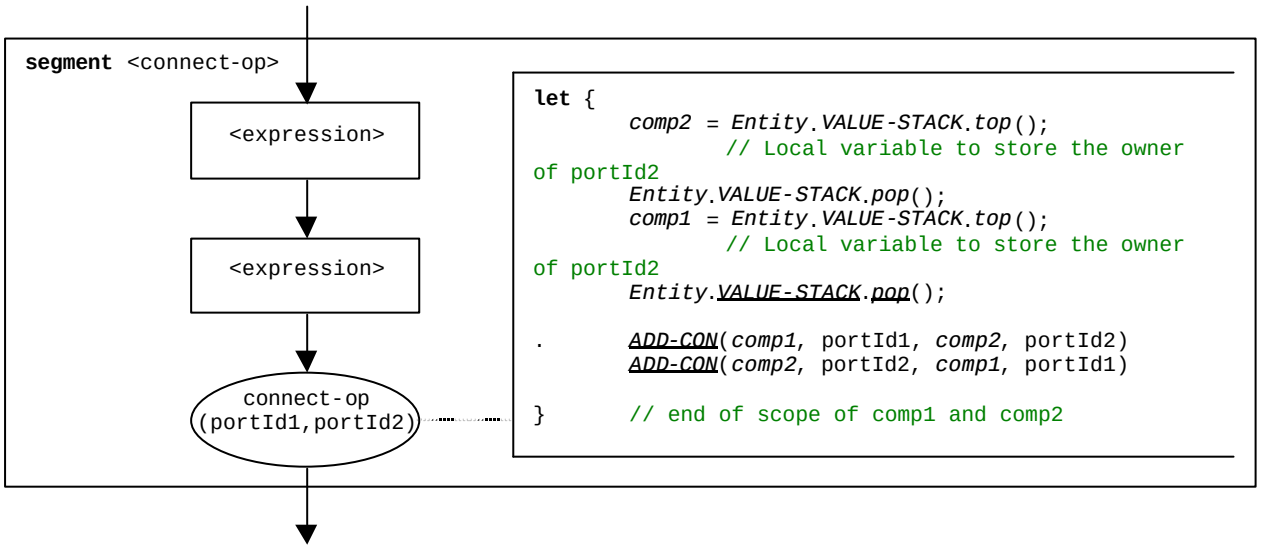


Figure B.43: Flow graph segment <connect-op>

B.3.7.7 Declaration of a constant

The syntactical structure of a constant declaration 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 preprocessing step before this semantics is applied (clause B.2.3). 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-declaration> in figure B.44 defines the execution of a constant declaration where the value of the constant is provided in form of an expression.

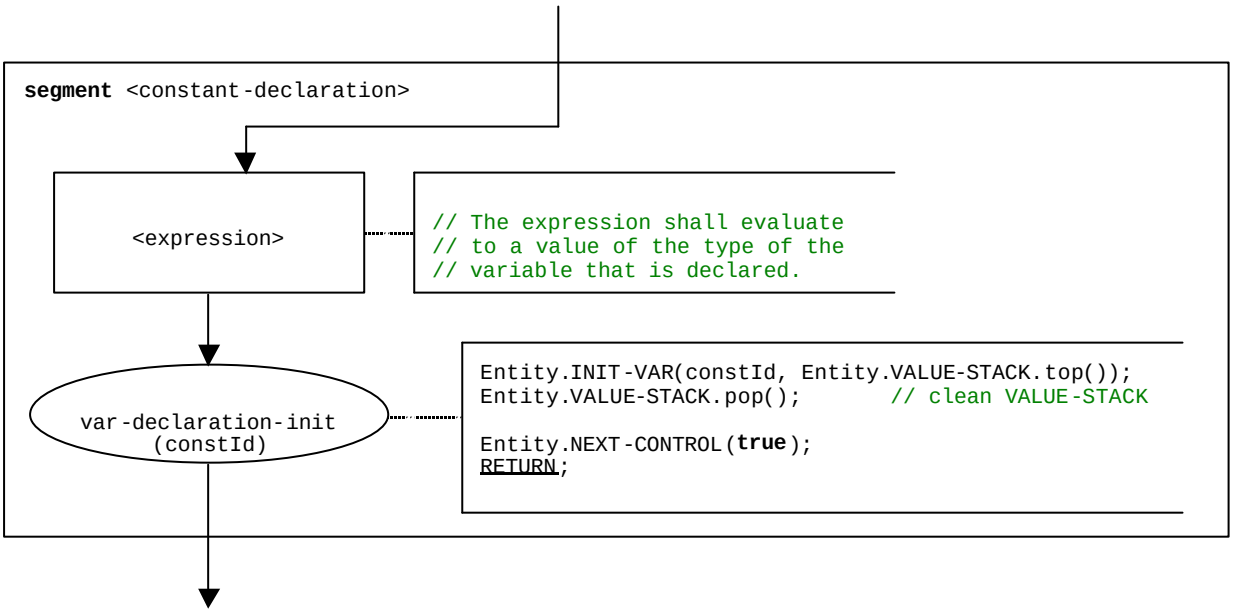


Figure B.44: Flow graph segment <constant-declaration>

B.3.7.8 Create operation

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

```
<componentTypeId>.create
```

The flow graph segment <create-op> in figure B.45 defines the execution of the **`create`** operation.

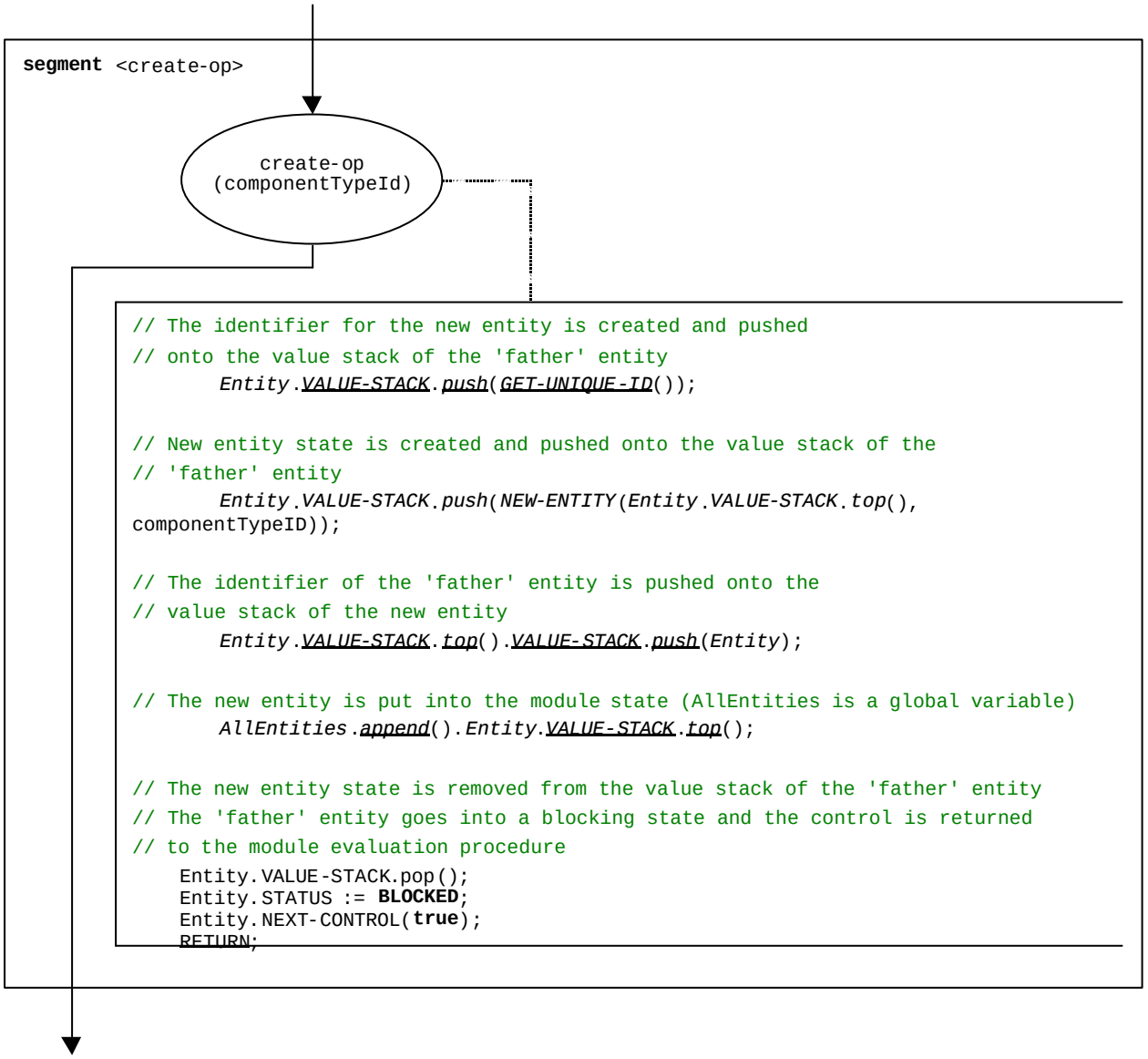


Figure B.45: Flow graph segment <create-op>

B.3.7.9 Declaration of a port

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. The flow graph segment <port-declaration> in figure B.46 defines the execution of a port declaration.

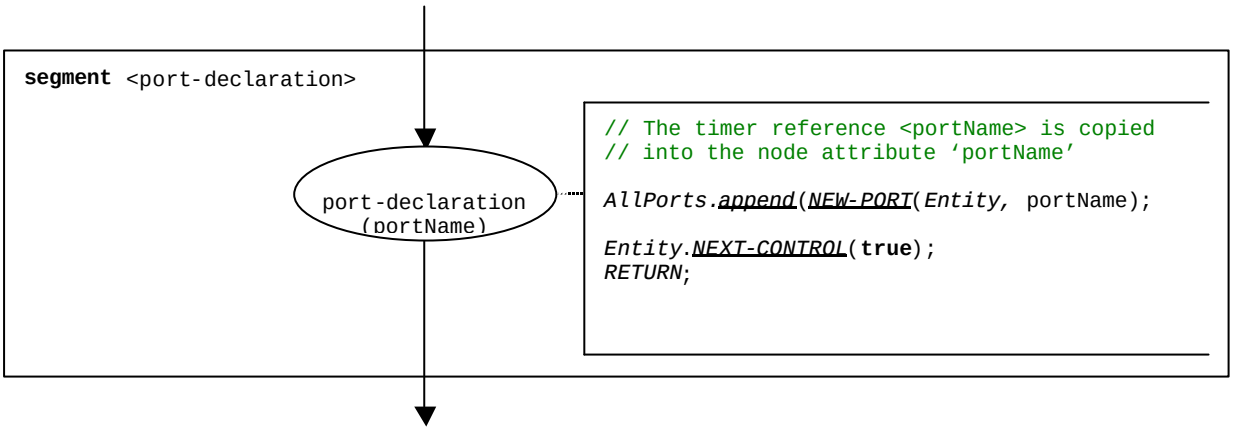


Figure B.46: Flow graph segment <port-declaration>

B.3.7.10 Declaration of a timer

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 variable with 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 B.47 defines the execution of the declaration of a timer.

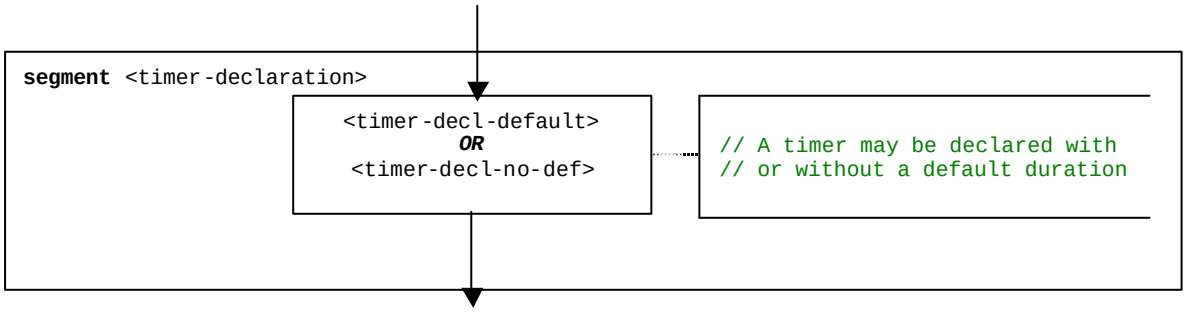


Figure B.47: Flow graph segment <timer-declaration>

B.3.7.10.1 Flow graph segment <timer-decl-default>

The flow graph segment <timer-decl-default> in figure B.48 defines the execution of a timer declaration where a default duration in form of an expression is provided.

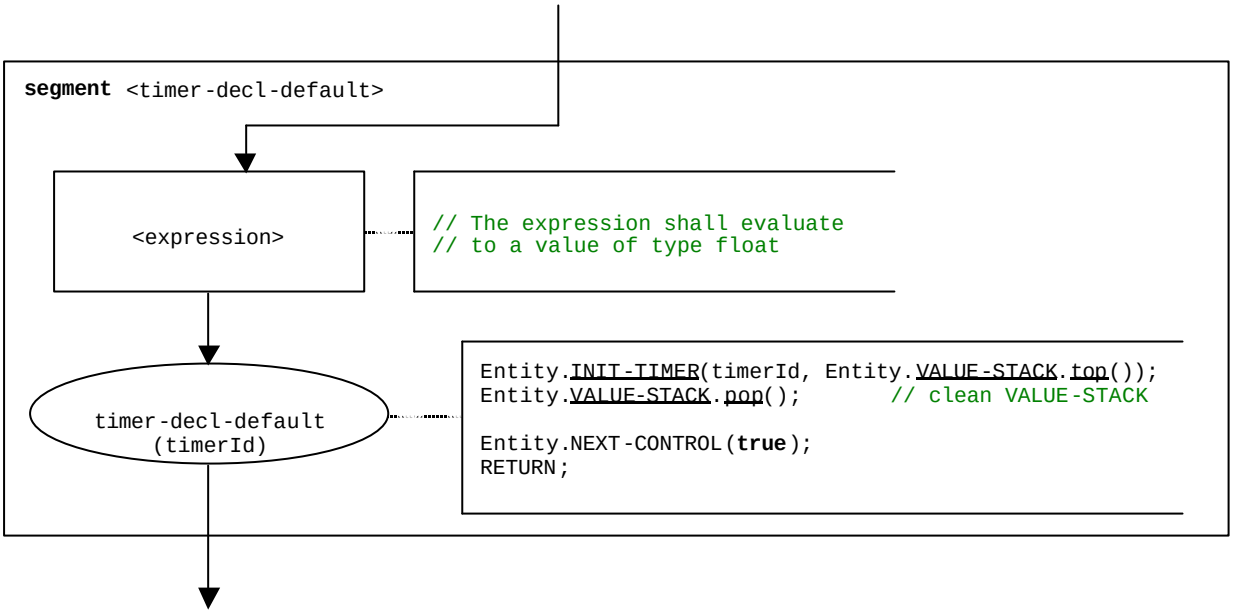


Figure B.48: Flow graph segment <timer-decl-default>

B.3.7.10.2 Flow graph segment <timer-decl-no-def>

The flow graph segment <timer-decl-no-def> in figure B.49 defines the execution of a timer declaration where no default duration is provided, i.e., the default duration of the timer is undefined.

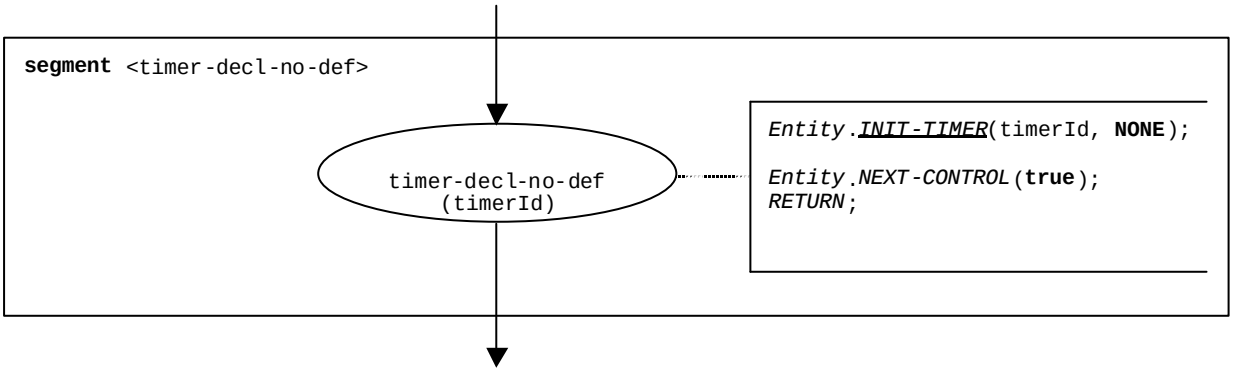


Figure B.49: Flow graph segment <timer-decl-no-def>

B.3.7.11 Declaration of a variable

The syntactical structure of a variable declaration is:

```
var <varType> <varId> [:= <varType_expression>]
```

The initialization of a variable by providing an initial value 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 B.50 defines the execution of the declaration of a variable.

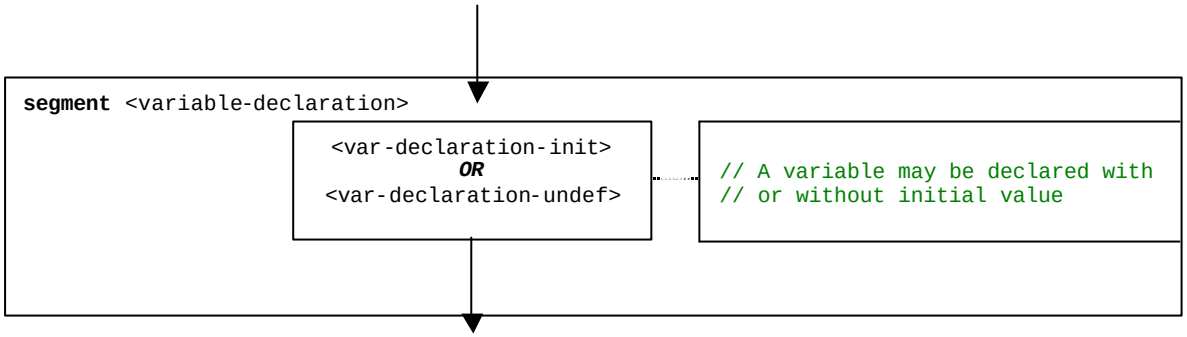


Figure B.50: Flow graph segment <variable-declaration>

B.3.7.11.1 Flow graph segment <var-declaration-init>

The flow graph segment <var-declaration-init> in figure B.51 defines the execution of a variable declaration where an initial value in form of an expression is provided.

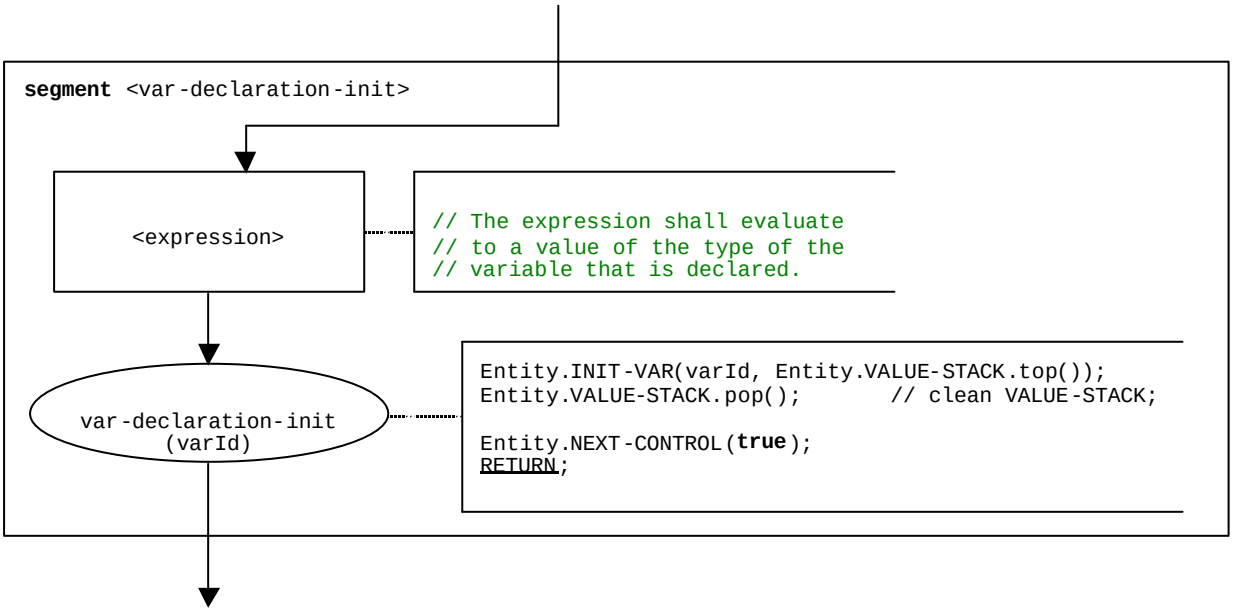


Figure B.51: Flow graph segment <var-declaration-init>

B.3.7.11.2 Flow graph segment <var-declaration-undef>

The flow graph segment <var-declaration-undef> in figure B.52 defines the execution of a variable declaration where no initial value is provided, i.e., the value of the variable is undefined.

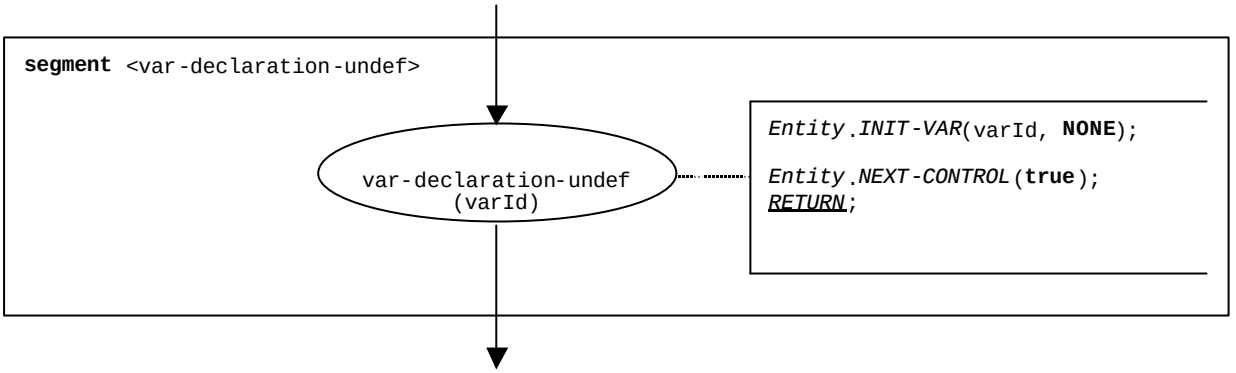


Figure B.52: Flow graph segment < var-declaration-undef >

B.3.7.12 Disconnect operation

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

disconnect(<component_expression$_1$>.<portId1>,<component_expression$_2$>.<portId2>)

The identifiers <portId1> and <portId2> are considered to be port identifiers of the corresponding test components. The components to which the ports belong are referenced by means of the component references <component_expression$_1$> and <component_expression$_2$>. The references may be stored in variables or is returned by a function. For simplicity we consider them as expressions which evaluate to a component reference. Thus, 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 B.53. In the flow graph segment 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 disconnect-op node is executed.

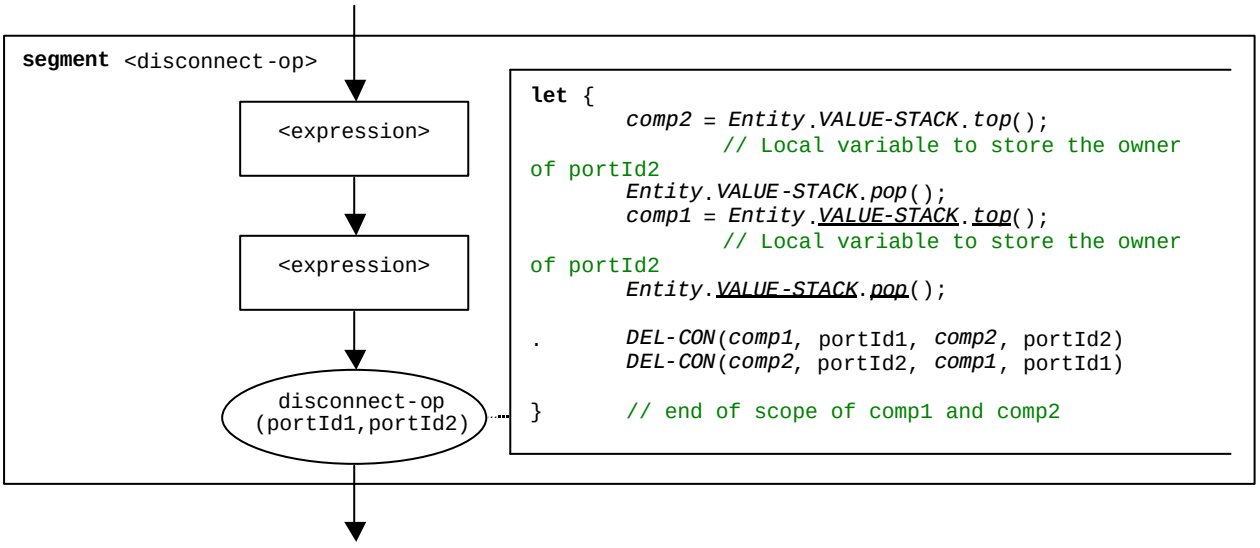


Figure B.53: Flow graph segment <disconnect-op>

B.3.7.13 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 B.54.

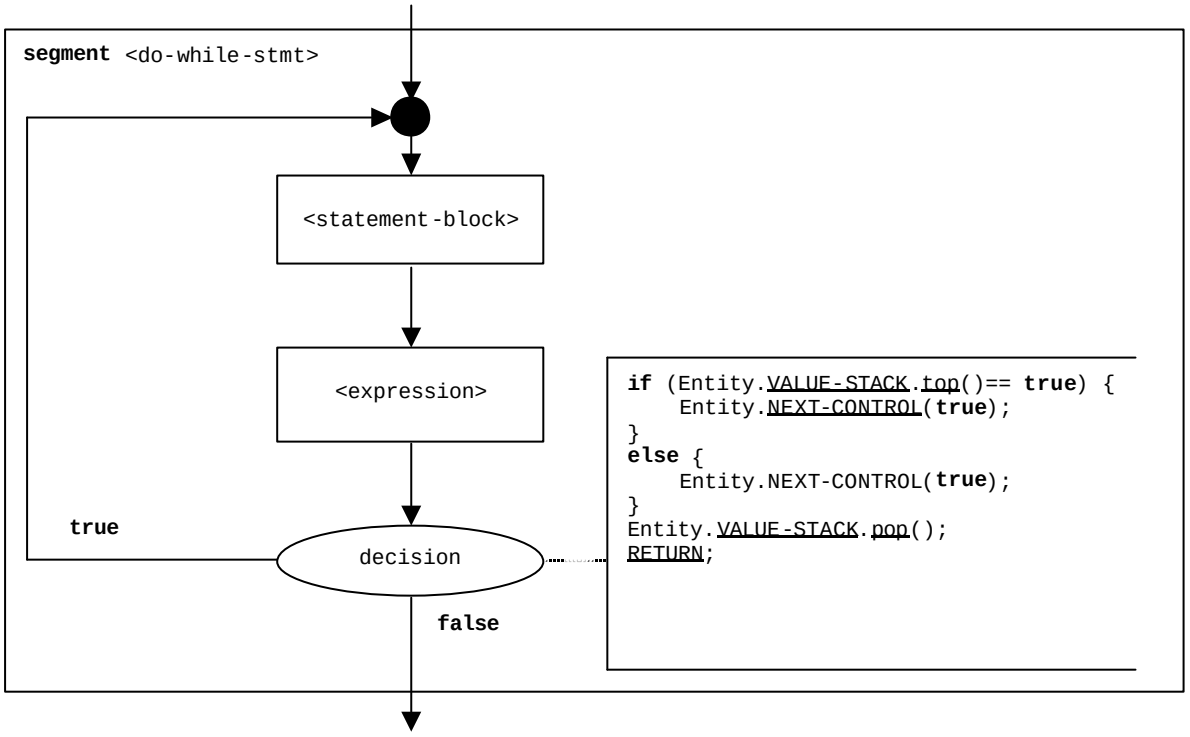


Figure B.54: Flow graph segment <do-while-stmt>

B.3.7.14 Done-all-components operation

The **done-all-components** operation refers to the usage of the keywords **all component** in the **done** operation (Clause B.7.16). The **done-all-components** operation can only be called by the **mtc**. It allows to check whether all parallel test components of a test case have terminated. The syntactical structure of the **done-all-components** operation is:

```
all component.done;
```

The execution of the **done-all-components** operation is defined by the flow graph segment <done-all-comp-op> in figure B.55.

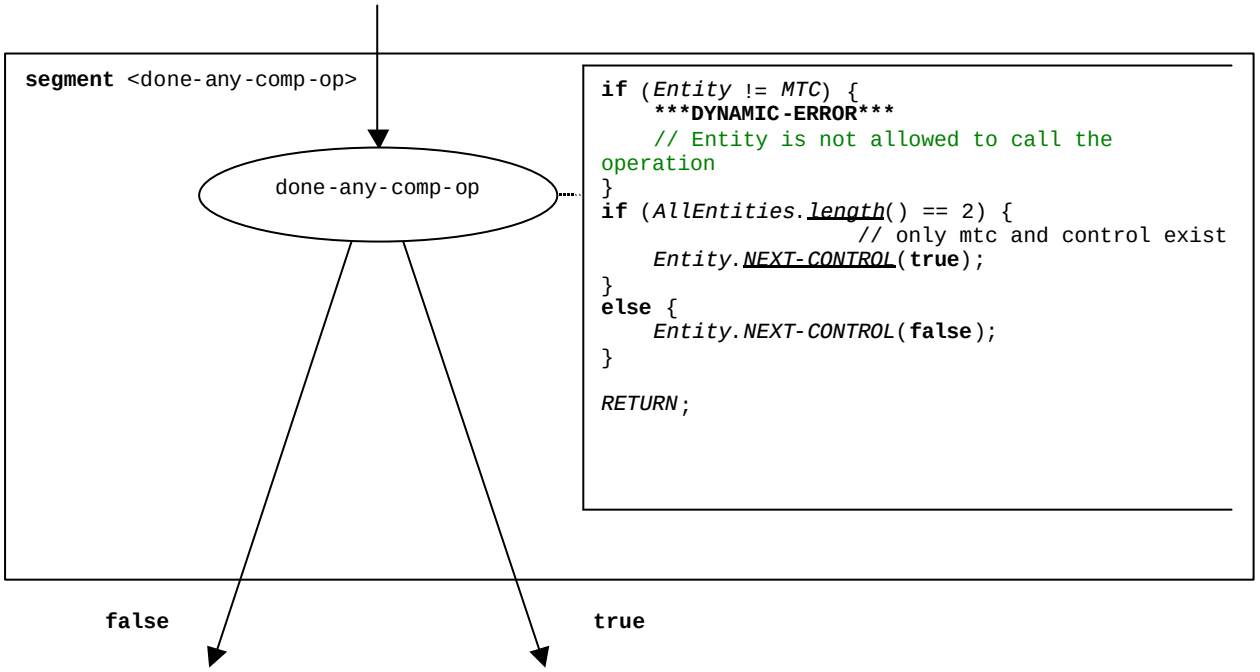


Figure B.55: Flow graph segment <done-all-comp-op>

B.3.7.15 Done-any-component operation

The **done-any-component** operation refers to the usage of the keywords **any component** in the **done** operation (Clause B.7.16). The **done-any-component** operation can only be called by the **mtc**. It allows to check whether a parallel test component of a test case has already terminated. The syntactical structure of the **done-any-component** operation is:

```
any component.done;
```

The execution of the **done-any-component** operation is defined by the flow graph segment <done-any-comp-op> in figure B.56.

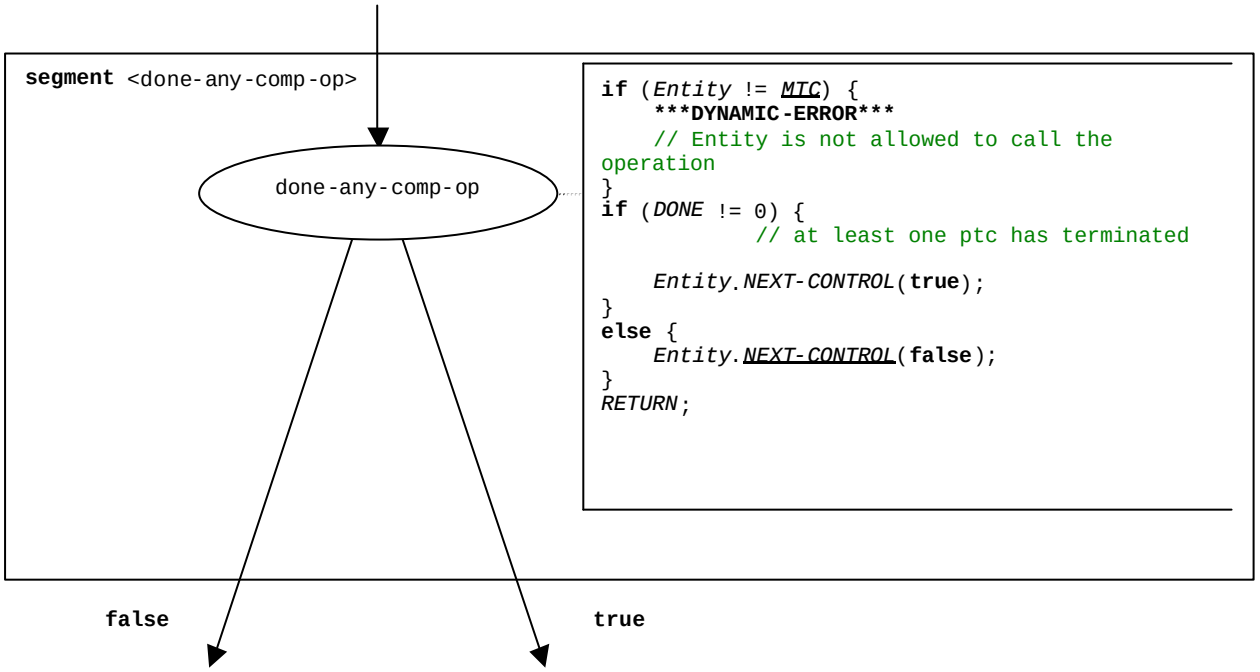


Figure B.56: Flow graph segment <done-any-comp-op>

B.3.7.16 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. For simplicity this is considered to be an expression that evaluates to a component reference.

The flow graph segment <done-component-op> in figure B.57 defines the execution of the **done** component operation.

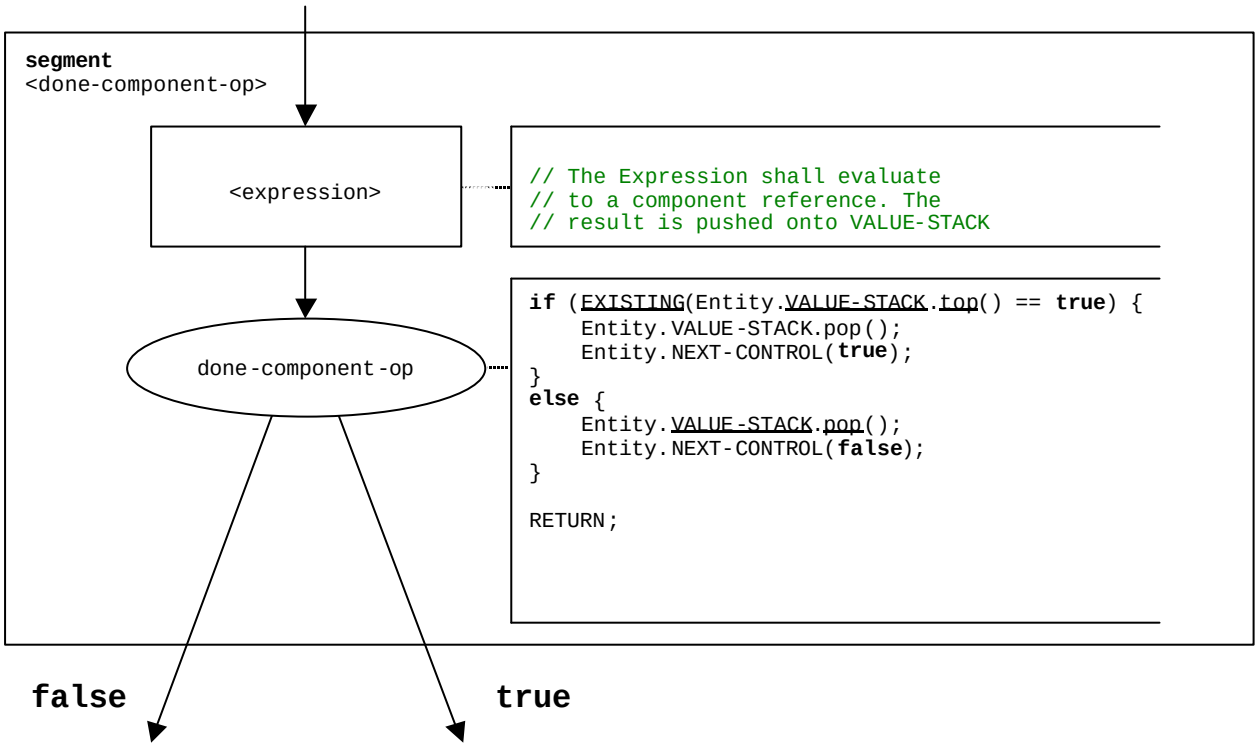


Figure B.57: Flow graph segment <done-component-op>

B.3.7.17 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-par$_1$>, … , <act-par$_n$>. Optionally the execute statement may be guarded by a duration provided in form of an expression that evaluates to a float. If within the specified duration the test case doesn't return a verdict, a timeout exception occurs, the test case is stopped and an **error** verdict is returned. However, TTCN-3 has no real-time semantics and, thus, the decision whether a timeout exception occurs or not is modelled in form of a non-deterministic choice.

NOTE: The operational semantics only models the non-deterministic choice. The <float_expression> is not evaluated.

If due to the non-deterministic choice 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 B.58 defines the execution of an **execute** statement.

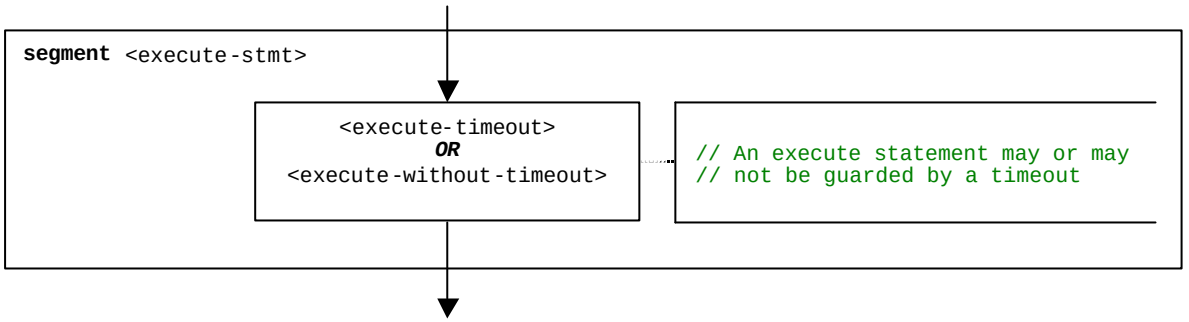


Figure B.58: Flow graph segment <execute-stmt>

B.3.7.17.1 Flow graph segment <execute-timeout>

The flow graph segment <execute-timeout> in figure B.59 defines the execution of an **execute** statement that is guarded by a timeout value.

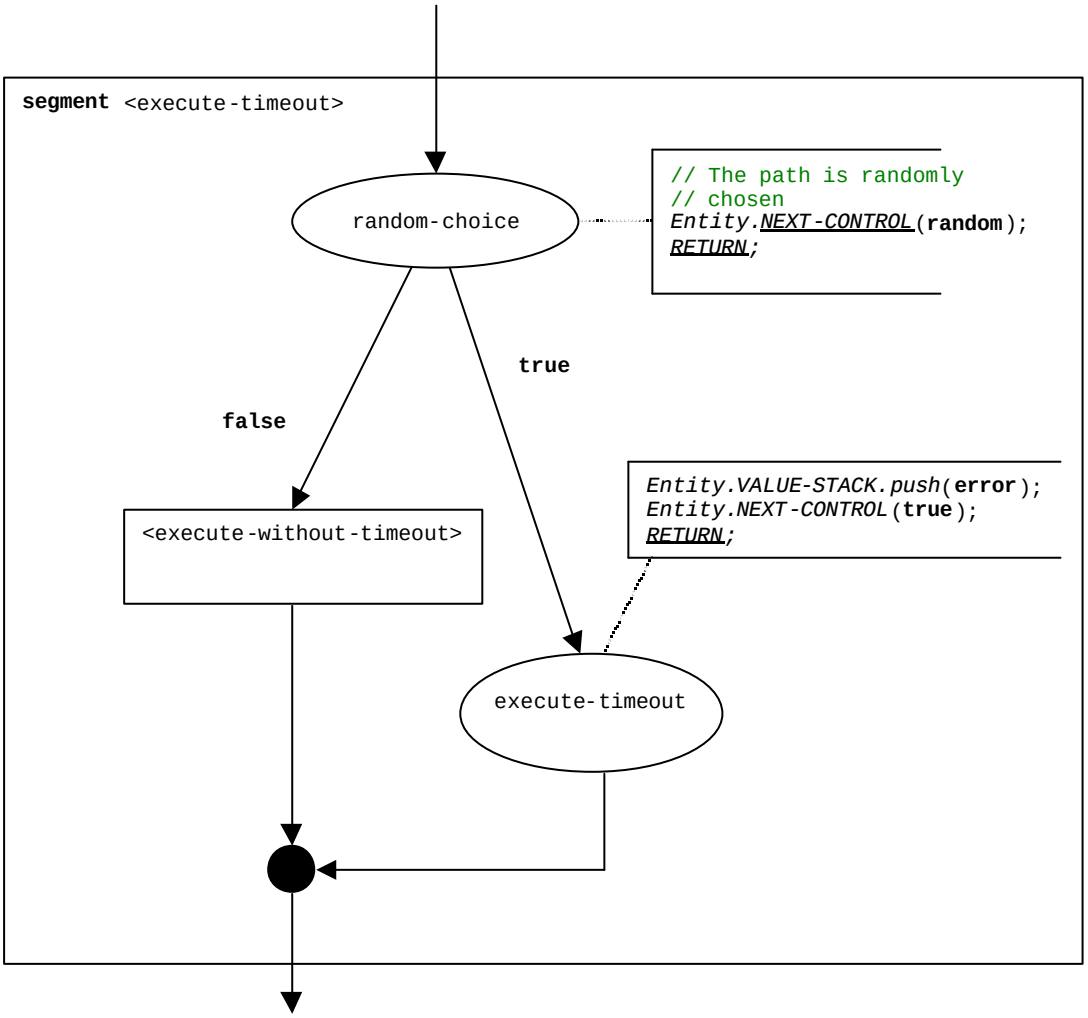


Figure B.59: Flow graph segment <execute-timeout>

B.3.7.17.2 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:

```
mtcType MyMTC := mtcType.create;

MyMTC.start(TestCaseName(P1…Pn);
```

The flow graph segment <execute-without-timeout> in figure B.60 defines the execution of an **execute** statement without the occurrence of a timeout exception by using the flow graph segments of the **create** and the **start** operations.

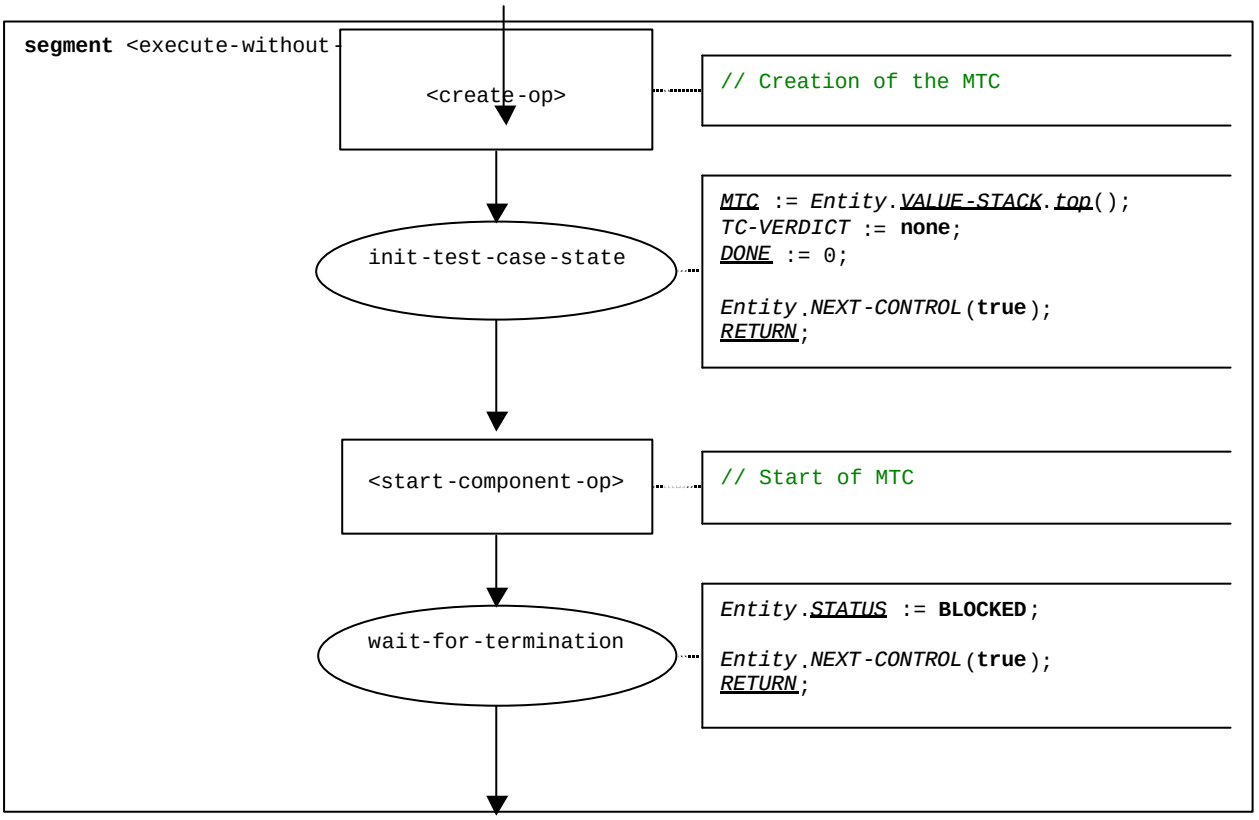


Figure B.60: Flow graph segment <execute-without-timeout>

B.3.7.18 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> | <operand-appl>
```

where:

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

The execution of an expression is defined by the flow graph segment <expression> shown in figure B.61.

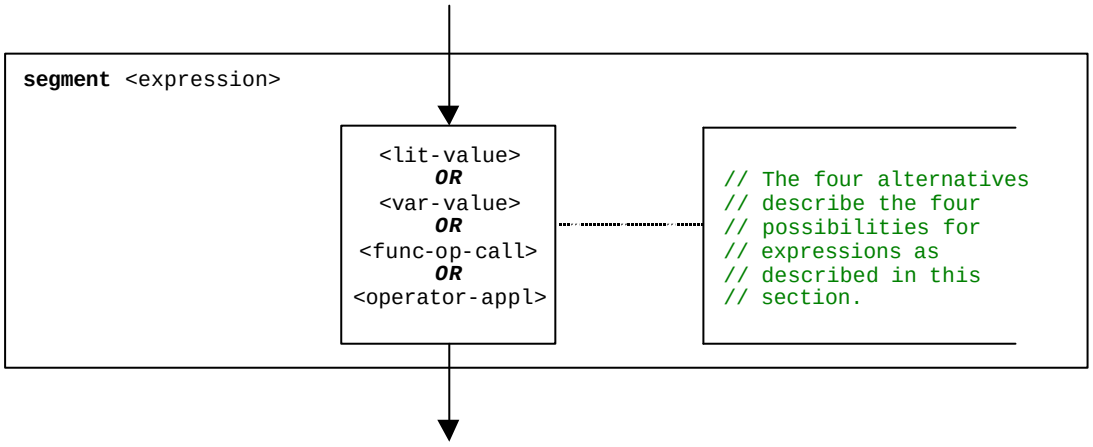


Figure B.61: Flow graph segment <expression>

B.3.7.18.1 Flow graph segment <lit-value>

The flow graph segment <lit-value> in figure B.62 pushes a literal value onto the value stack of an entity.

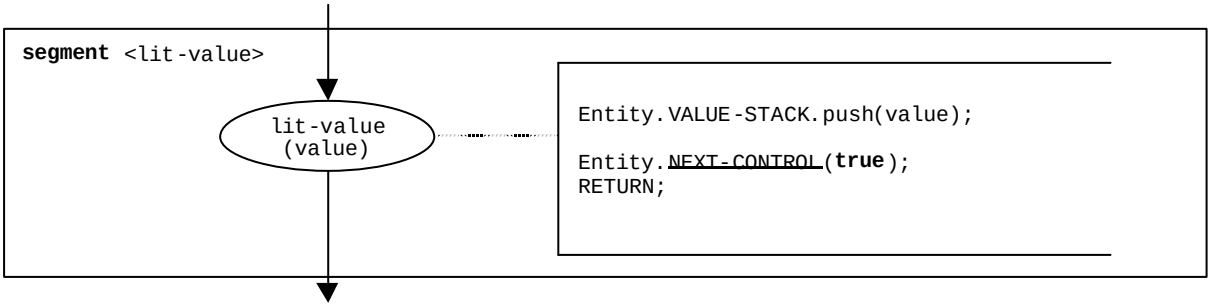


Figure B.62: Flow graph segment <lit-value>

B.3.7.18.2 Flow graph segment <var-value>

The flow graph segment <var-value> in figure B.63 pushes the value of a variable onto the value stack of an entity.

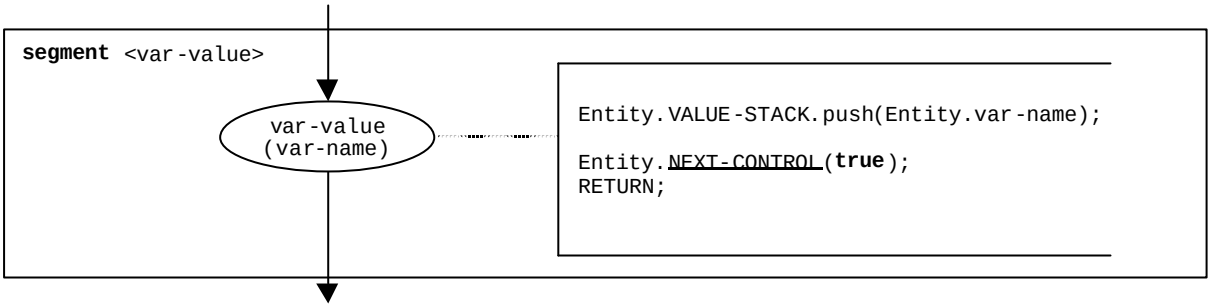


Figure B.63: Flow graph segment <var-value>

B.3.7.18.3 Flow graph segment <func-op-call>

The flow graph segment <func-op-call> in figure B.64 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.

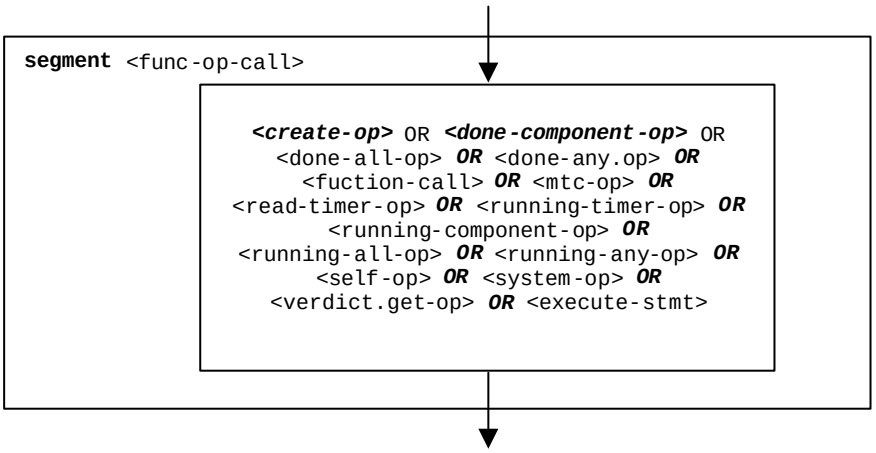


Figure B.64: Flow graph segment <func-op-call>

B.3.7.18.4 Flow graph segment <operator-appl>

The flow-graph representation in figure B.65 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.

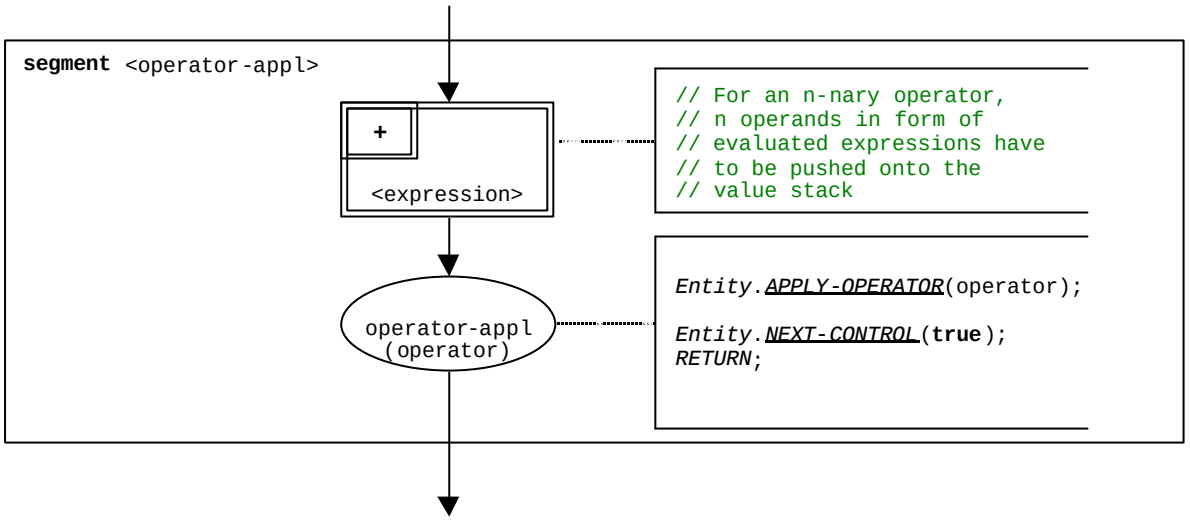


Figure B.65: Flow graph segment <operator-appl>

B.3.7.19 Flow graph segment <finalize-component-init>

The flow graph segment `<finalize-component-init>` is part of the flow graph representing the behaviour of a component type definition. Its execution is defined in figure B.66:

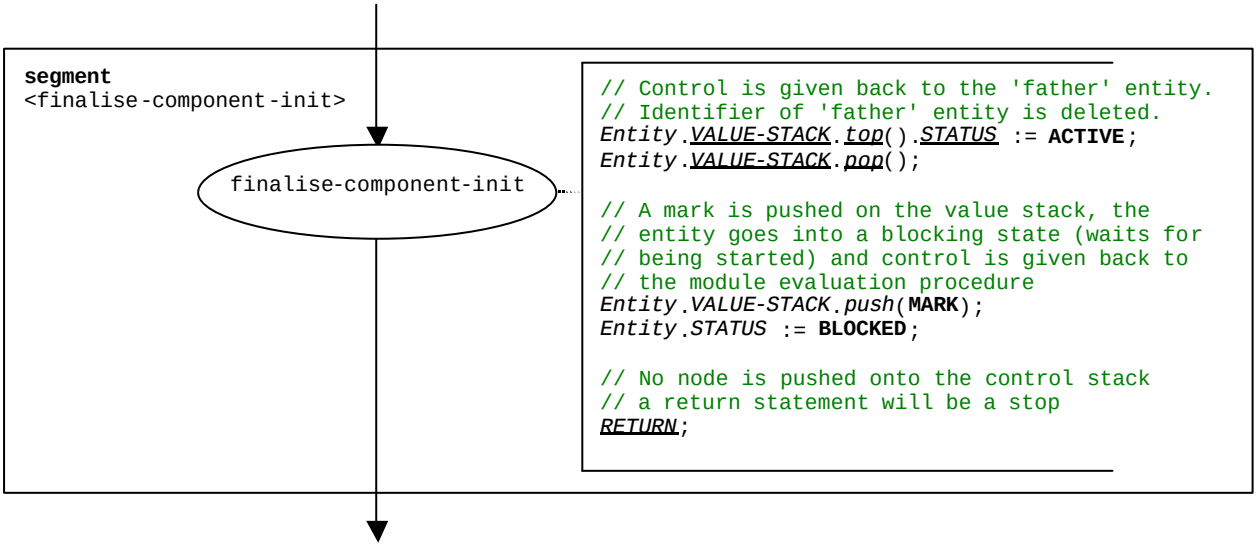


Figure B.66: Flow graph segment <finalize-component-init>

B.3.7.20 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 B.67:

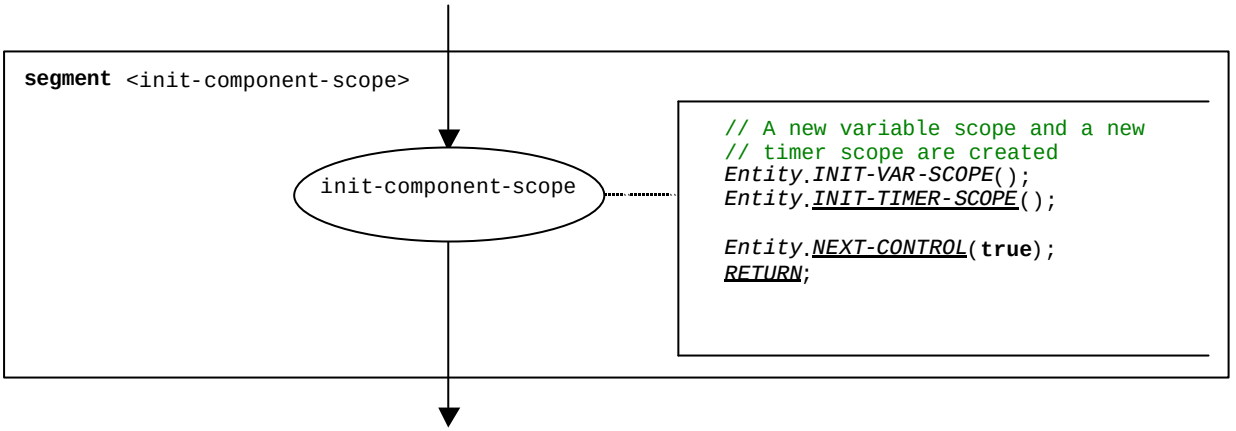


Figure B.67: Flow graph segment <init-component-scope>

B.3.7.21 For statement

The syntactical structure of the **for-statement** is:

```
for (<assignment>, <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. 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 B.68. The initial <assignment> describes the initialization of the index variable. The <assignment> in the **true** branch of the decision node describes the manipulation of the index variable.

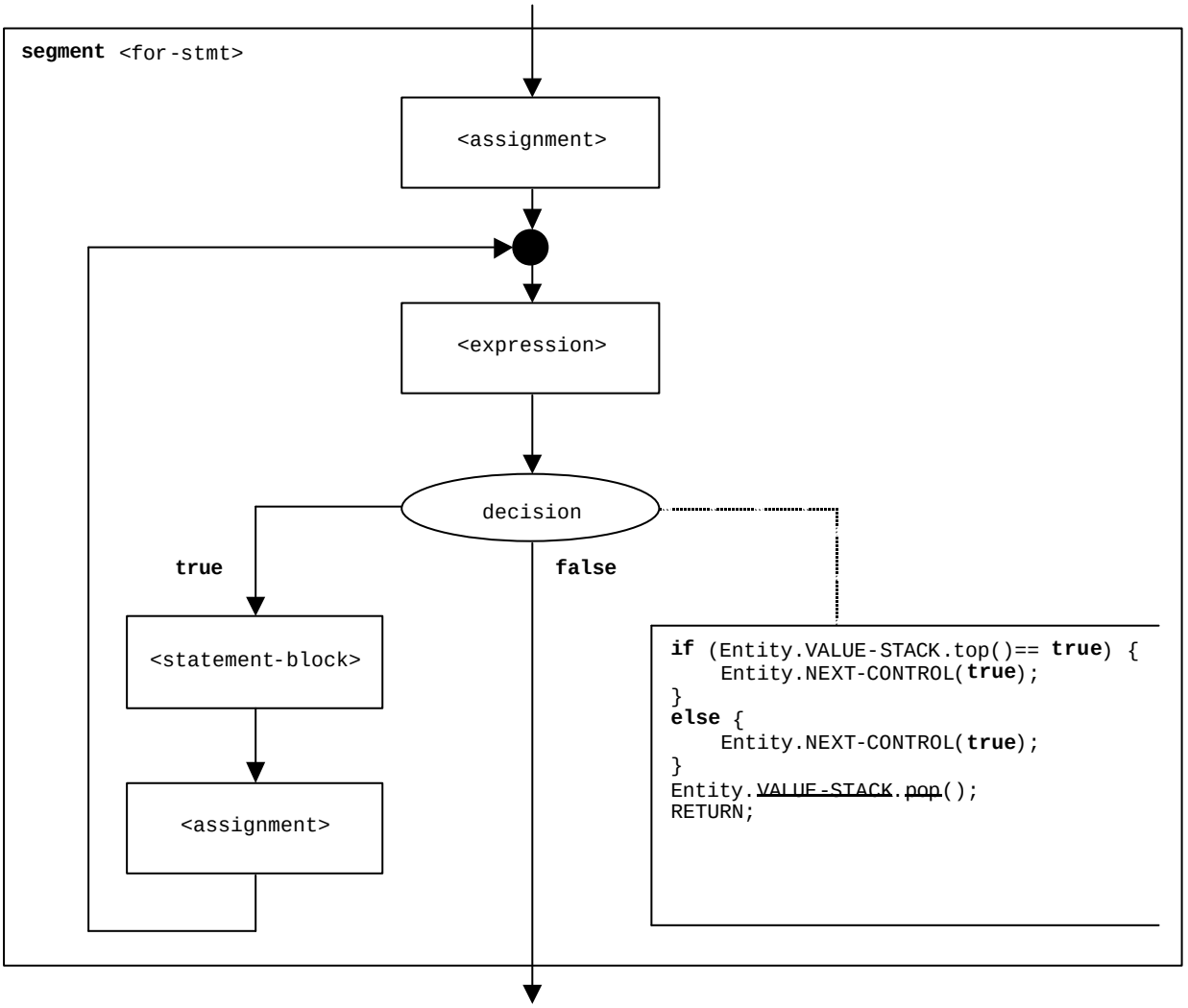


Figure B.68: Flow graph segment <for-stmt>

B.3.7.22 Function call

The syntactical structure of a function call is:

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

The <function-name> denotes to the name of a function and <act-par-descr$_1$>, … , <act-par-descr$_n$> describe the description of the actual parameter values of the function call. In case of a value parameter the description of an actual parameter may be provided in form of an expression that has to be evaluated before the call can be executed.

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 B.69 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, the control of the behaviour that calls the function is transferred.

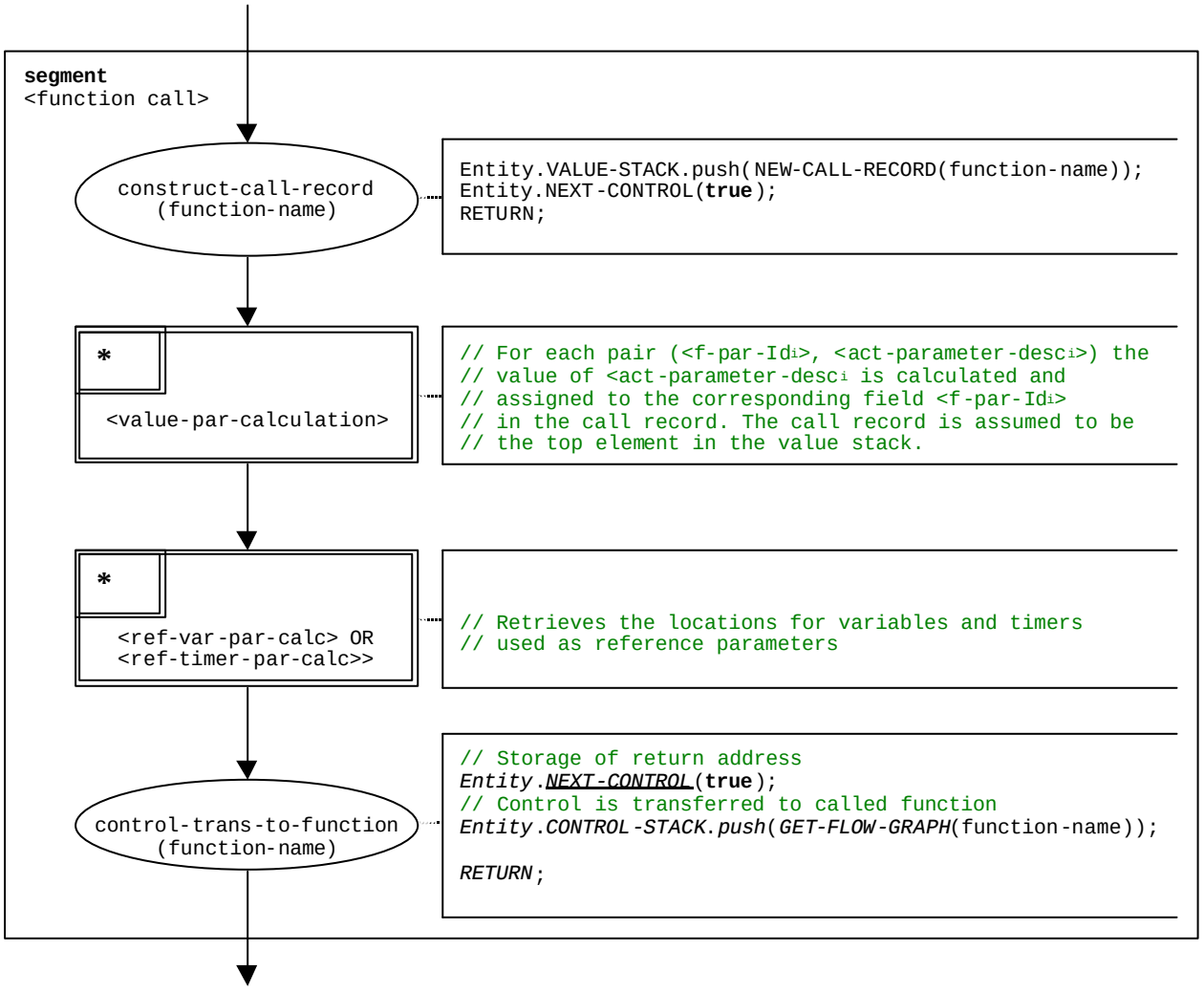


Figure B.69: Flow graph segment <function-call>

B.3.7.23 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 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$_i$> that 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 B.70.

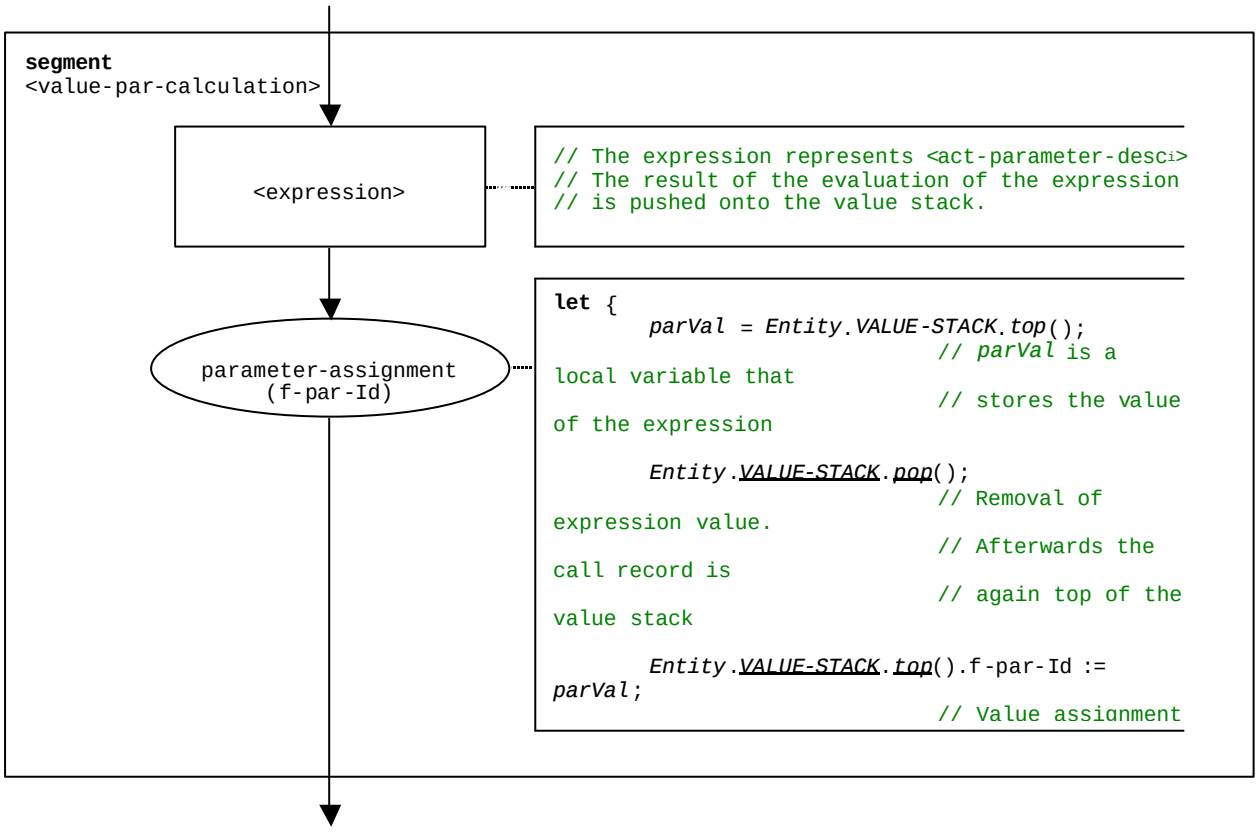


Figure B.70: Flow graph segment <value-par-calculation>

B.3.7.24 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 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 B.71.

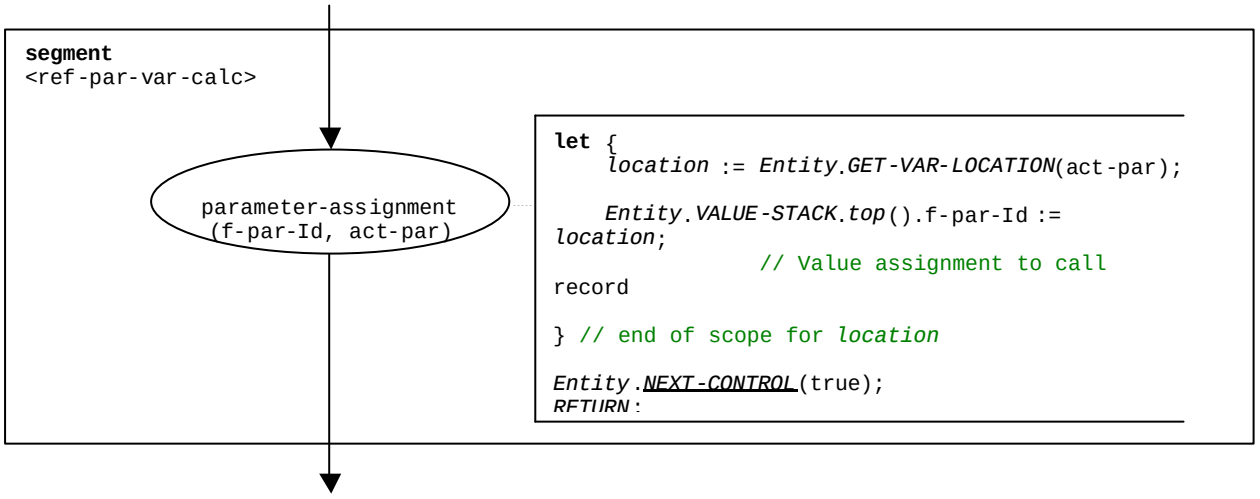


Figure B.71: Flow graph segment <ref-par-var-calc>

B.3.7.25 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 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 B.72.

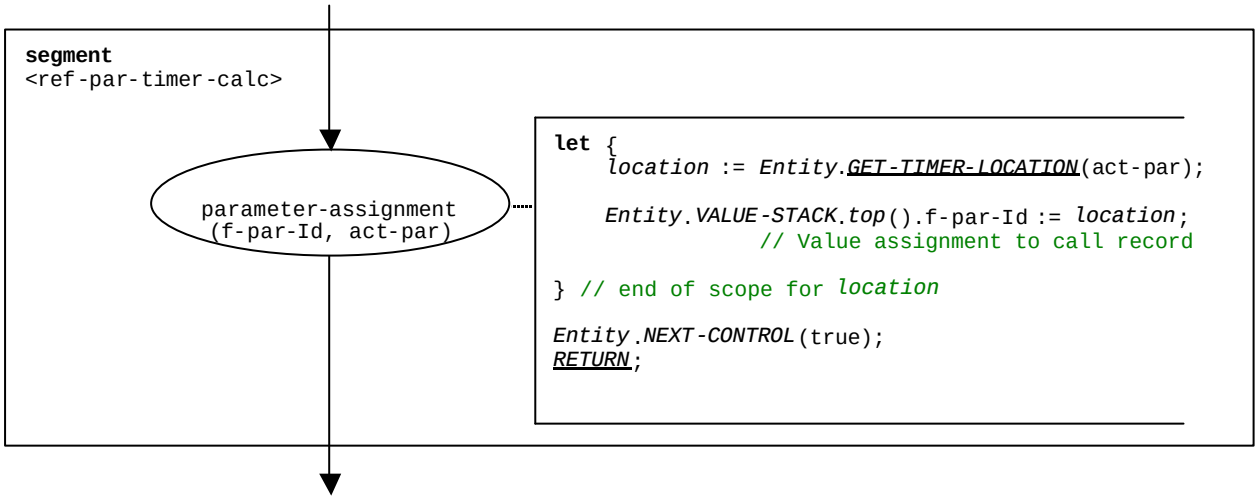


Figure B.72: Flow graph segment <ref-par-timer-calc>

B.3.7.26 Flow graph segment <parameter-handling>

The flow graph-segment <parameter-handling> is used in the beginning of function calls. It initializes a new scope and creates variables and timers for the handling of parameters. It is assumed that the call record of the called function is lying on top of the value stack.

The execution of flow graph-segment <parameter-handling> is shown in figure B.73.

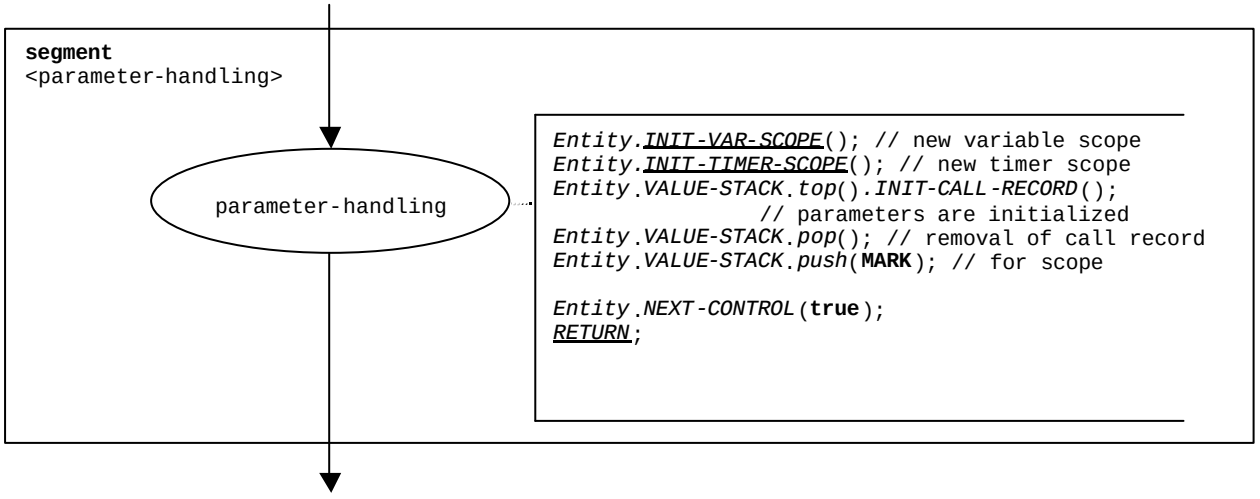


Figure B.73: Flow graph segment <parameter-handling>

B.3.7.27 Getcall operation

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

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

The optional <component_expression> in the **from** clause refers to the sender of the call that is handled by the **getcall** operation. 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 call matches to the matching specification <matchingSpec> and to the (optional) **from** clause.

The flow graph segment <getcall-op> in figure B.74 defines the execution of a **getcall** operation.

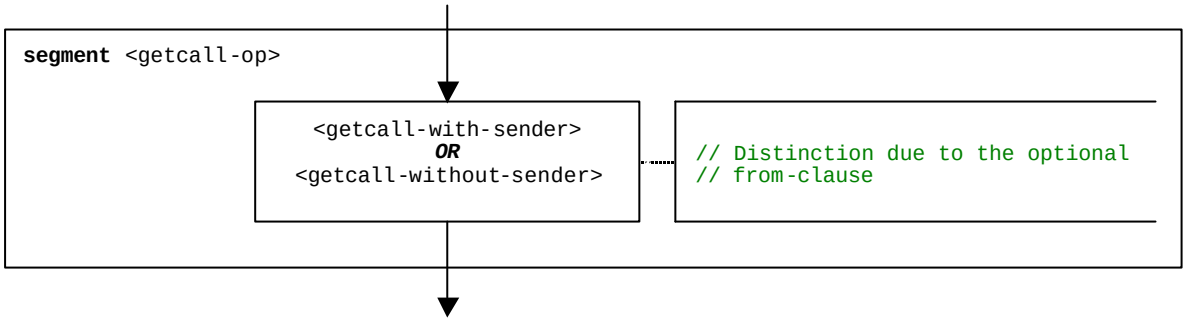


Figure B.74: Flow graph segment <getcall-op>

B.3.7.27.1 Flow graph segment <getcall-with-sender>

The flow graph segment `<getcall-with-sender>` in figure B.75 defines the execution of a **`getcall`** operation where the sender is specified in form of an expression.

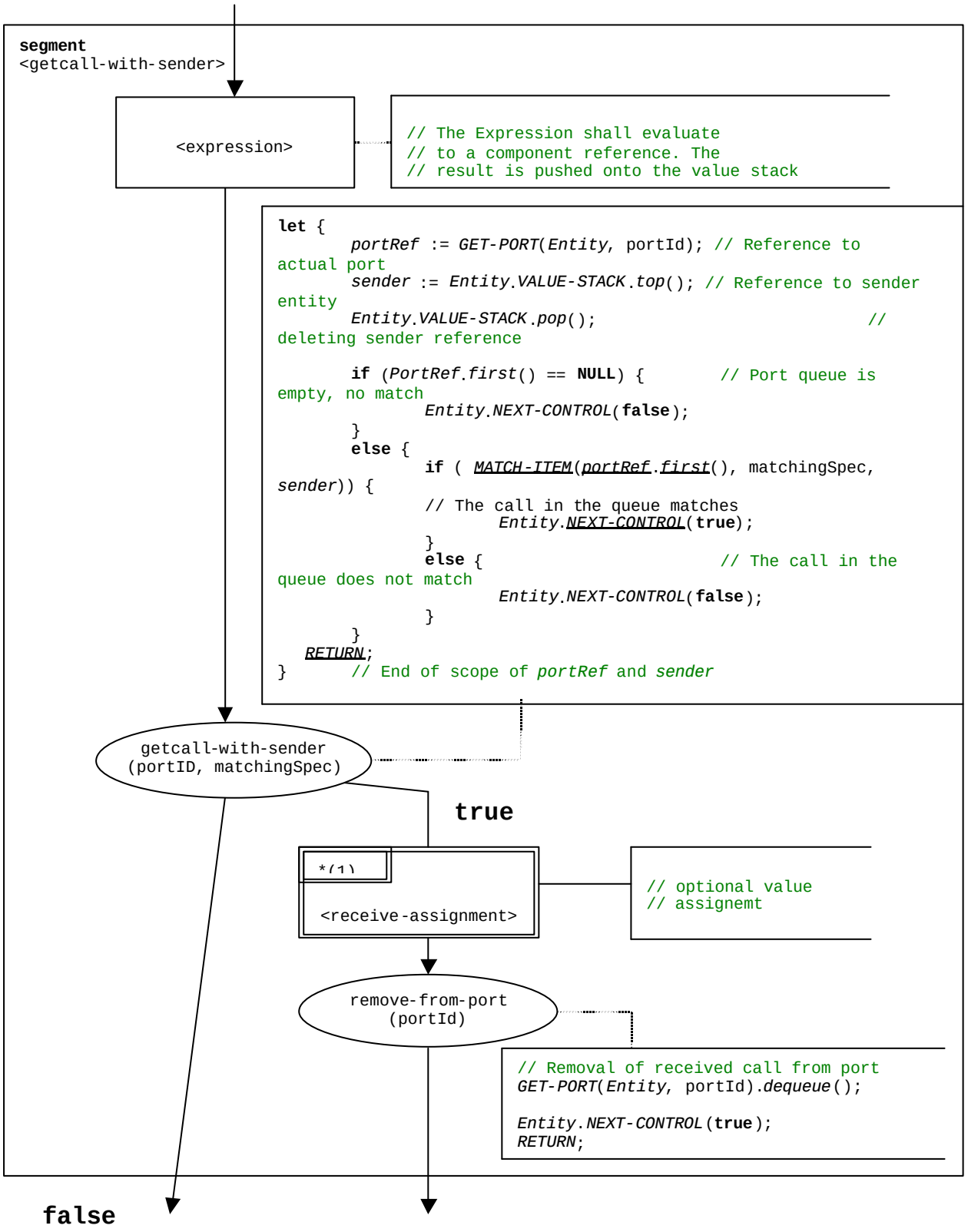


Figure B.75: Flow graph segment <getcall-with-sender>

B.3.7.27.2 Flow graph segment <getcall-without-sender>

The flow graph segment `<getcall-with-sender>` in figure B.76 defines the execution of a **`getcall`** operation without a **`from`** clause.

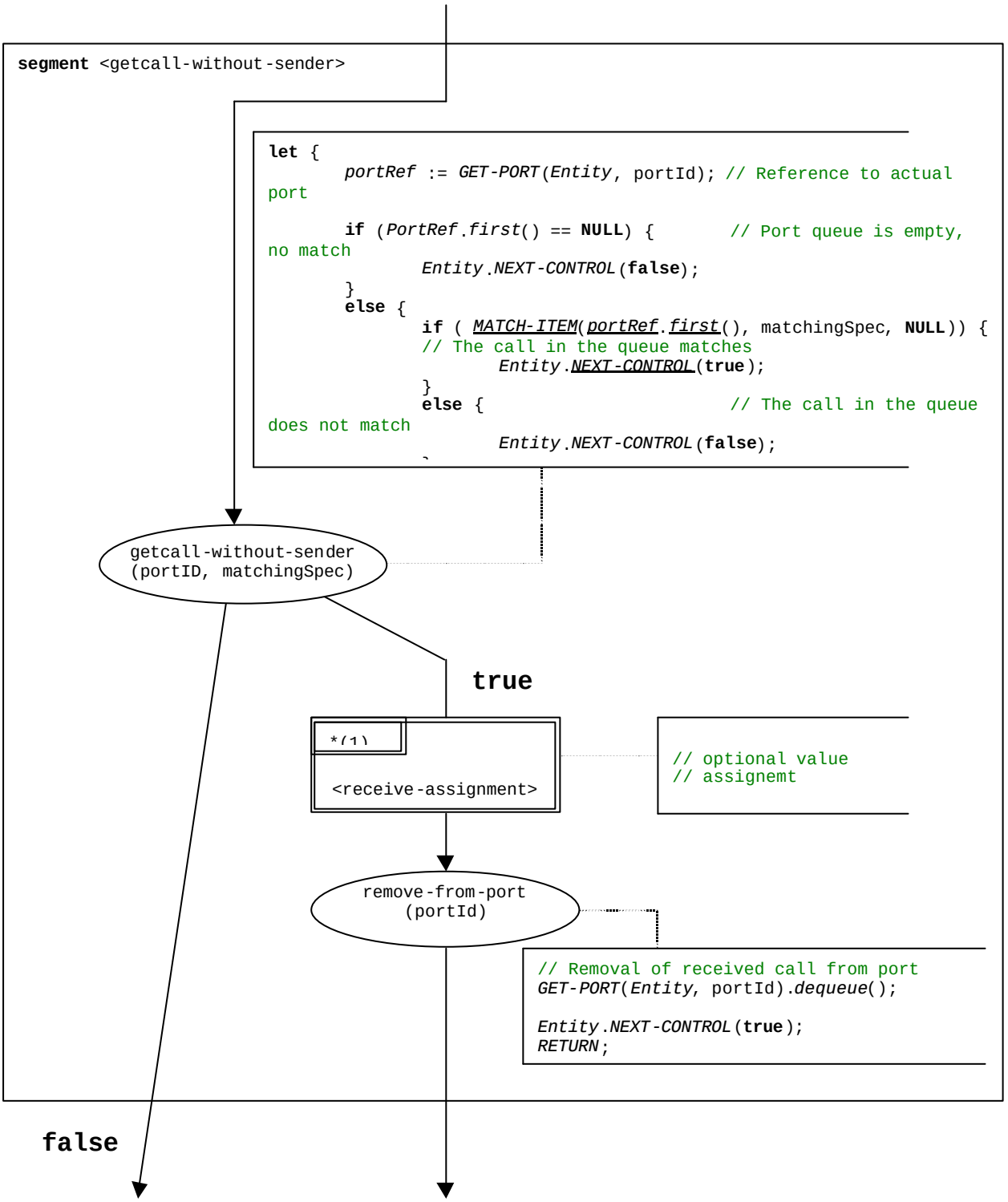


Figure B.76: Flow graph segment <getcall-without-sender>

B.3.7.28 Getreply operation

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

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

The optional <component_expression> in the **from** clause refers to the sender of the reply that is handled by the **getreply** operation. 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 the received information if the reply matches to the matching specification <matchingSpec> and to the (optional) **from** clause.

The flow graph segment <getreply-op> in figure B.77 defines the execution of a **getreply** operation.

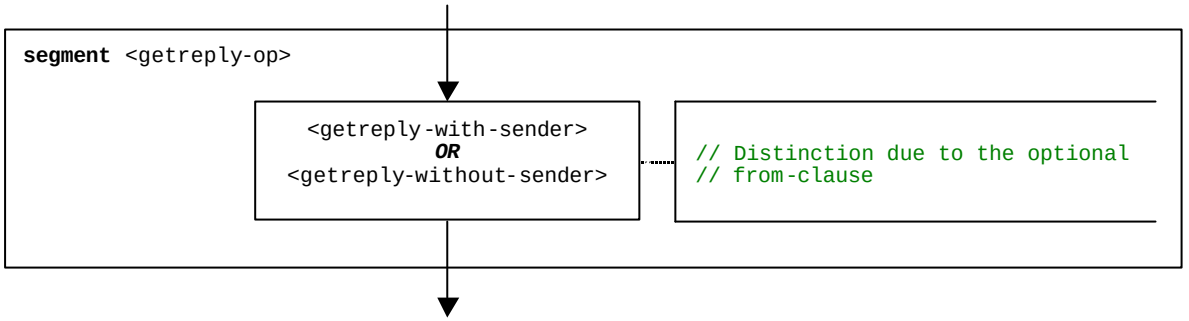


Figure B.77: Flow graph segment <getreply-op>

B.3.7.28.1 Flow graph segment <getreply-with-sender>

The flow graph segment `<getreply-with-sender>` in figure B.78 defines the execution of a **`getreply`** operation where the sender is specified in form of an expression.

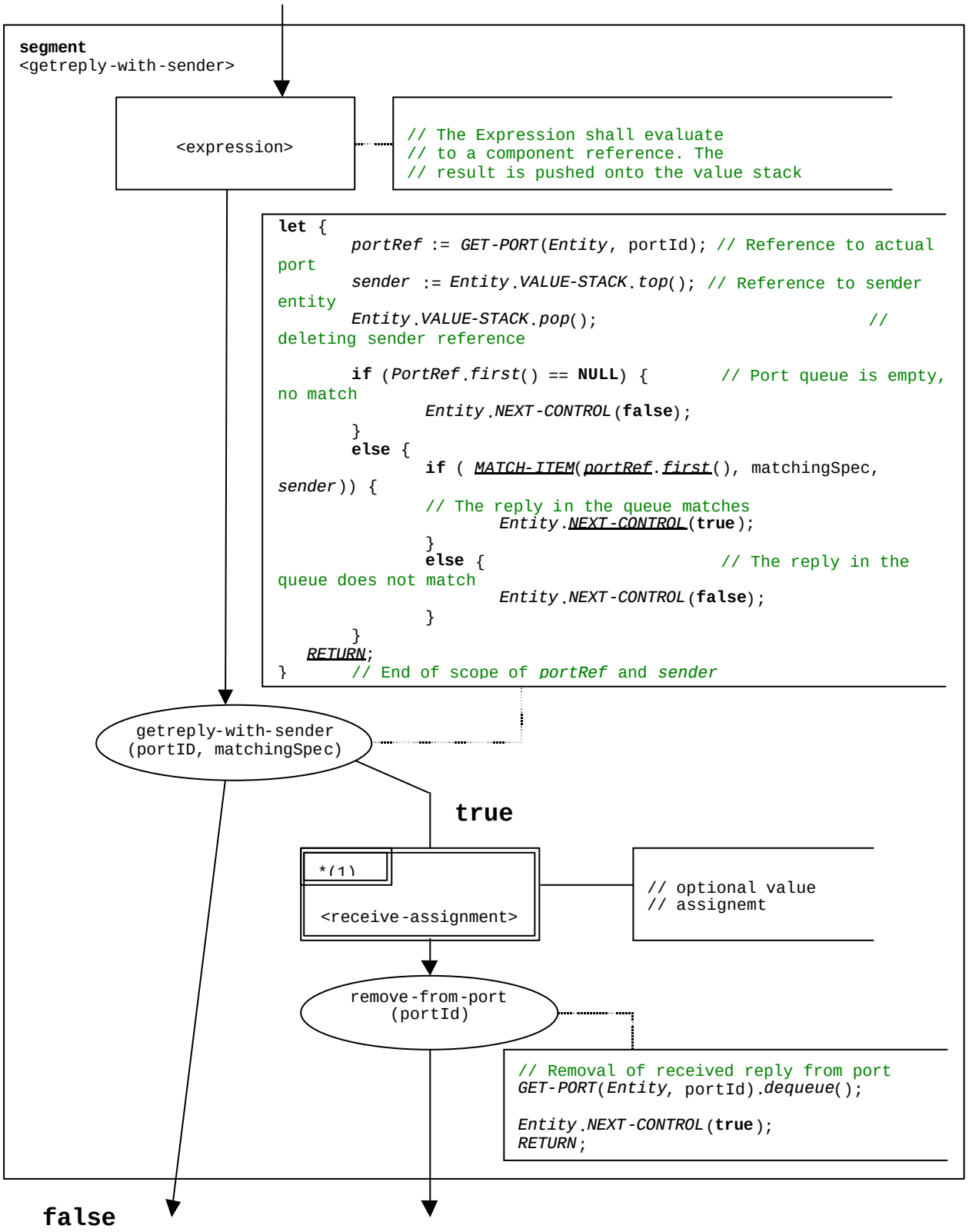


Figure B.78: Flow graph segment <getreply-with-sender>

B.3.7.28.2 Flow graph segment <getreply-without-sender>

The flow graph segment `<getreply-with-sender>` in figure B.79 defines the execution of a **`getreply`** operation without a **`from`** clause.

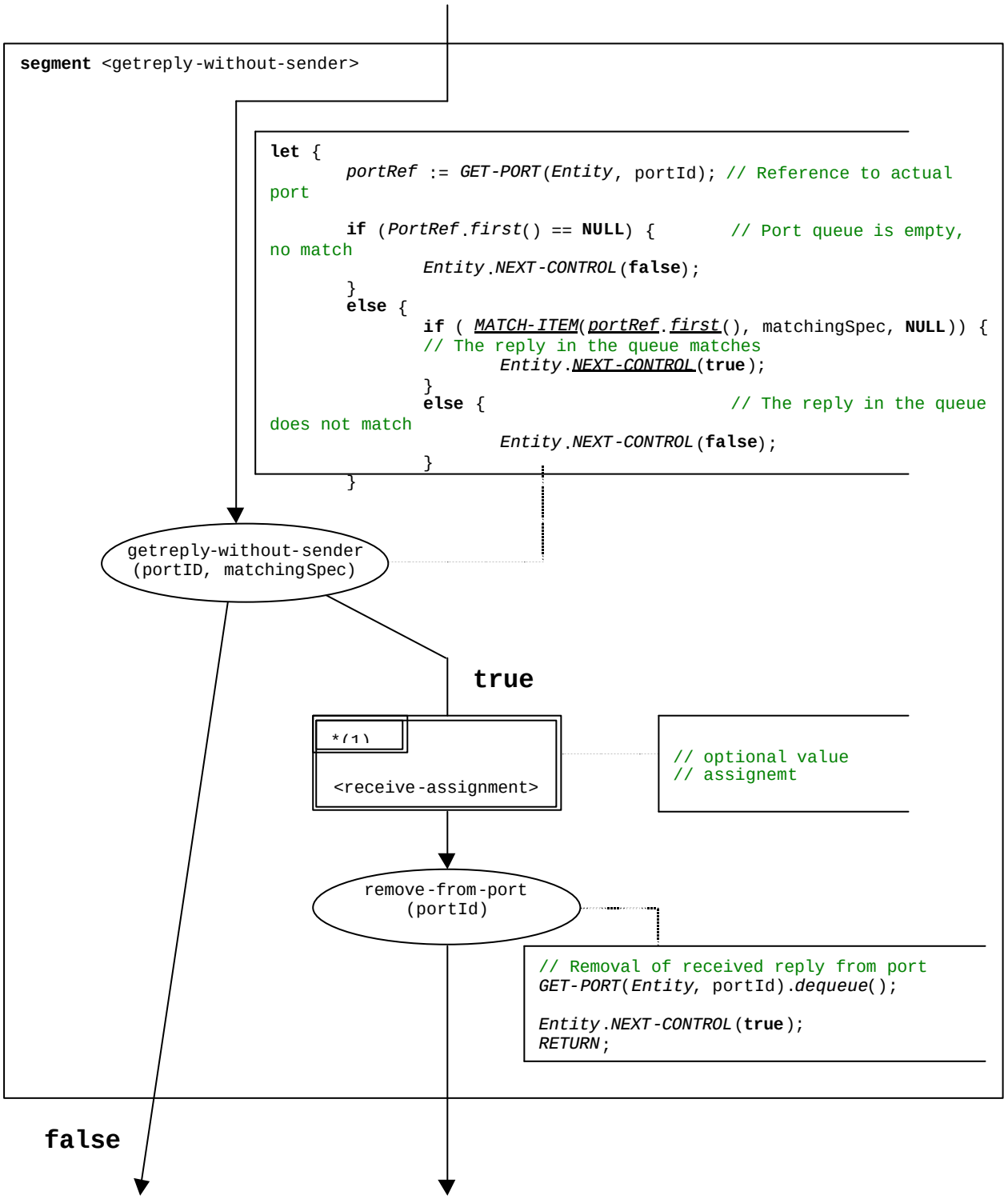


Figure B.79: Flow graph segment <getreply-without-sender>

B.3.7.29 Goto statement

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

```
goto <labelId>
```

The flow graph segment <goto-stmt> in figure B.80 defines the execution of the **goto** statement.

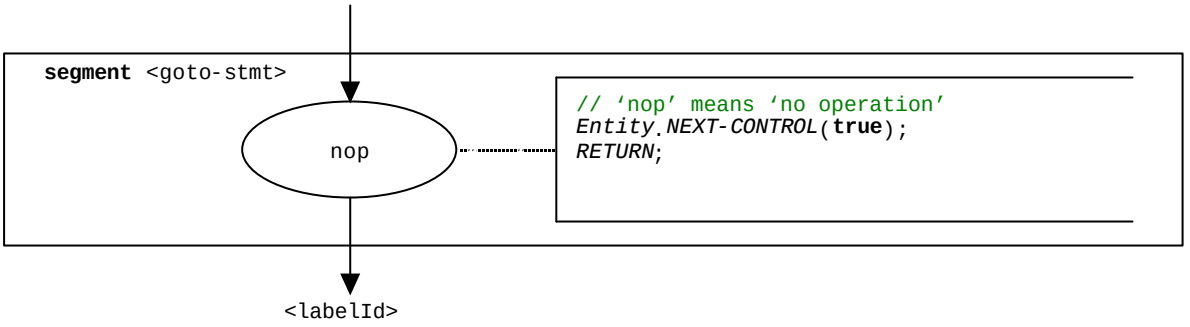


Figure B.80: Flow graph segment <goto-stmt>

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 B.3.7.31).

B.3.7.30 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 B.81 defines the execution of the **if-else** statement.

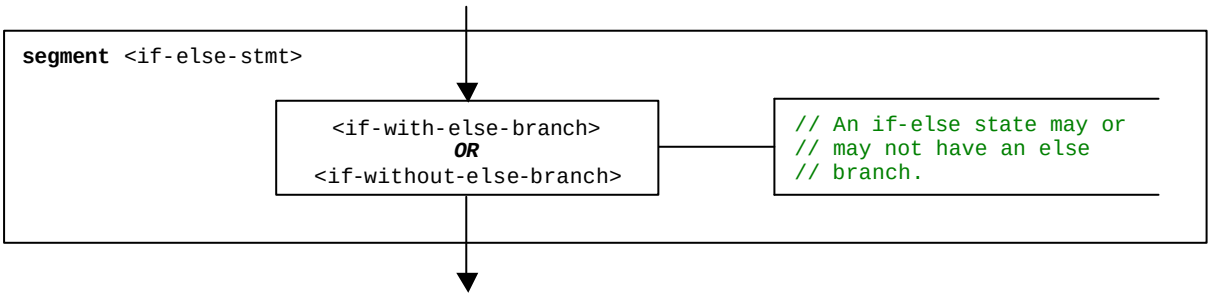


Figure B.81: Flow graph segment <if-else-stmt>

B.3.7.30.1 Flow graph segment <if-with-else-branch>

Figure B.82 describes the execution of an **`if-else`** statement that includes an else branch. The `<statement-block>` in the **`true`** branch of the decision node in figure B.82, corresponds to $\texttt{<statement-block}_1\texttt{>}$ in the syntactical structure above. The other `<statement-block>` corresponds to $\texttt{<statement-block}_2\texttt{>}$ above.

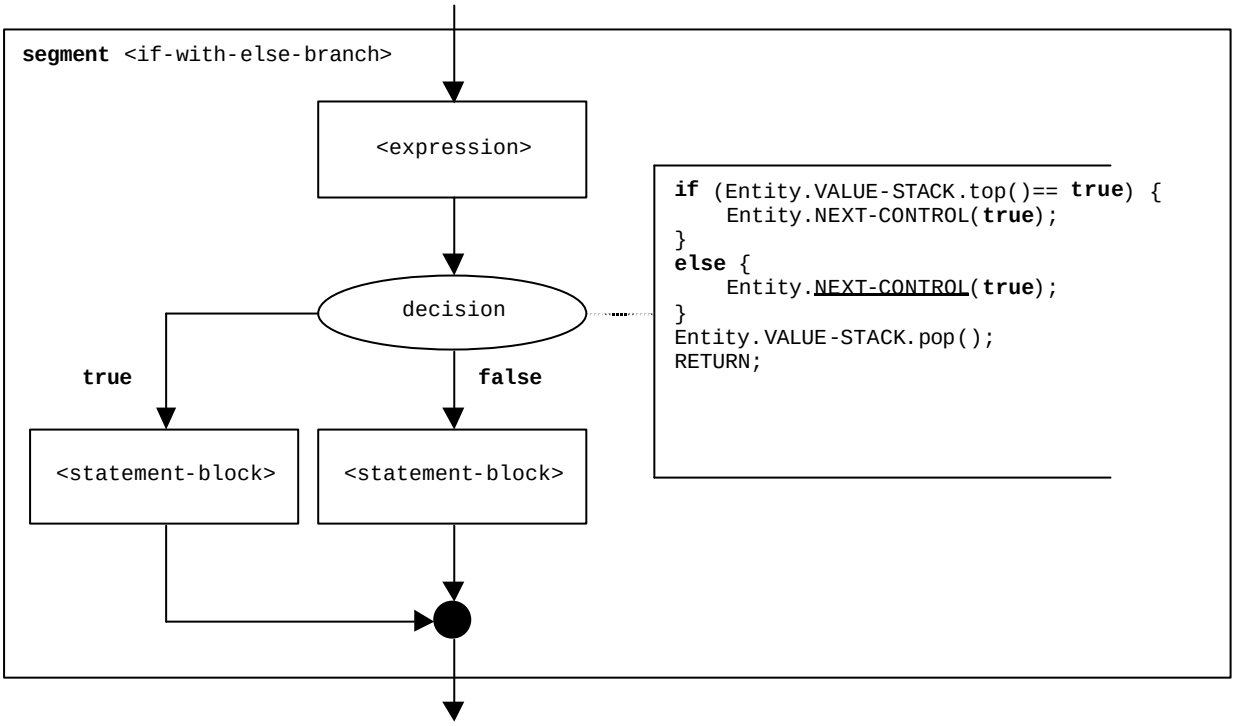


Figure B.82: Flow graph segment <if-with-else-branch>

B.3.7.30.2 Flow graph segment <if-without-else-branch>

Figure B.83 describes the execution of an **`if-else`** statement that includes no else branch. The `<statement-block>` in the **`true`** branch of the decision node in figure B.82, corresponds to `<statement-block`$_1$`>` in the syntactical structure above.

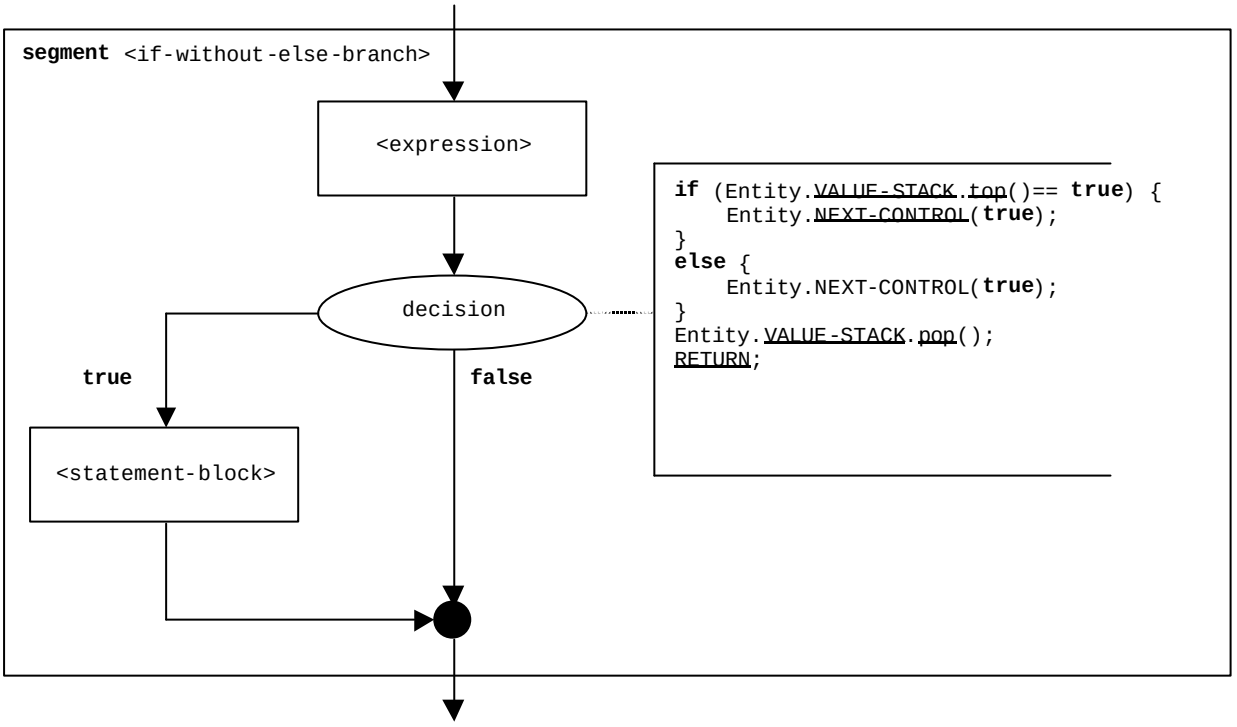


Figure B.83: Flow graph segment <if-without-else-branch>

B.3.7.31 Label statement

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

`label` `<labelId>`

The flow graph segment <label-stmt> in figure B.84 defines the execution of the **`label`** statement.

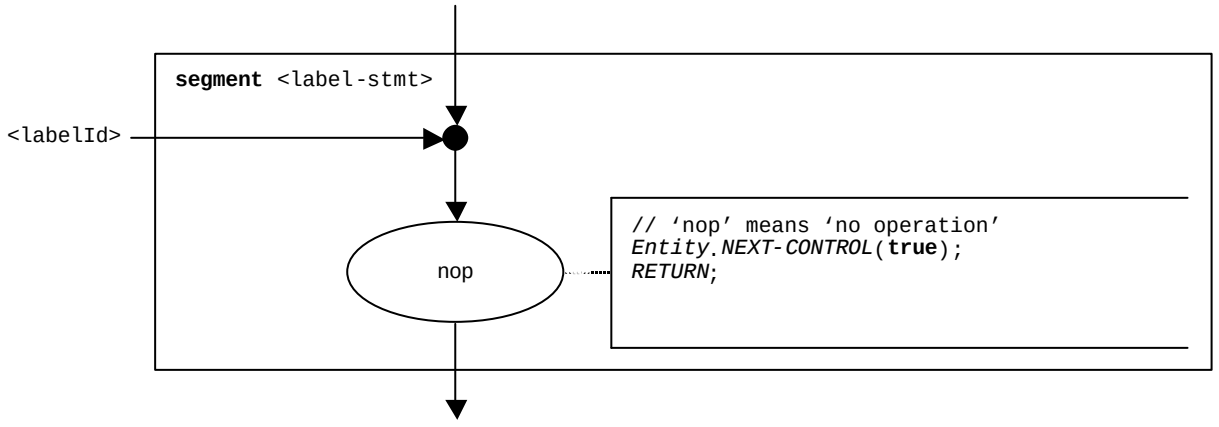


Figure B.84: Flow graph segment <label-stmt>

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 B.3.7.29).

B.3.7.32 Log statement

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

```
log (<informal-description>)
```

The flow graph segment <log-stmt> in figure B.85defines the execution of the **log** statement.

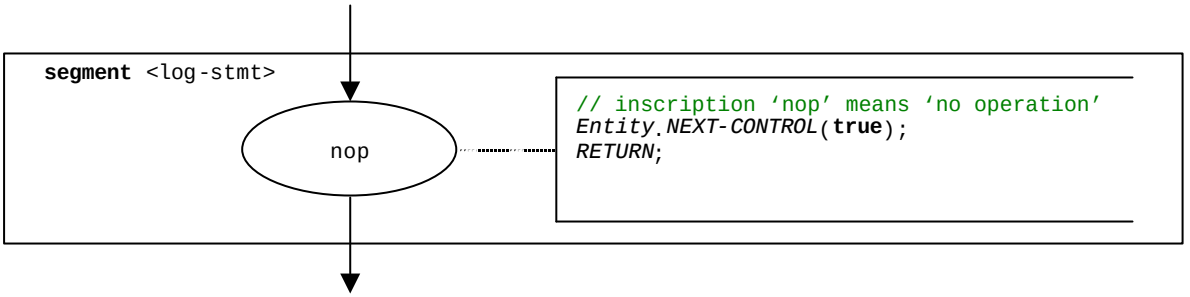


Figure B.85: Flow graph segment <log-stmt>

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.

B.3.7.33 Map operation

The syntactical structure of a 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. For simplicity it is considered to be an expression that evaluates to a component reference. Thus, 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 B.86.

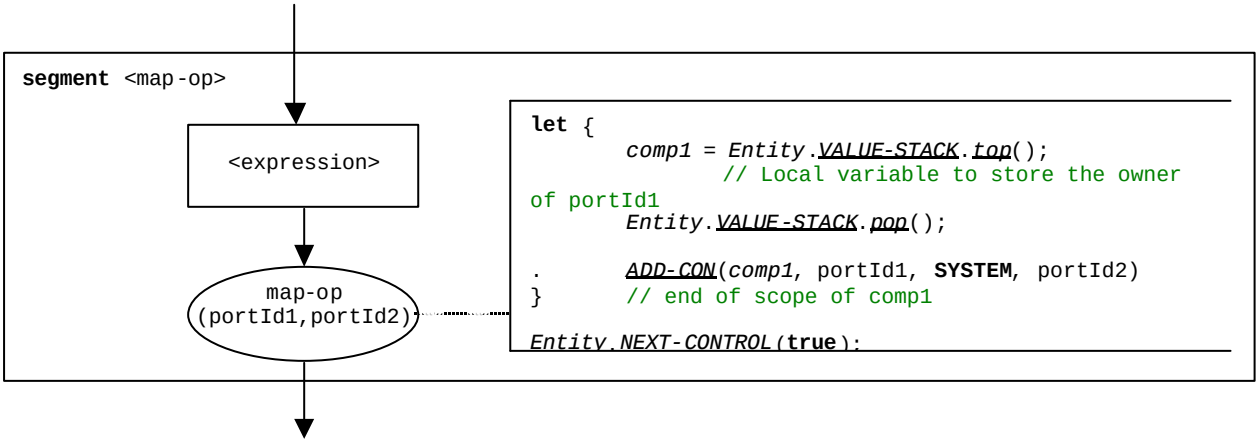


Figure B.86: Flow graph segment <map-op>

B.3.7.34 MTC operation

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

```
mtc
```

The flow graph segment <mtc-op> in figure B.87 defines the execution of the **mtc** operation.

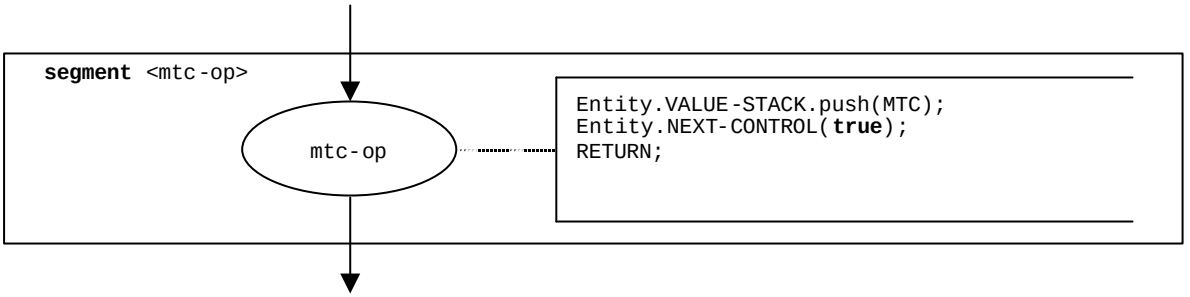


Figure B.87: Flow graph segment <mtc-op>

B.3.7.35 Raise operation

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

```
<portId>.raise (<exceptSpec>) [to <component_expression>]
```

The optional <component_expression> in the to clause refers to the receiver entity. It may be provided in form of a variable value or the return value of a function.

The flow graph segment <raise-op> in figure B.88 defines the execution of a **raise** operation.

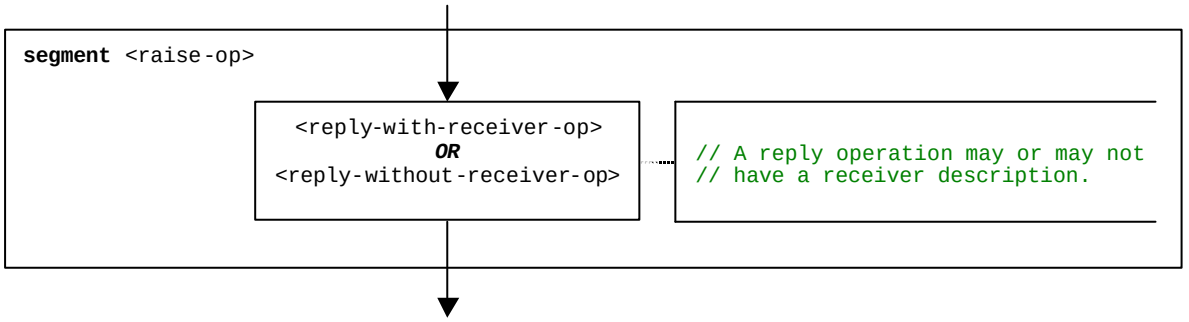


Figure B.88: Flow graph segment <raise-op>

B.3.7.35.1 Flow graph segment <raise-with-receiver-op>

The flow graph segment `<raise-with-receiver-op>` in figure B.89 defines the execution of a **`raise`** operation where the receiver is specified in form of an expression.

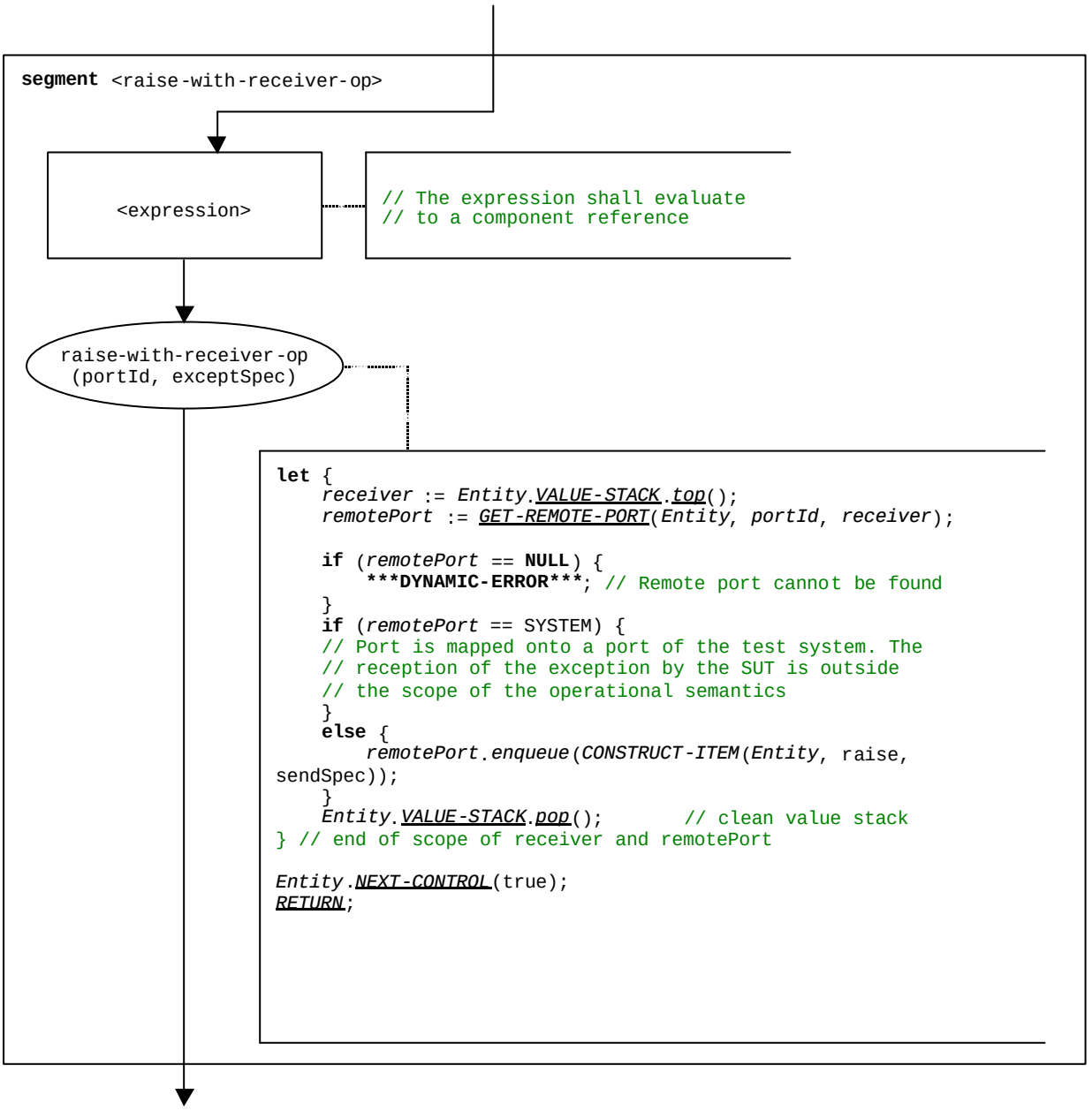


Figure B.89: Flow graph segment <raise-with-receiver-op>

B.3.7.35.2 Flow graph segment <raise-without-receiver-op>

The flow graph segment `<raise-without-receiver-op>` in figure B.90 defines the execution of a raise operation without **`to`**-clause.

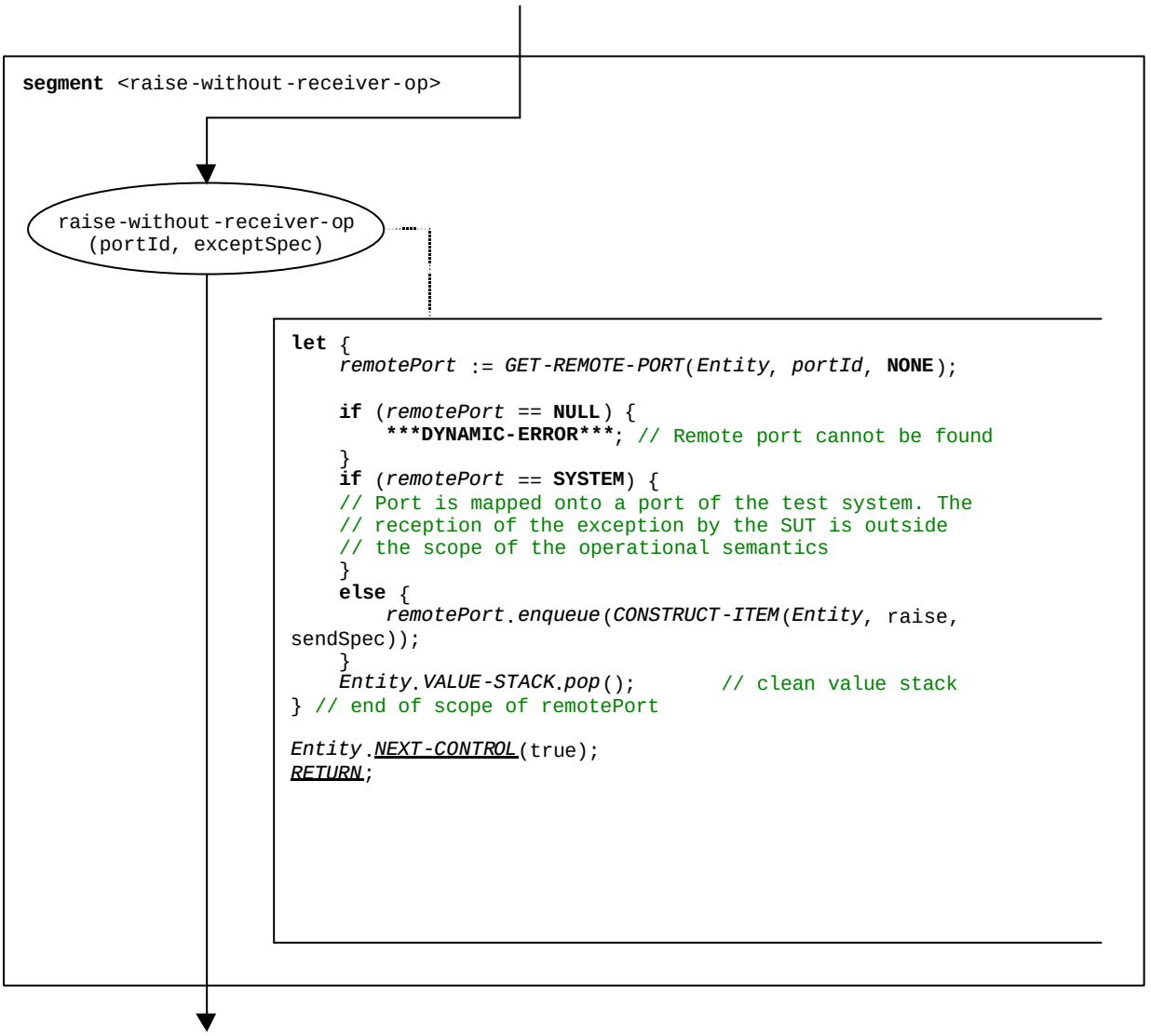


Figure B.90: Flow graph segment <raise-without-receiver-op>

B.3.7.36 Read timer operation

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

```
<timerId>.read
```

The flow graph segment <read-timer-op> in figure B.91 defines the execution of the **read** timer operation.

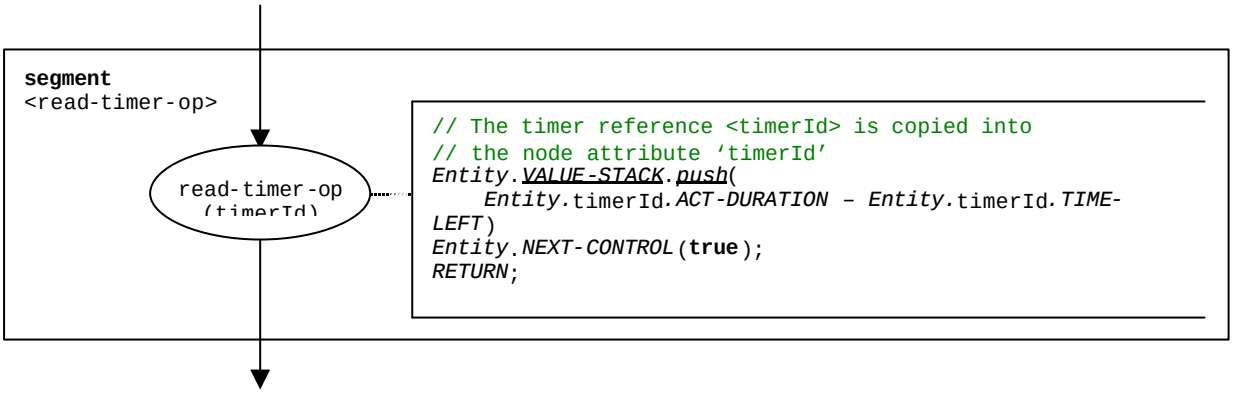


Figure B.91: Flow graph segment <read-timer-op>

B.3.7.37 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 B.92 defines the execution of a **receive** operation.

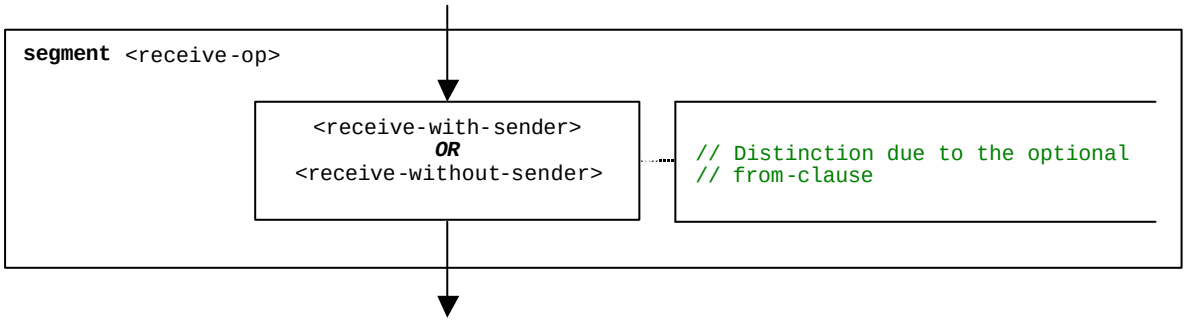


Figure B.92: Flow graph segment <receive-op>

B.3.7.37.1 Flow graph segment <receive-with-sender>

The flow graph segment `<receive-with-sender>` in figure B.93 defines the execution of a **`receive`** operation where the sender is specified in form of an expression.

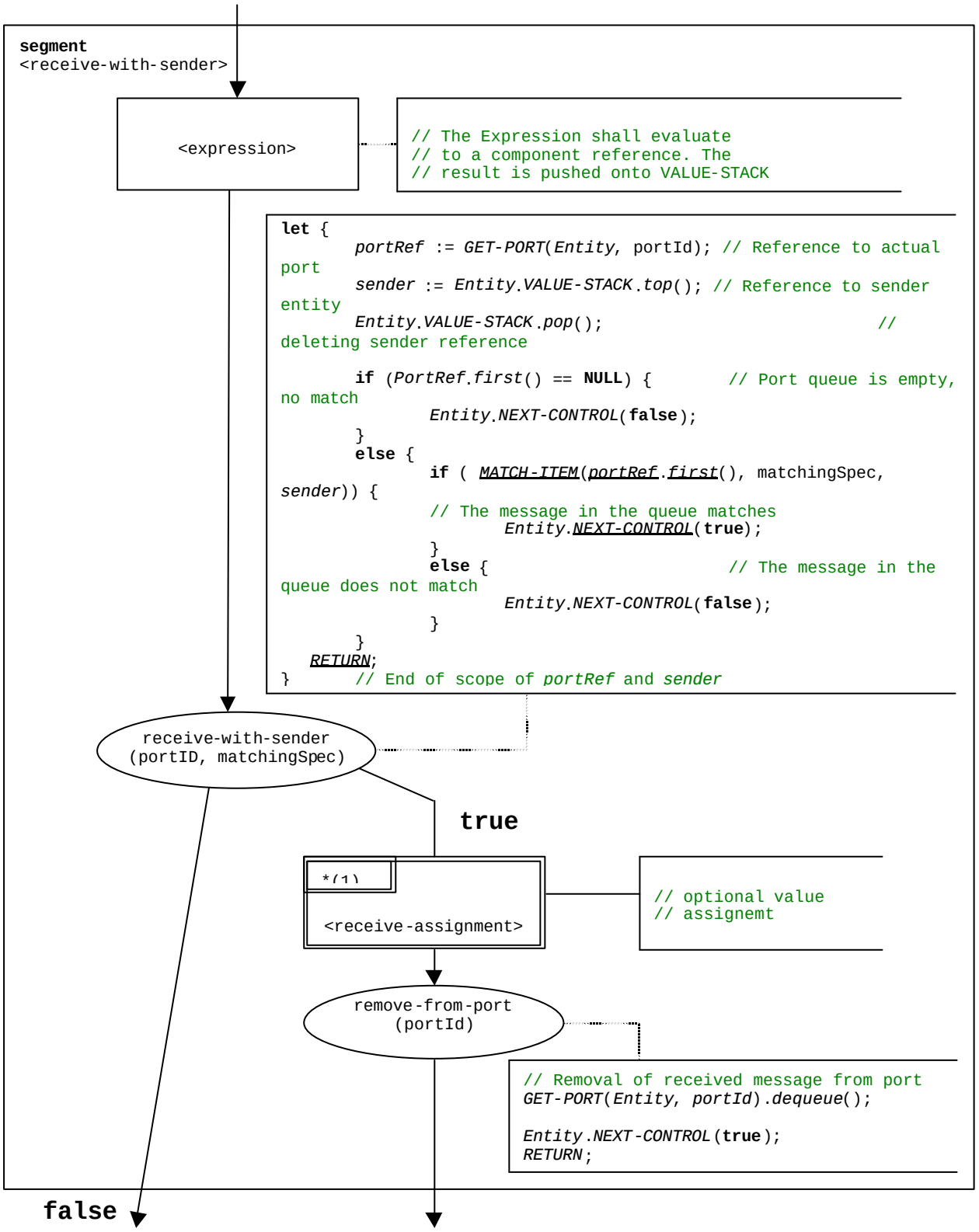


Figure B.93: Flow graph segment <receive-with-sender>

B.3.7.37.2 Flow graph segment <receive-without-sender>

The flow graph segment `<receive-with-sender>` in figure B.94 defines the execution of a **`receive`** operation without a **`from`** clause.

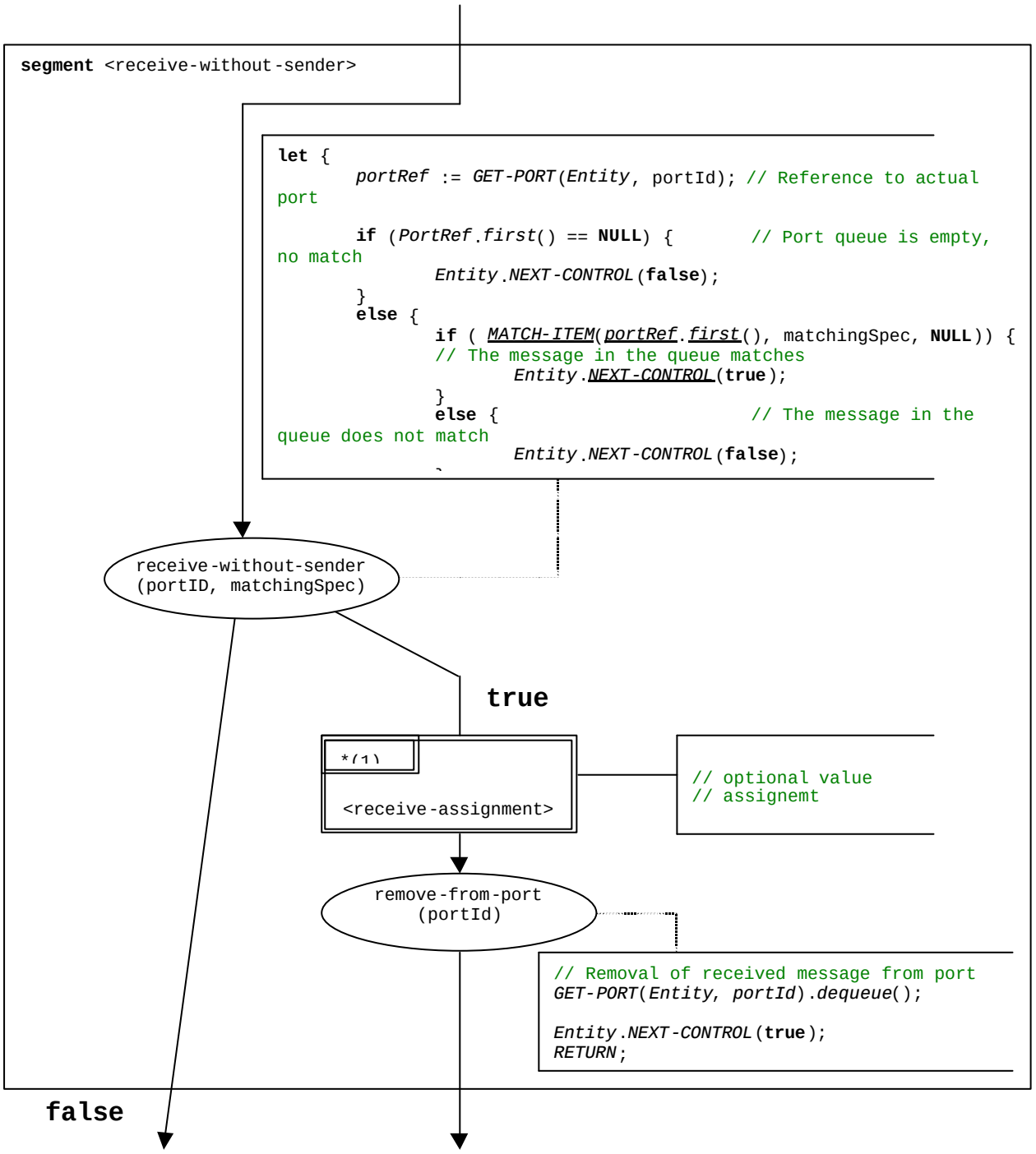


Figure B.94: Flow graph segment <receive-without-sender>

B.3.7.37.3 Flow graph segment <receive-assignment>

The flow graph segment `<receive-assignment>` in figure B.95 defines the retrieval of information from received messages and their assignment to variables.

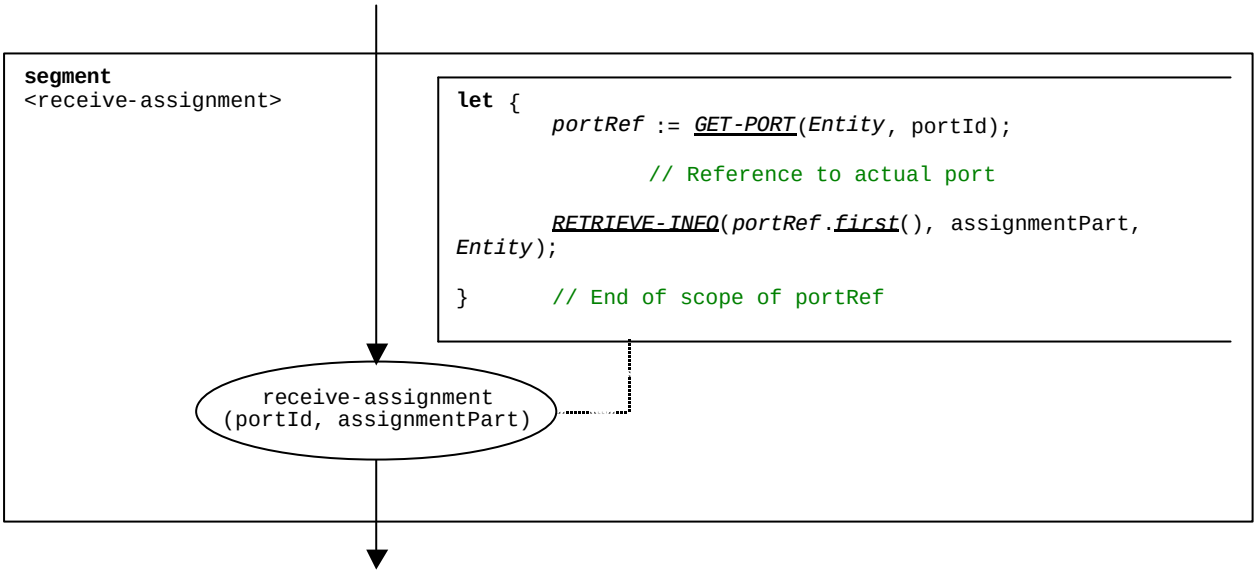


Figure B.95: Flow graph segment <receive-assignment>

B.3.7.38 Reply operation

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

```
<portId>.reply (<replySpec>) [to <component_expression>]
```

The optional `<component_expression>` in the to clause refers to the receiver entity. It may be provided in form of a variable value or the return value of a function.

The flow graph segment `<reply-op>` in figure B.96 defines the execution of a **`reply`** operation.

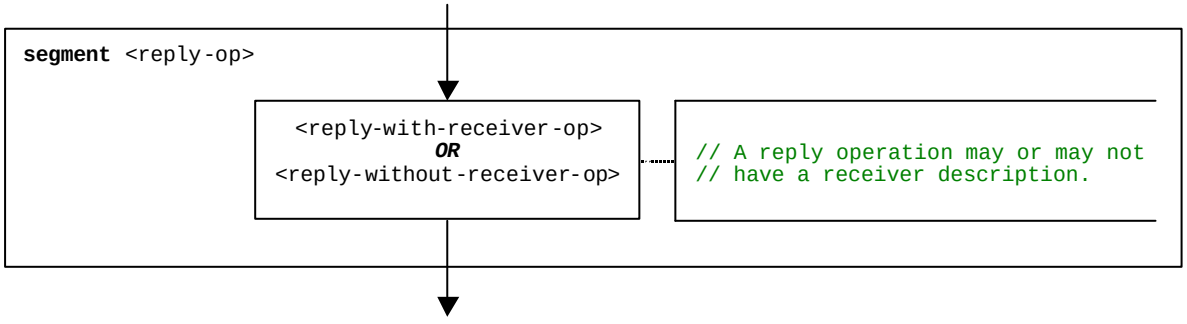


Figure B.96: Flow graph segment <reply-op>

B.3.7.38.1 Flow graph segment <reply-with-receiver-op>

The flow graph segment `<reply-with-receiver-op>` in figure B.97 defines the execution of a **`reply`** operation where the receiver is specified in form of an expression.

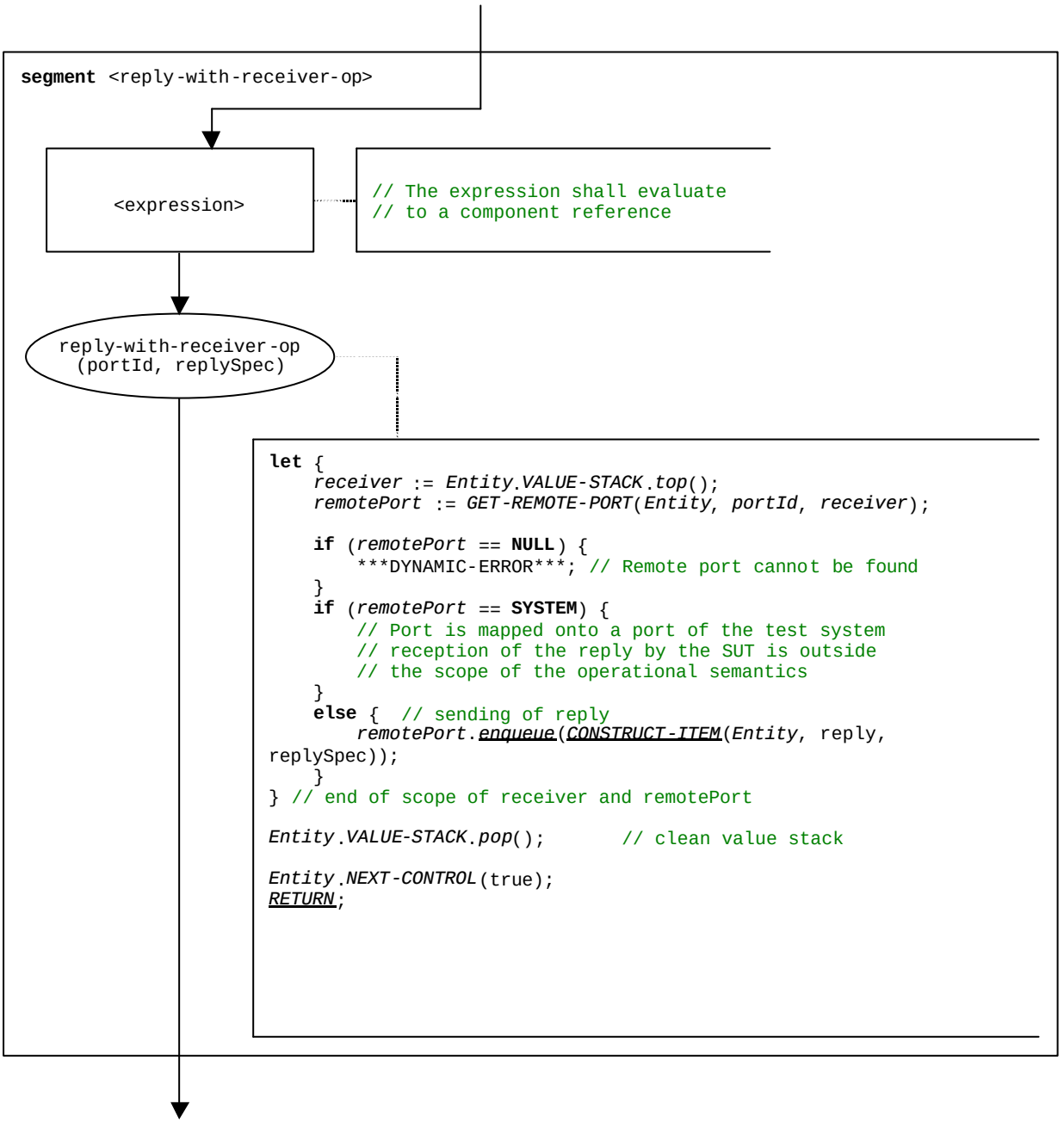


Figure B.97: Flow graph segment <reply-with-receiver-op>

B.3.7.38.2 Flow graph segment <reply-without-receiver-op>

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

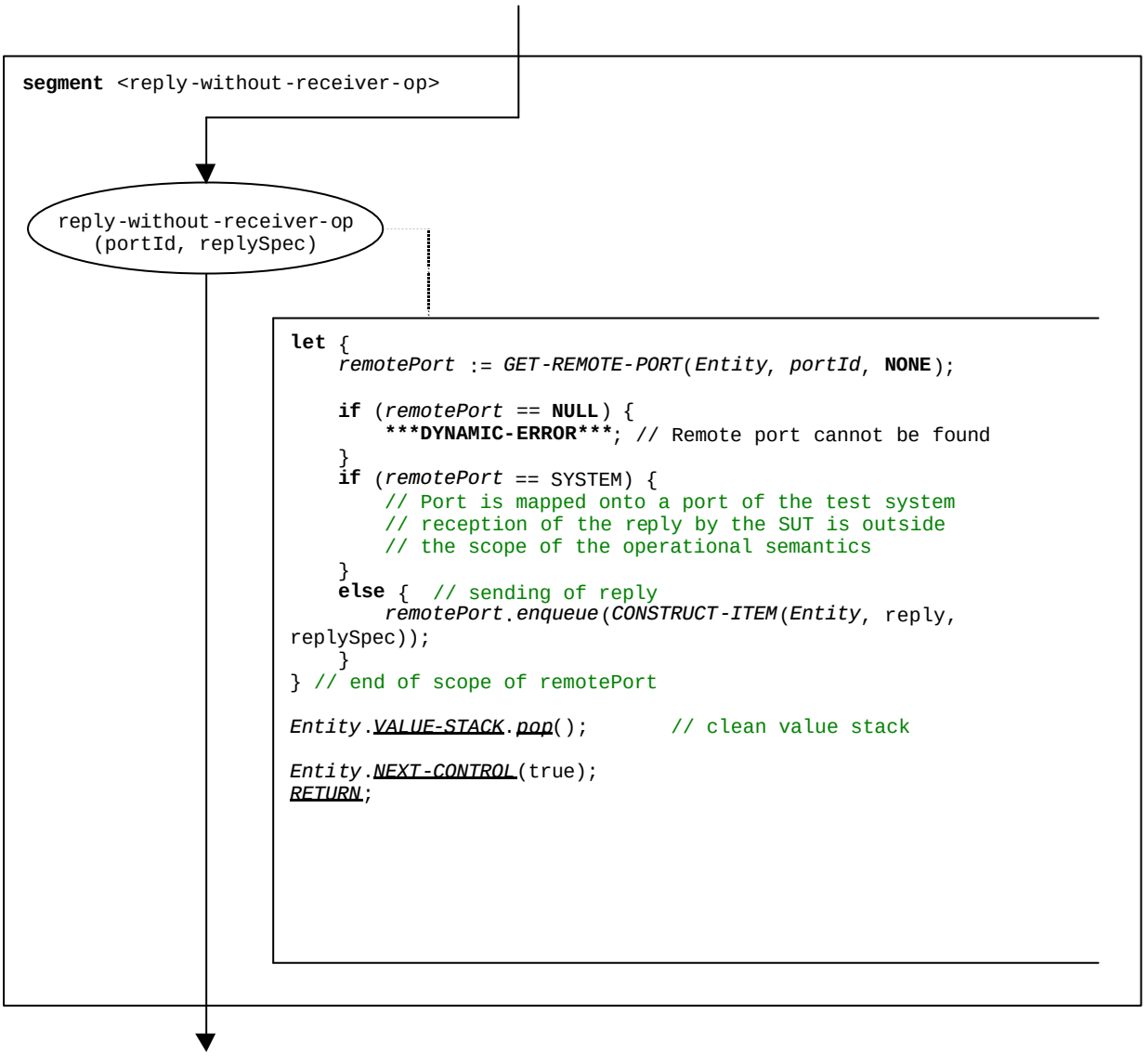


Figure B.98: Flow graph segment <reply-without-receiver-op>

B.3.7.39 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`** operation, if it is the last statement in a behaviour description.

NOTE: Due to the replacement of shorthand notations Test cases and module control will always end with a **`stop`** operation. Only other test components may terminate with a **`return`** statement.

The flow graph segment <return-stmt> in figure B.99 defines the execution of a **`return`** statement.

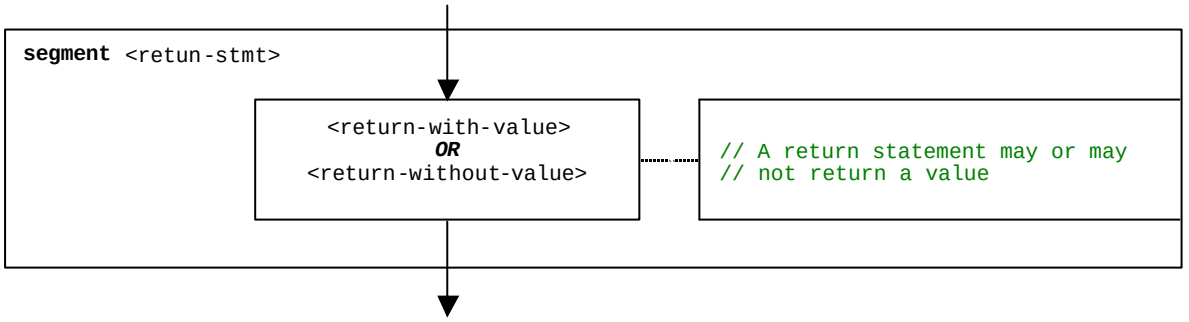


Figure B.99: Flow graph segment <return-stmt>

B.3.7.39.1 Flow graph segment <return-with-value>

The flow graph segment `<return-with-value>` in figure B.100 defines the execution of a **`return`** that returns a value specified in form of an expression.

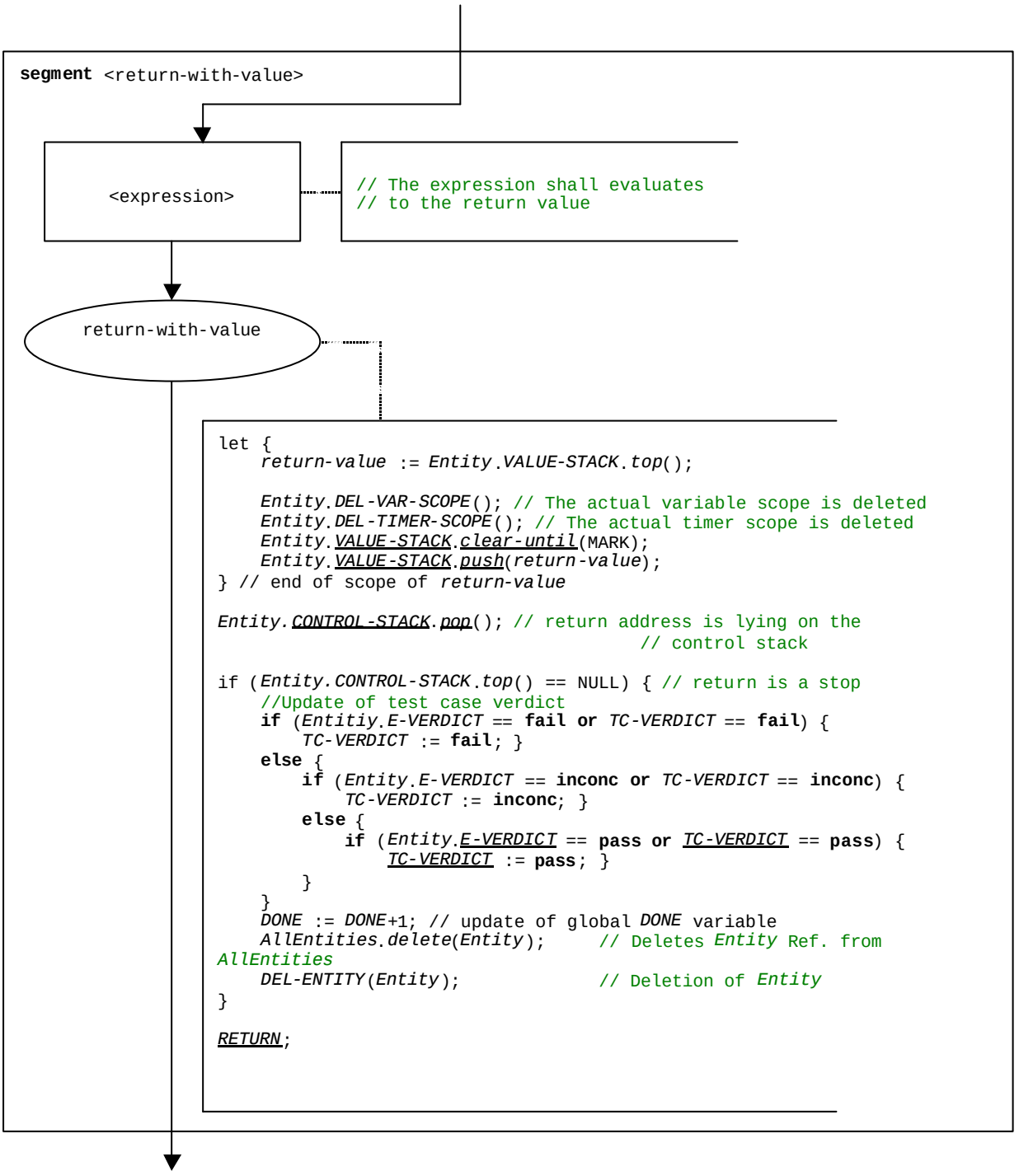


Figure B.100: Flow graph segment <return-with-value>

B.3.7.39.2 Flow graph segment <return-without-value>

The flow graph segment `<return-without-value>` in figure B.101 defines the execution of a **`return`** statement that returns no value.

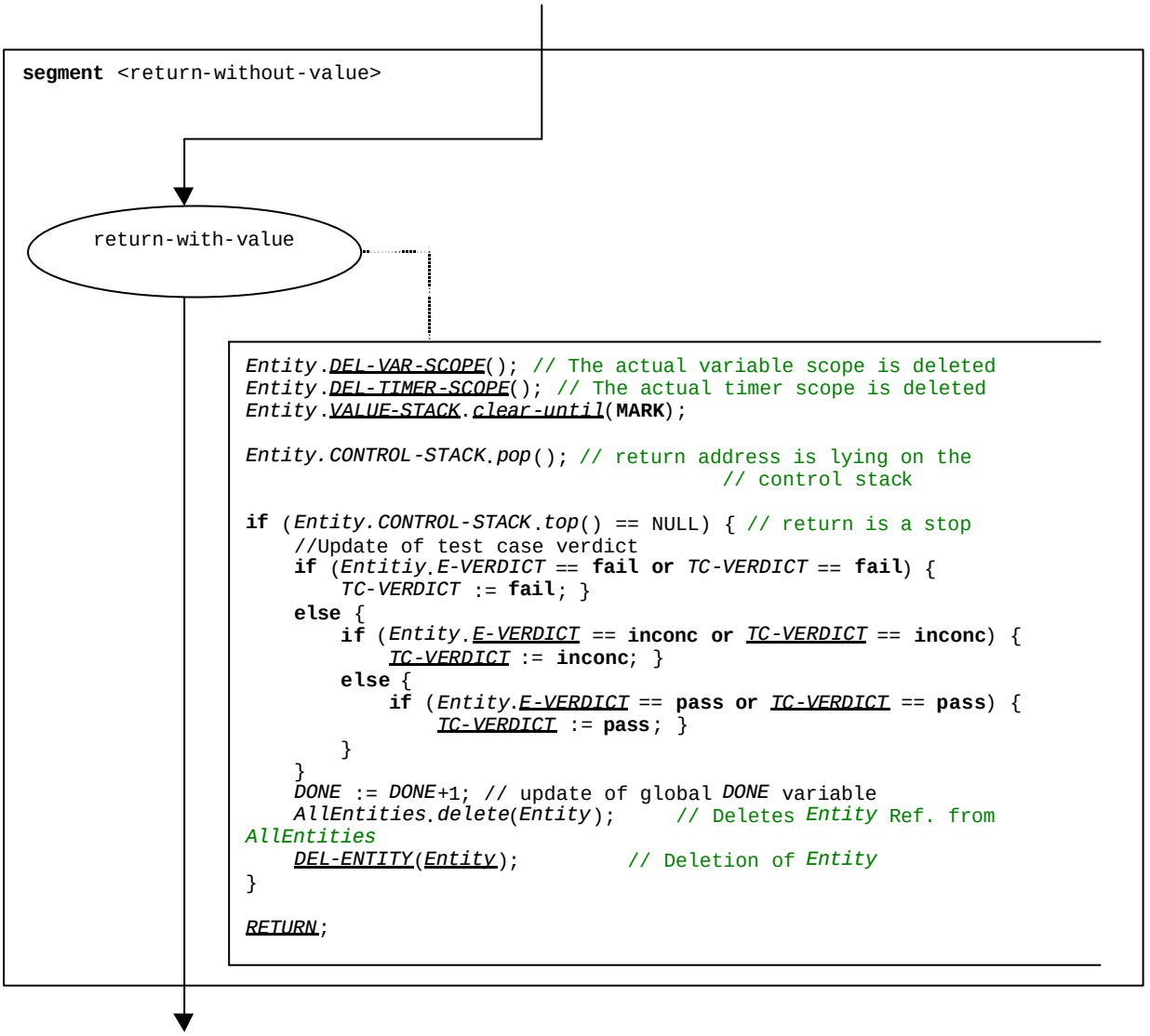


Figure B.101: Flow graph segment <return-without-value>

B.3.7.40 Running-all-components operation

The **running-all-components** operation refers to the usage of the keywords **all components** in the **running** component operation (Clause 42). The **running-all-components** operation can only be called by the **mtc**. It allows checking whether all parallel test components of a test case are running. The syntactical structure of the **running-all-components** operation is:

```
all component.running
```

The execution of the **running-all-components** operation is defined by the flow graph segment <running-all-comp-op> in figure B.102.

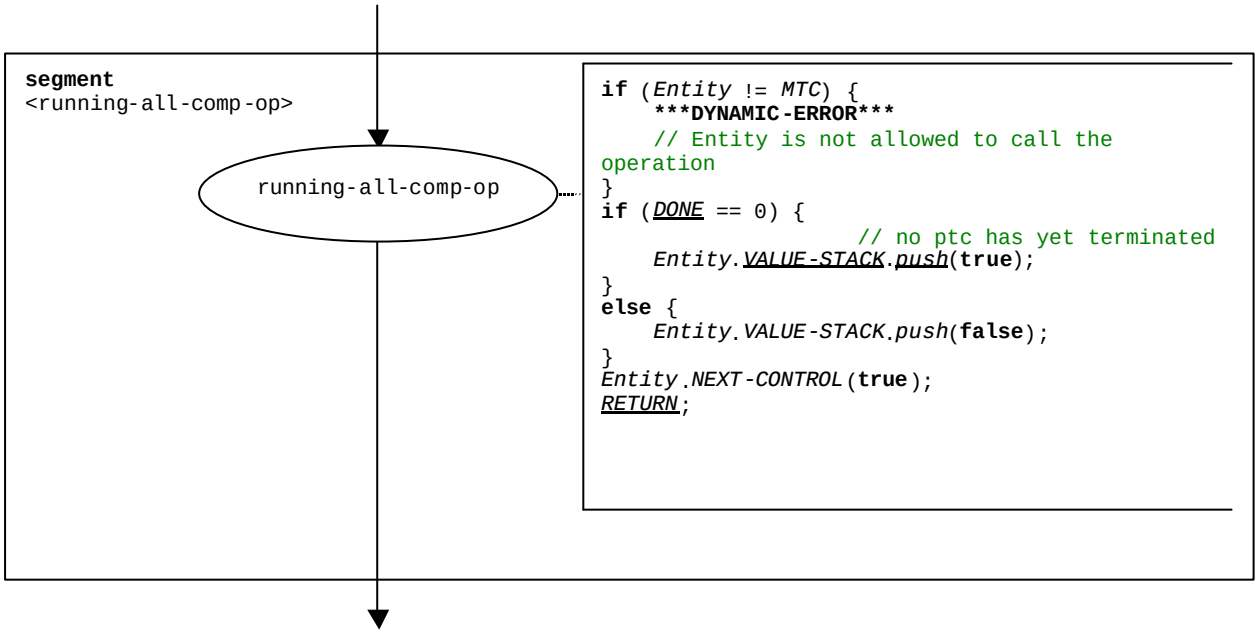


Figure B.102: Flow graph segment <running-all-comp-op>

B.3.7.41 Running-any-component operation

The **running-any-component** operation refers to the usage of the keywords **any component** in the **running** component operation (Clause 42). The **running-any-components** operation can only be called by the **mtc**. It allows checking if at least one parallel test component of a test case is still running. The syntactical structure of the **running-any-components** operation is:

```
any component.running
```

The execution of the **running-any-components** operation is defined by the flow graph segment <running-any-comp-op> in figure B.103.

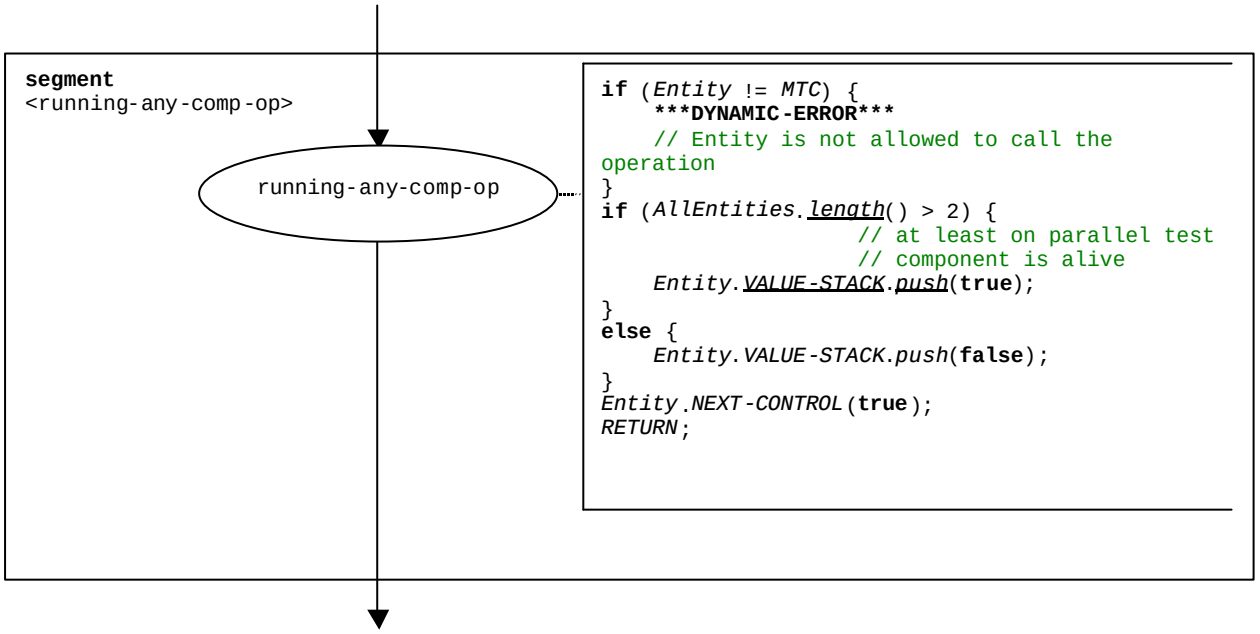


Figure B.103: Flow graph segment <running-any-comp-op>

B.3.7.42 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 stopped. Using a component reference identifies the component to be checked. The reference may be stored in a variable or be returned by a function. For simplicity this is considered to be an expression that evaluates to a component reference.

The flow graph segment <running-component-op> in figure B.104 defines the execution of the **running** component operation.

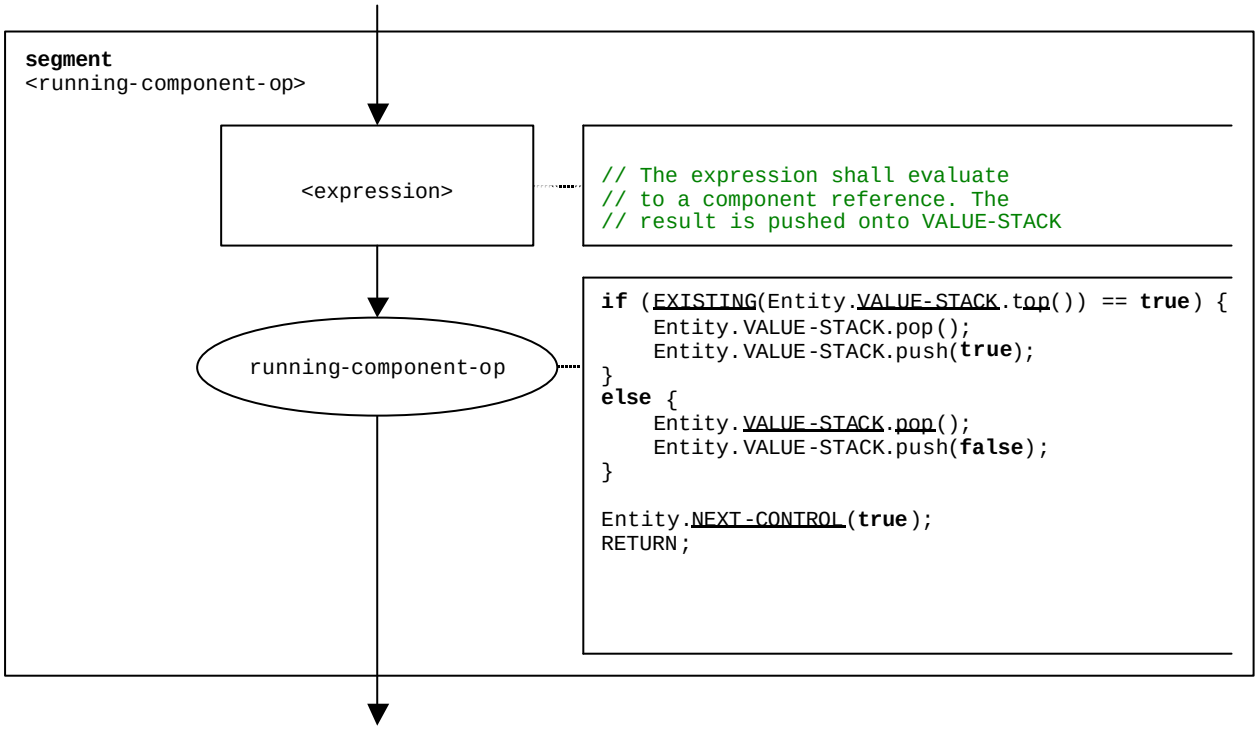


Figure B.104: Flow graph segment <running-component-op>

B.3.7.43 Running timer operation

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

```
<timerId>.running
```

The flow graph segment <running-timer-op> in figure B.105 defines the execution of the **running** timer operation.

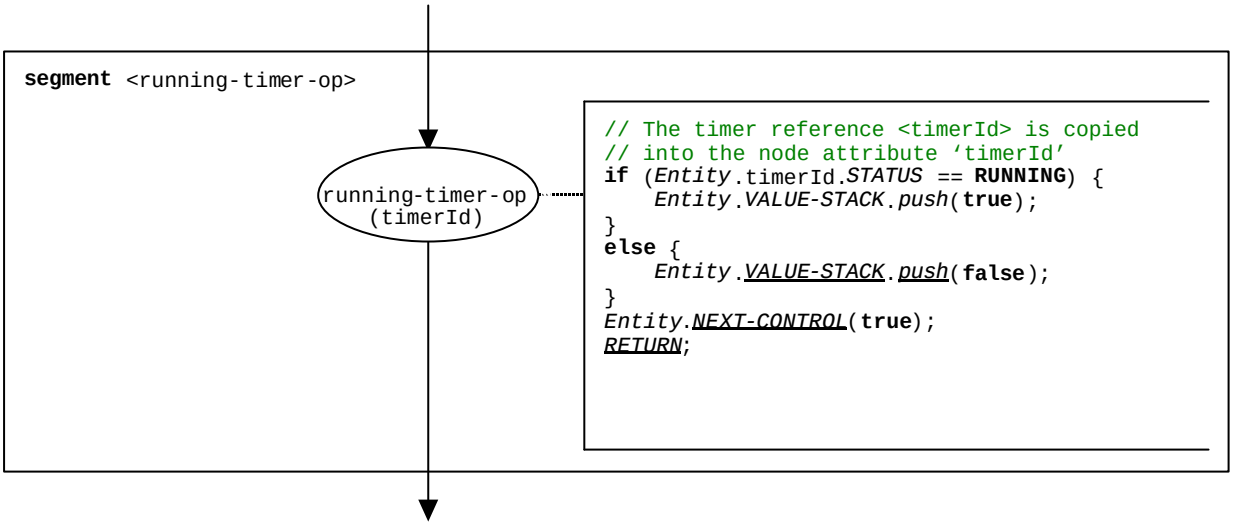


Figure B.105: Flow graph segment <running-timer-op>

B.3.7.44 Send operation

The syntactical structure of the send operation is:

```
<portId>.send (<send-spec>) [to <component_expression>]
```

The optional <component_expression> in the to clause refers to the receiver entity. It may be provided in form of a variable value or the return value of a function.

The flow graph segment <send-op> in figure B.106 defines the execution of a **send** operation.

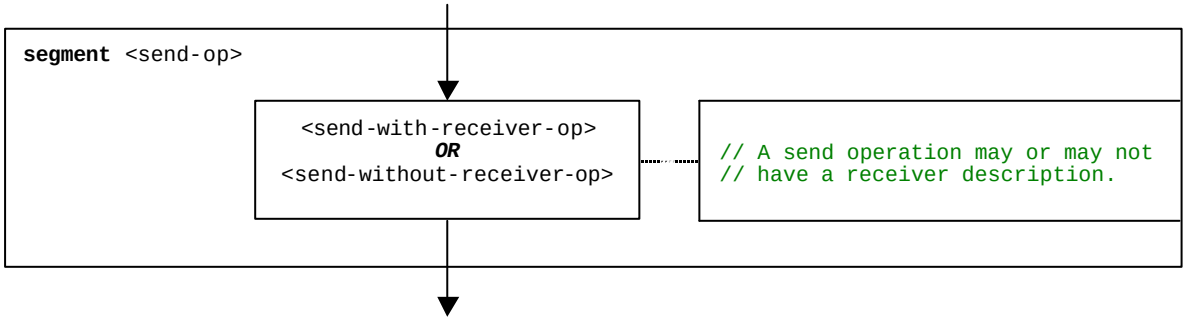


Figure B.106: Flow graph segment <send-op>

B.3.7.44.1 Flow graph segment <send-with-receiver-op>

The flow graph segment `<send-with-receiver-op>` in figure B.107 defines the execution of a **`send`** operation where the receiver is specified in form of an expression.

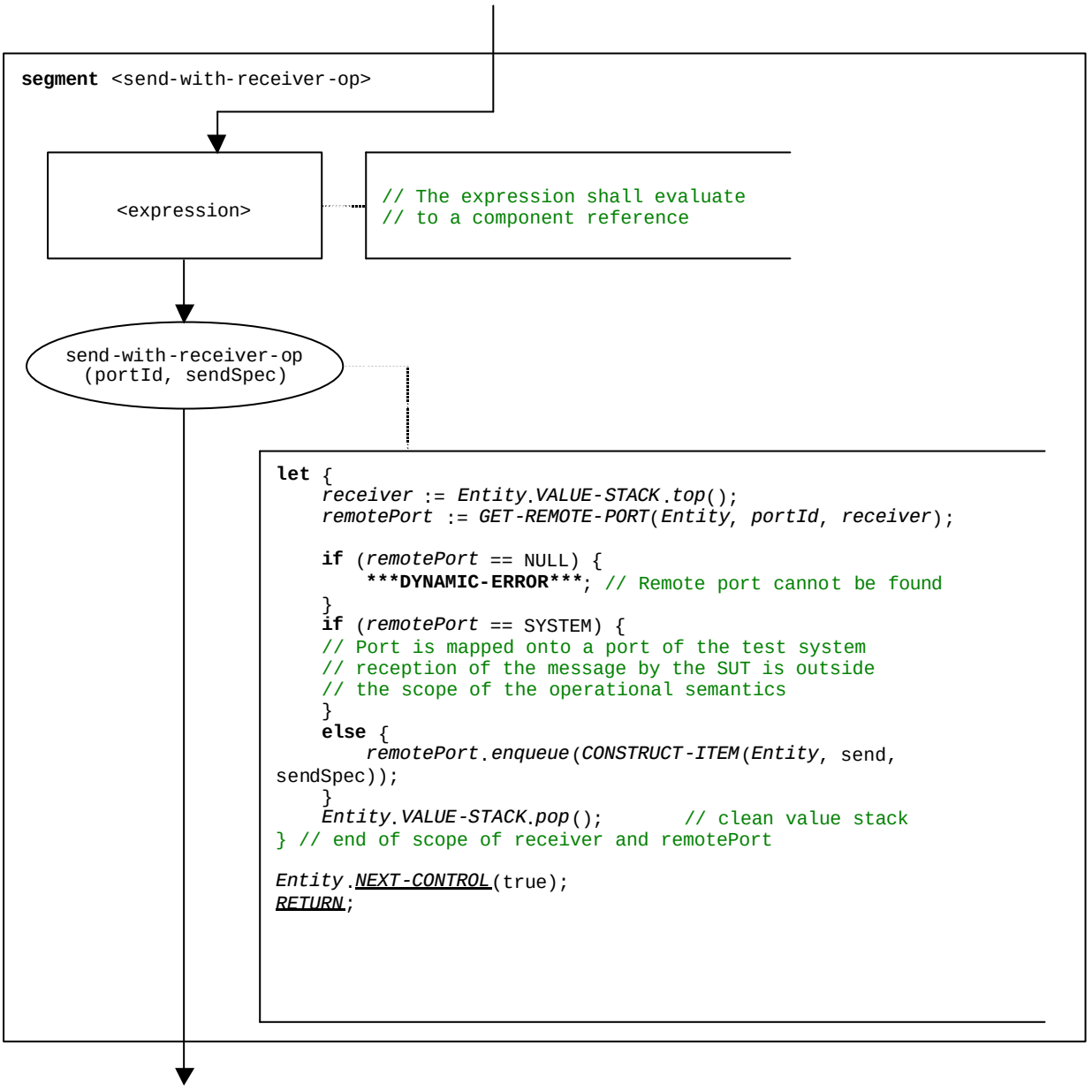


Figure B.107: Flow graph segment <send-with-receiver-op>

B.3.7.44.2 Flow graph segment <send-without-receiver-op>

The flow graph segment <send-without-receiver-op> in figure B.108 defines the execution of a **send** operation without **to**-clause.

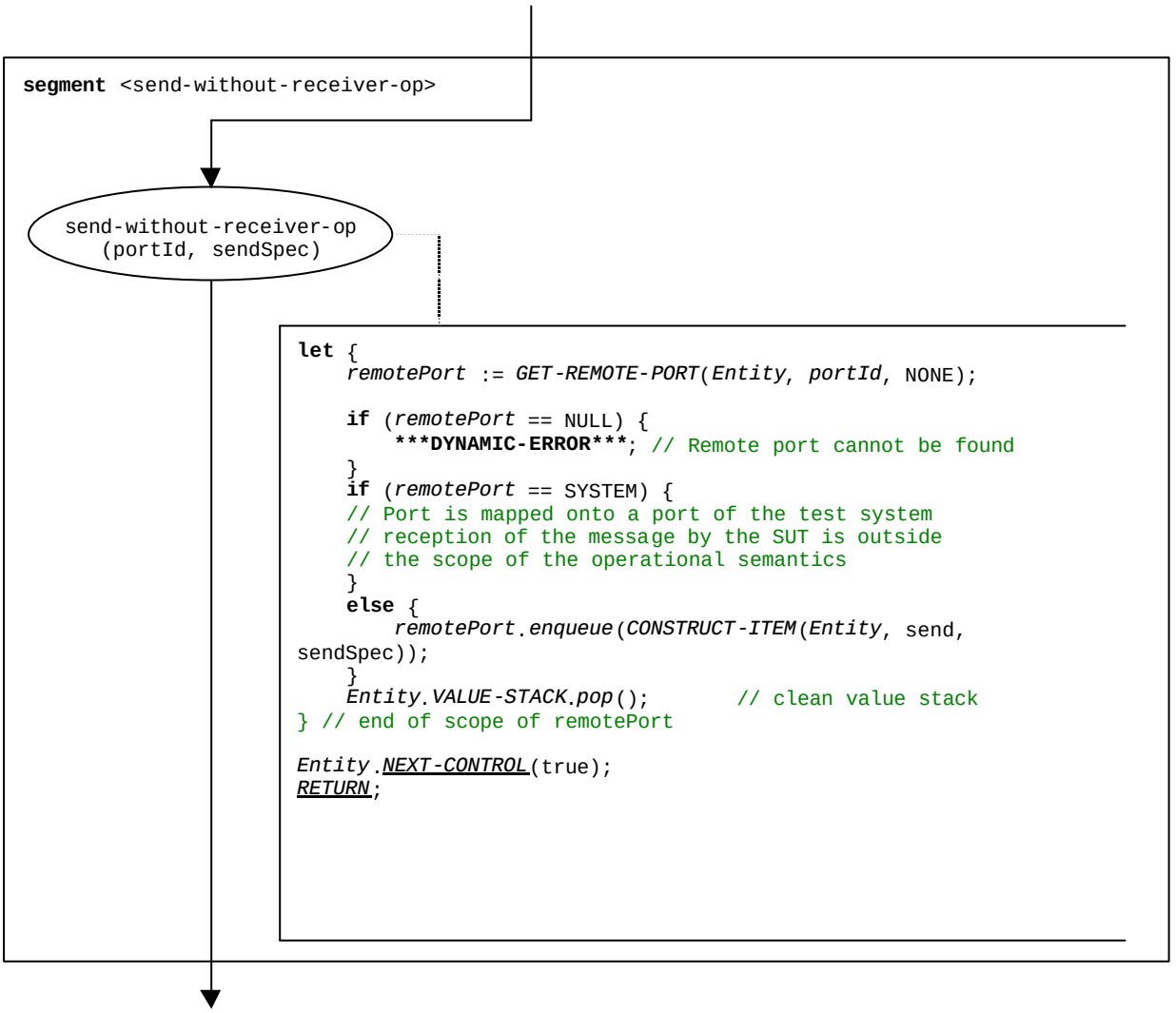


Figure B.108: Flow graph segment <send-without-receiver-op>

B.3.7.45 Self operation

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

```
self
```

The flow graph segment <self-op> in figure B.109 defines the execution of the **self** operation.

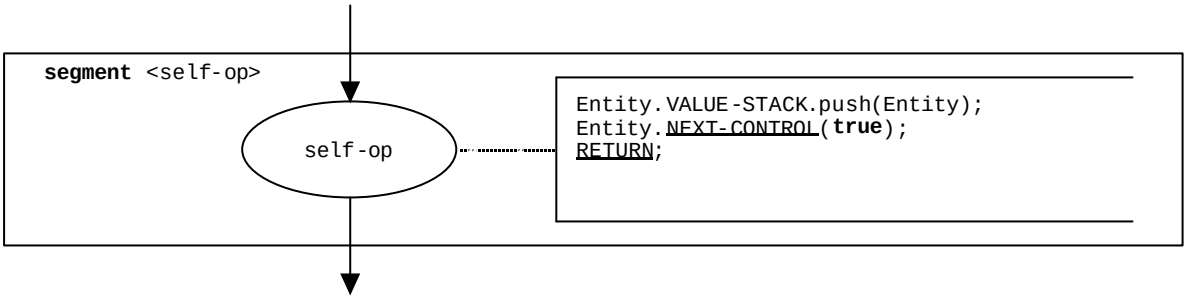


Figure B.109: Flow graph segment <self-op>

B.3.7.46 Start component operation

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

<component_expression>.**start**(<function-name>(<act-par-desc$_1$>, … , <act-par-desc$_n$>))

The **start** component operation starts a newly created 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. For simplicity this is considered to be an expression that evaluates to a component reference.

The <function-name> denotes to the name of the function that defines the behaviour of the new component and <act-par-descr$_1$>, …, <act-par-descr$_n$> provide the description of the actual parameter values of <function-name>. In case of a value parameter the description of an actual parameter may be provided in form of an expression that has to be evaluated before the call can be executed. The handling of formal and actual parameter is similar to their handling in function calls (Clause B.3.7.22).

The flow graph segment <start-component-op> in figure B.110 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.

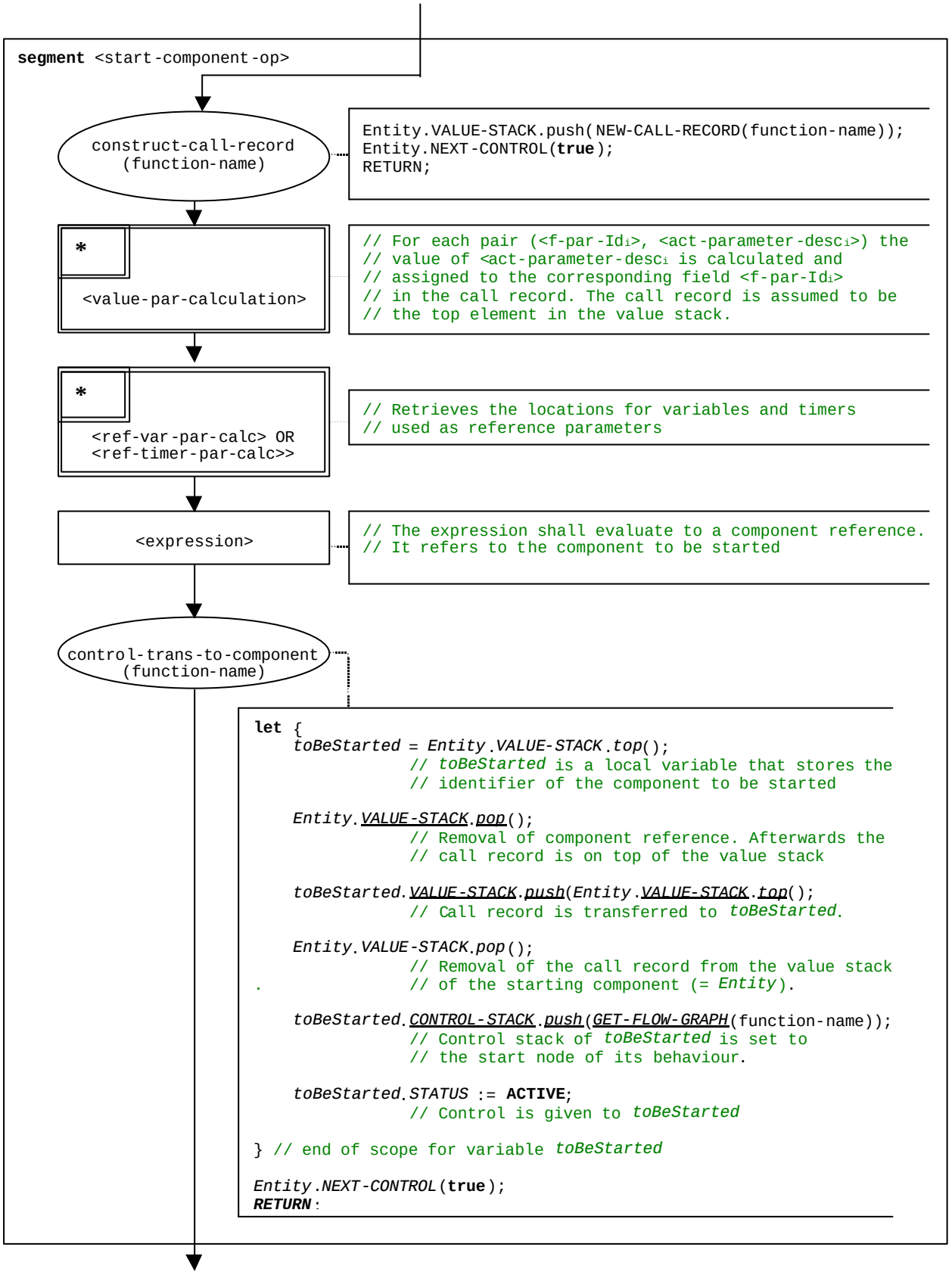


Figure B.110: Flow graph segment <start-component-op>

B.3.7.47 Start port operation

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

```
<portId>.start
```

The flow graph segment <start-port-op> in figure B.111 defines the execution of the **start** port operation.

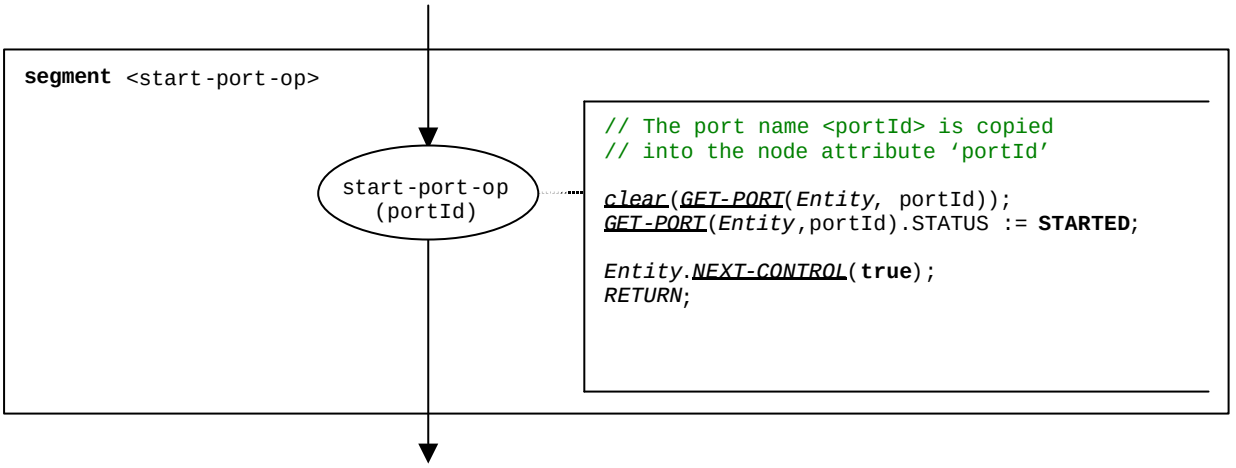


Figure B.111: Flow graph segment <start-port-op>

B.3.7.48 Start timer operation

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

```
<timerId>.start [(<float_expression>)]
```

The optional <float_expression> parameter of the timer **start** operation denotes the optional duration with which the timer shall be started. It is an expression that 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 B.112 defines the execution of the **start** timer operation.

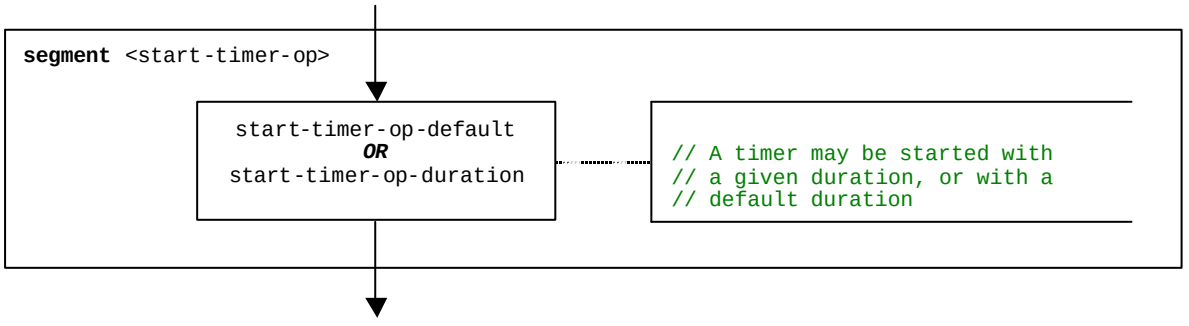


Figure B.112: Flow graph segment <start-timer-op>

B.3.7.48.1 Flow graph segment <start-timer-op-default>

The flow graph segment <start-timer-op-default> in figure B.113 defines the execution of the **`start`** timer operation with the default value.

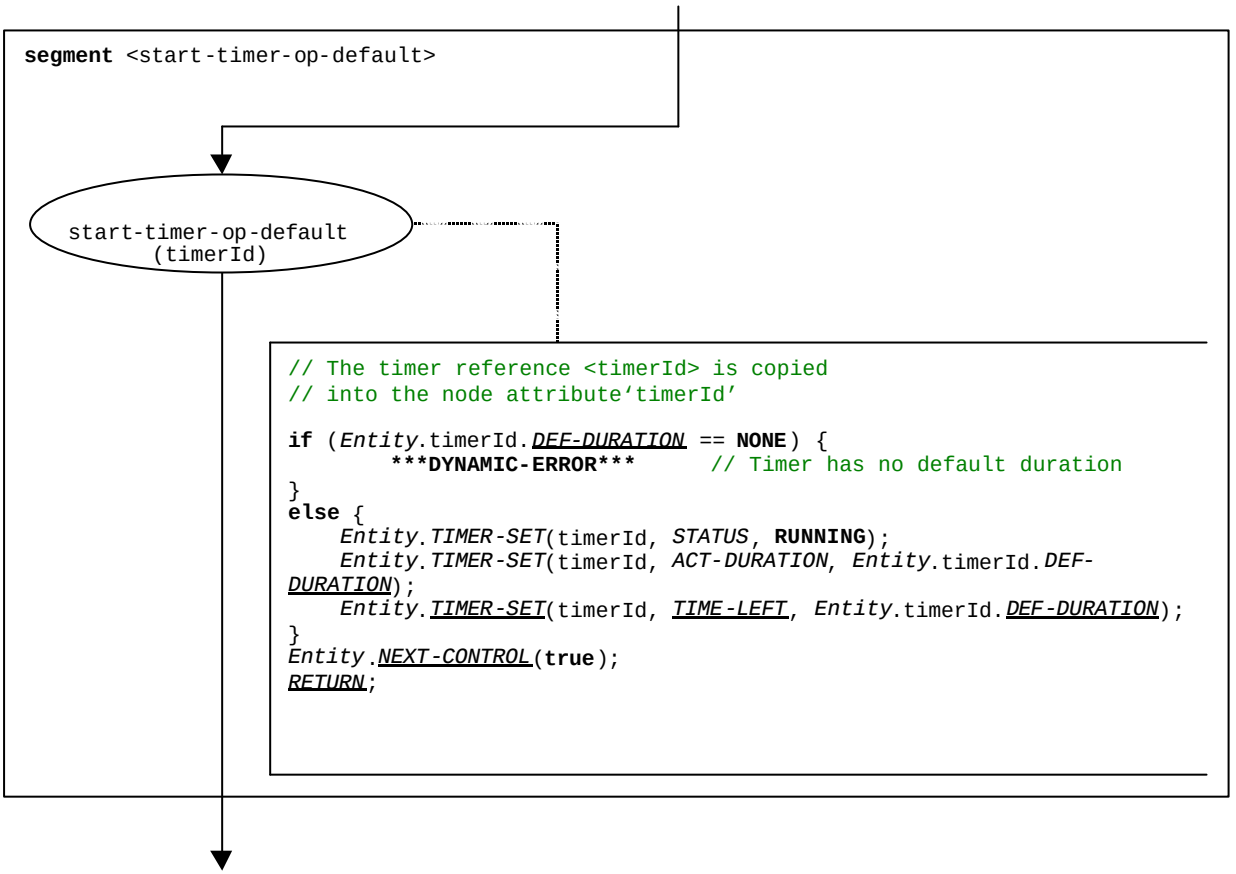


Figure B.113: Flow graph segment <start-timer-op-default>

B.3.7.48.2 Flow graph segment <start-timer-op-duration>

The flow graph segment <start-timer-op-duration> in figure B.114 defines the execution of the **`start`** timer operation with a provided duration.

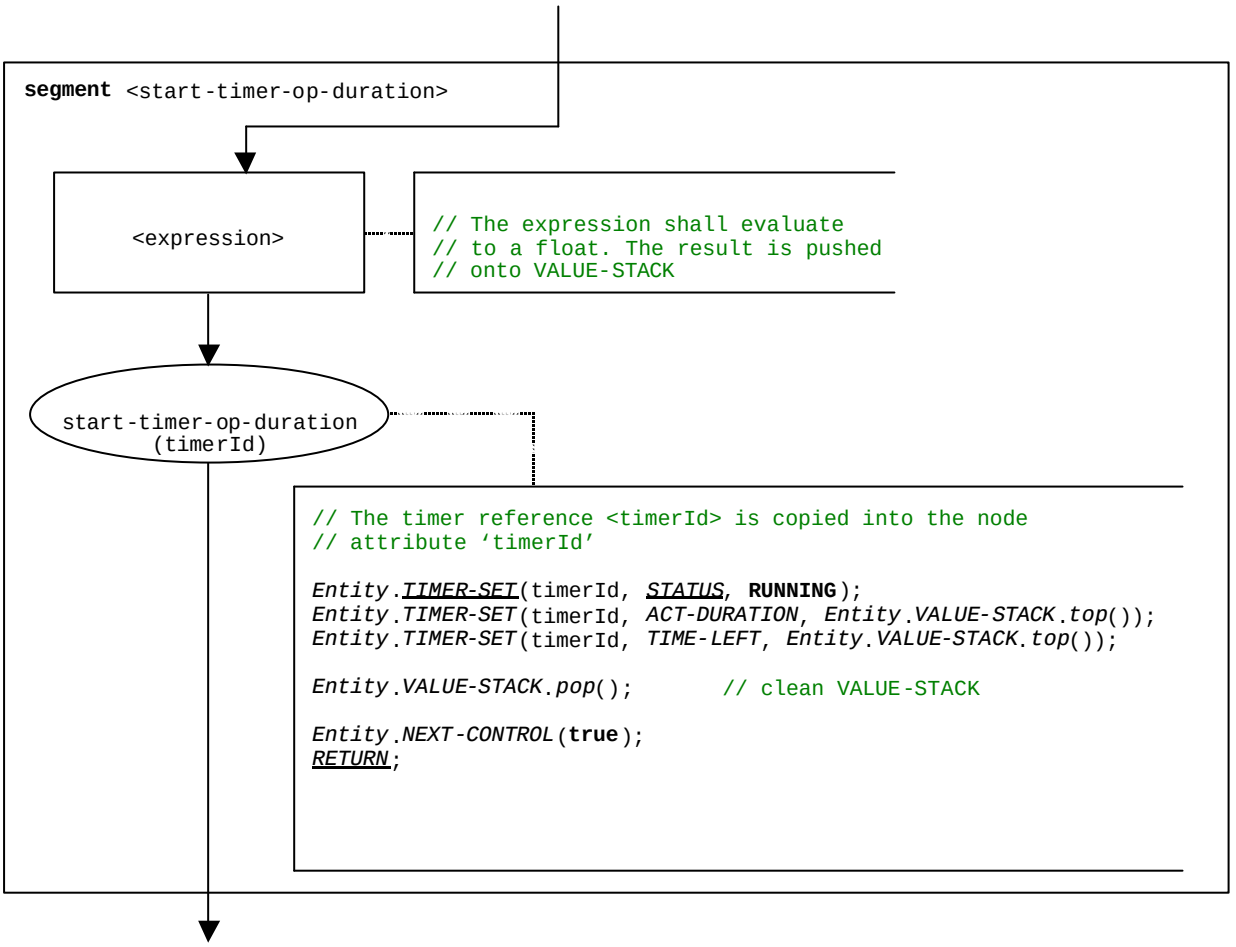


Figure B.114: Flow graph segment <start-timer-op-duration>

B.3.7.49 Statement block

The syntactical structure of a statement block is:

{ <statement$_1$>; … ; <statement$_n$> **}**

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.

The flow graph segment <statement-block> in figure B.115 defines the execution of a statement block.

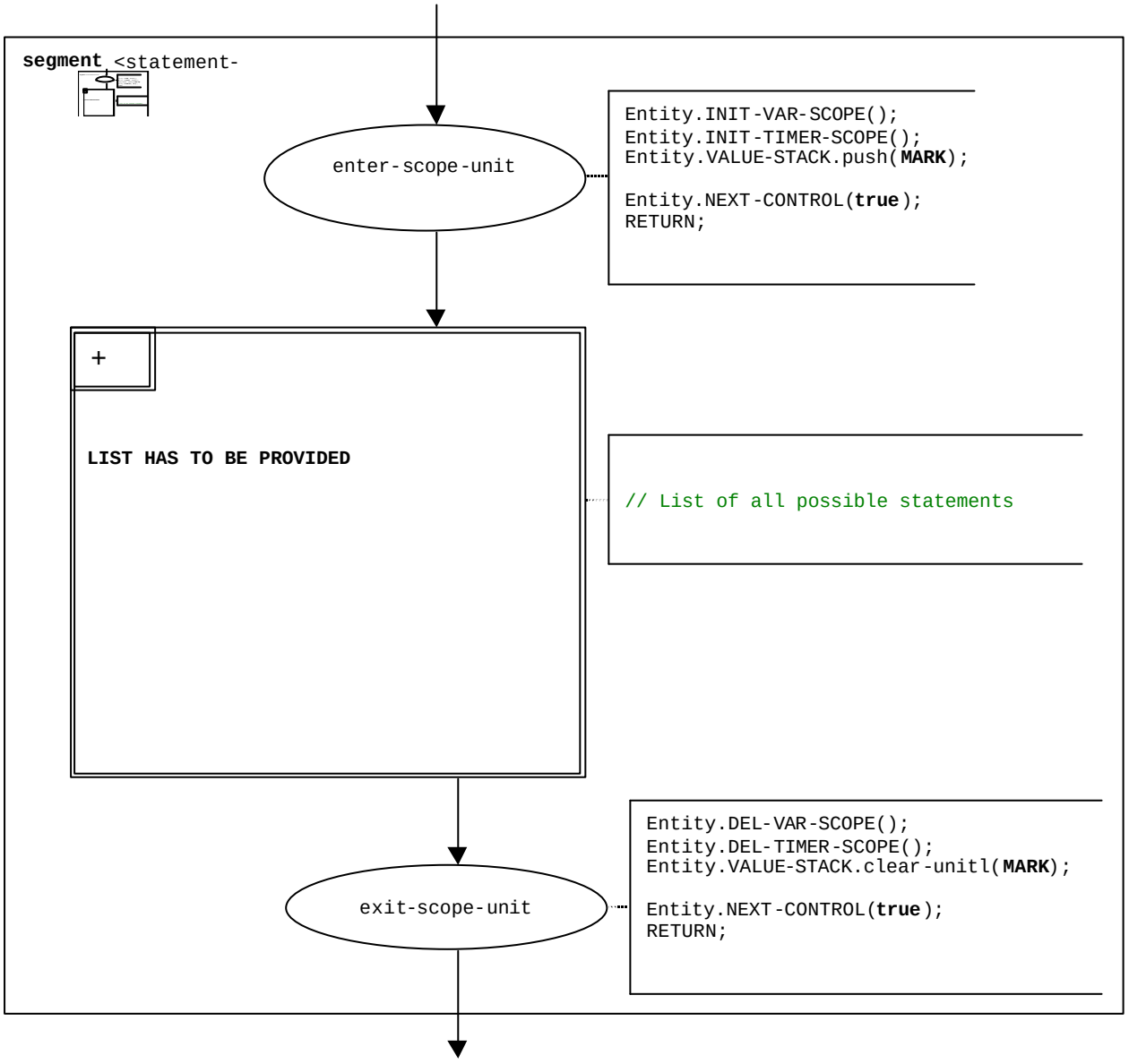


Figure B.115: Flow graph segment <statement-block>

B.3.7.50 Stop operation

The syntactical structure of the **stop** entity operation is:

```
stop
```

The effect of the stop operation depends on the entity that executes the stop operation:

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.

b) If the **stop** operation 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 stop operation is executed by a test component, the global test case verdict *TC-VERDICT* and the global *DONE* variable are updated. Then the component disappears completely from the module.

The flow graph segment <stop-entity-op> in figure B.116 defines the execution of the **stop** entity operation.

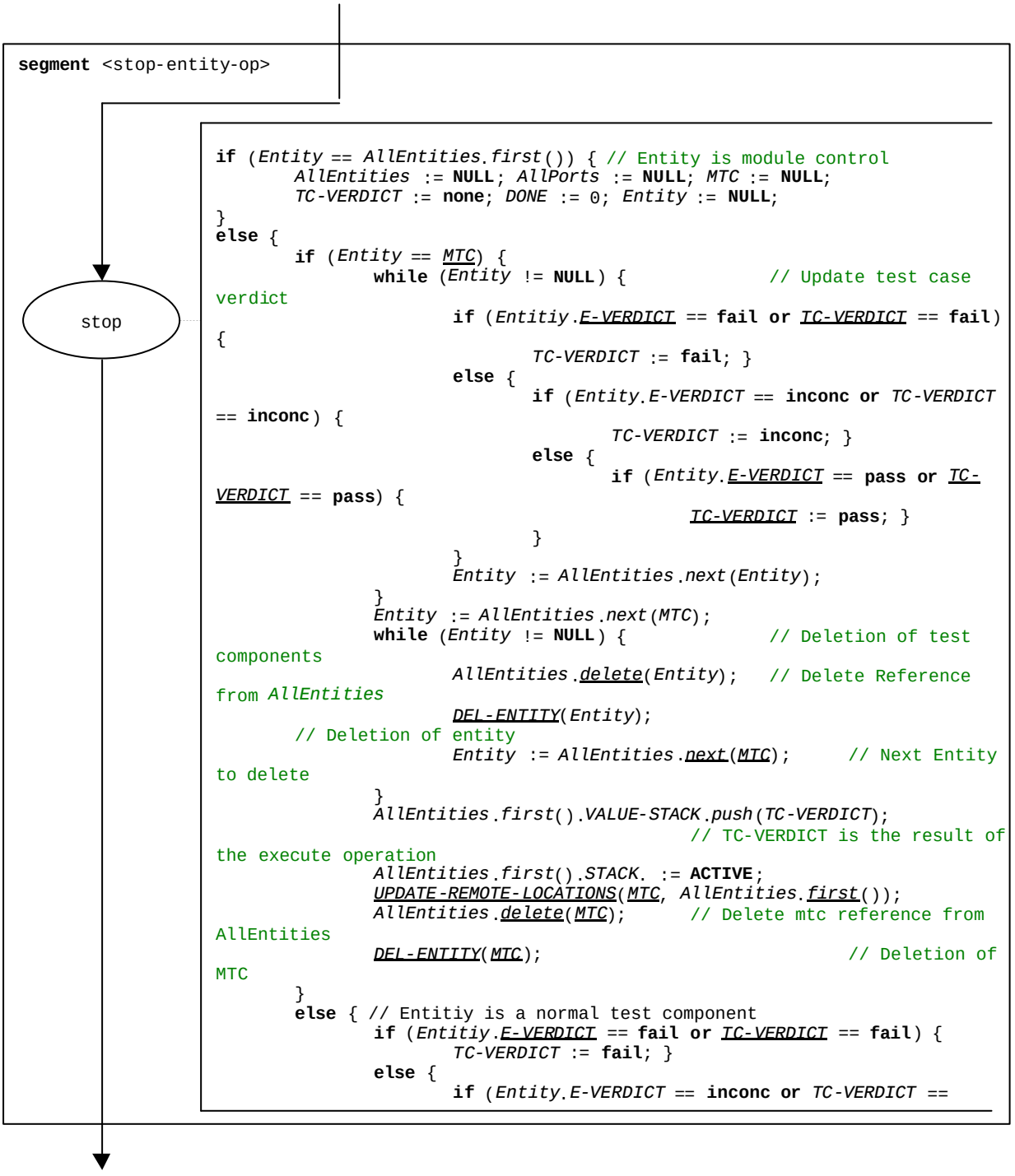


Figure B.116: Flow graph segment <stop-entity-op>

B.3.7.51 Stop port operation

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

```
<portId>.stop
```

The flow graph segment <stop-port-op> in figure B.117 defines the execution of the **stop** port operation.

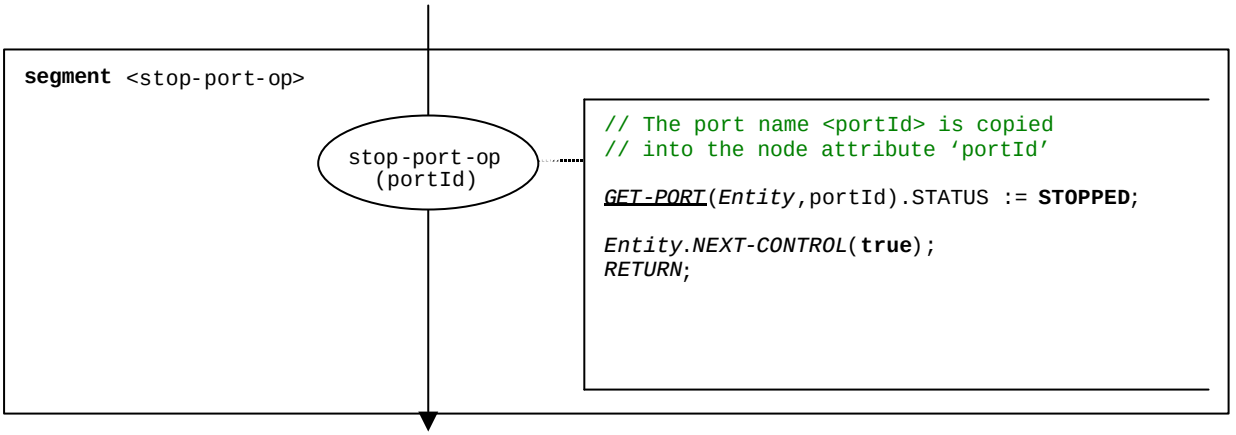


Figure B.117: Flow graph segment <stop-port-op>

B.3.7.52 Stop timer operation

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

```
<timerId>.stop
```

The flow graph segment <stop-timer-op> in figure B.118 defines the execution of the **stop** timer operation.

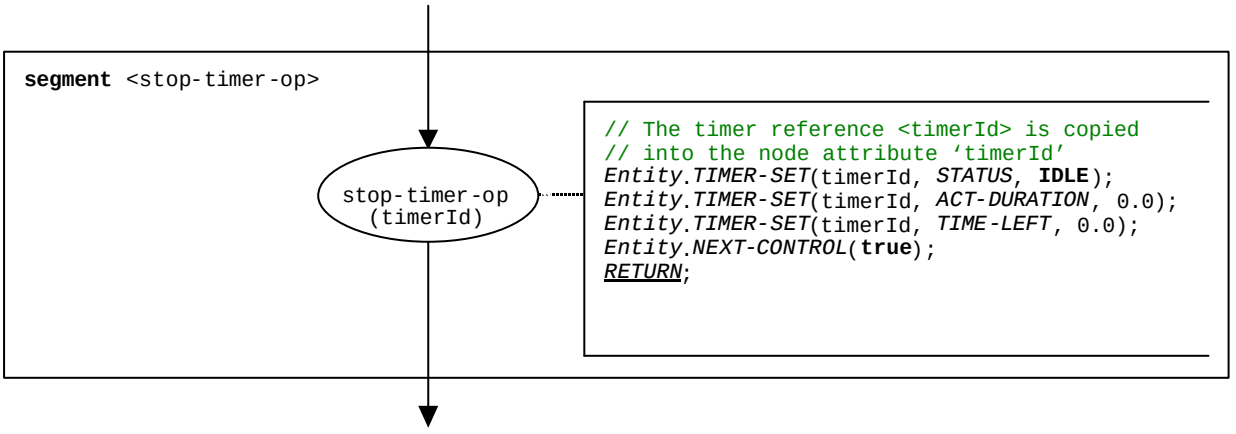


Figure B.118: Flow graph segment <stop-timer-op>

B.3.7.53 Sut.action operation

The syntactical structure of the **sut.action** operation is:

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

The flow graph segment <sut.action-op> in figure B.119 defines the execution of the **sut.action** operation.

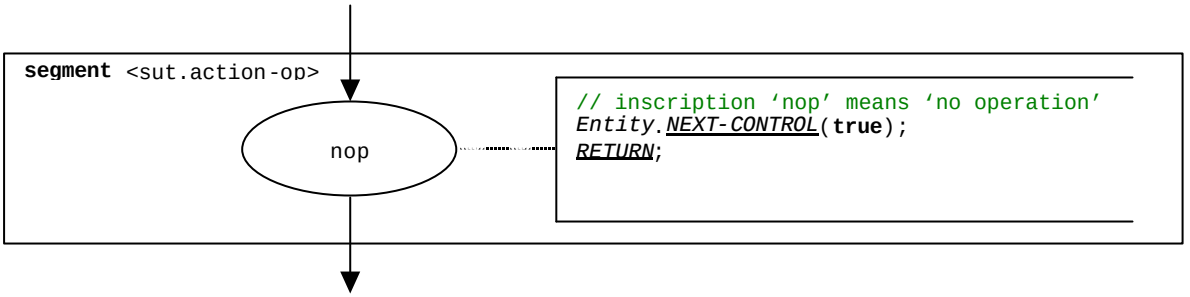


Figure B.119: Flow graph segment <sut.action-op>

NOTE: The <informal description> parameter of the **sut.action** operation has no meaning for the operational semantics and is therefore not represented in the flow graph segment.

B.3.7.54 System operation

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

```
system
```

The flow graph segment <system-op> in figure B.120 defines the execution of the **system** operation.

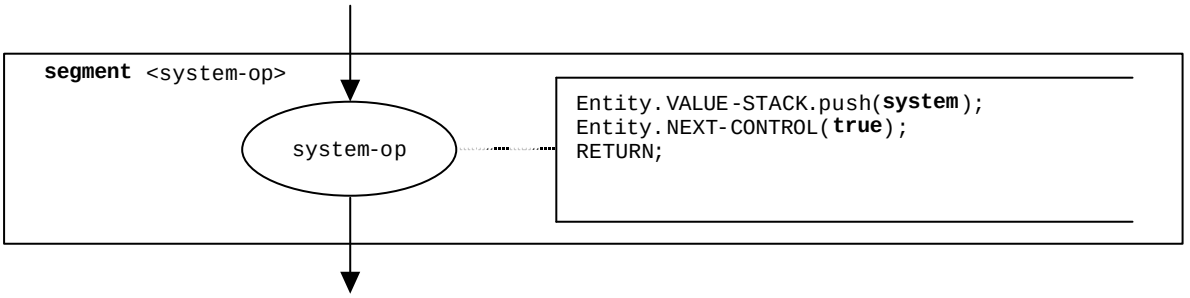


Figure B.120: Flow graph segment <system-op>

B.3.7.55 Timeout timer operation

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

```
<timerId>.timeout
```

The flow graph segment <timeout-timer-op> in figure B.121 defines the execution of the **timeout** timer operation.

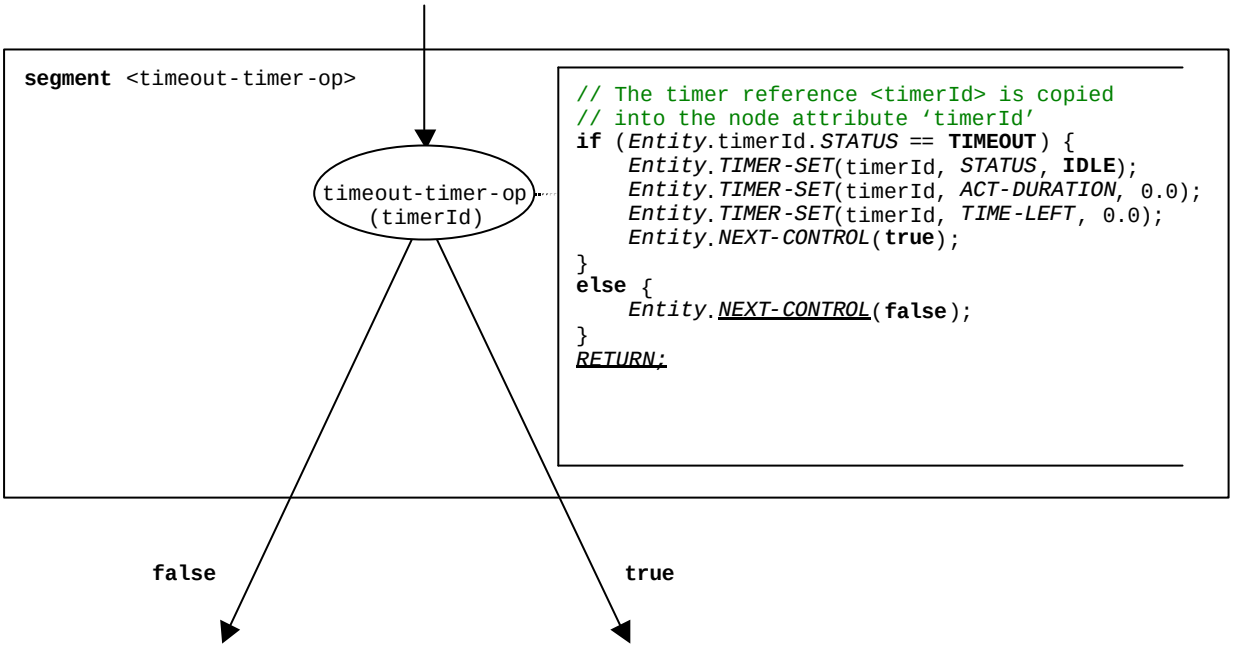


Figure B.121: Flow graph segment <running-timer-op>

NOTE: A **timeout** operation is embedded in an **alt** statement. Depending on whether the **timeout** evaluates to **true** or **false**, either execution continues with the statement that follows the **timeout** operation (**true** branch), or the next alternative in the **alt** statement has to be checked (**false** branch).

B.3.7.56 Unmap operation

The syntactical structure of a 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 component to which the <portId1> belongs is referenced by means of the component reference <component_expression>. The reference may be stored in variables or is returned by a function. For simplicity it is considered to be an expression that evaluates to a component reference. Thus, 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 <map-op> shown in figure B.122.

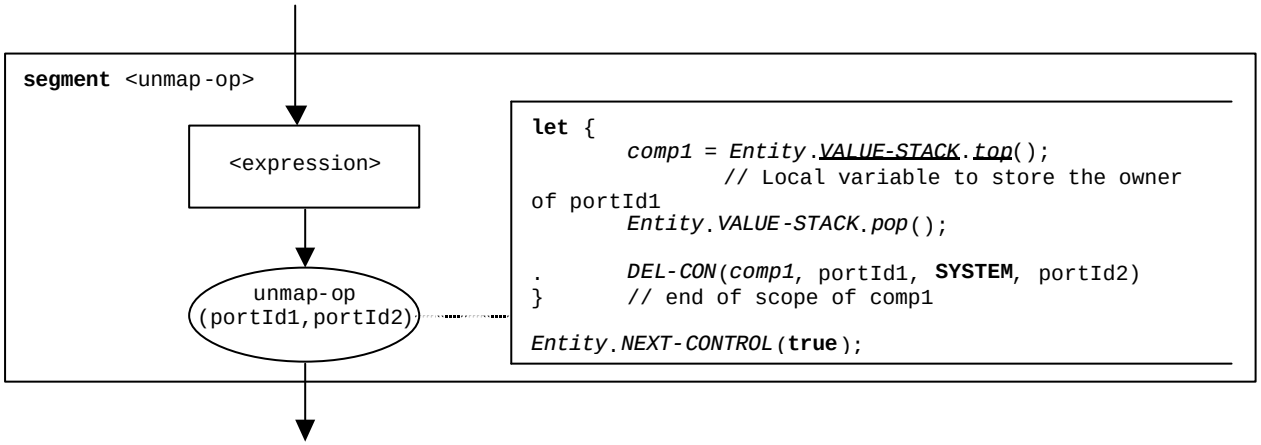


Figure B.122: Flow graph segment <unmap-op>

B.3.7.57 Verdict.get operation

The syntactical structure of the **verdict.get** operation is:

verdict.get

The flow graph segment <verdict.get-op> in figure B.123 defines the execution of the **verdict.get** operation.

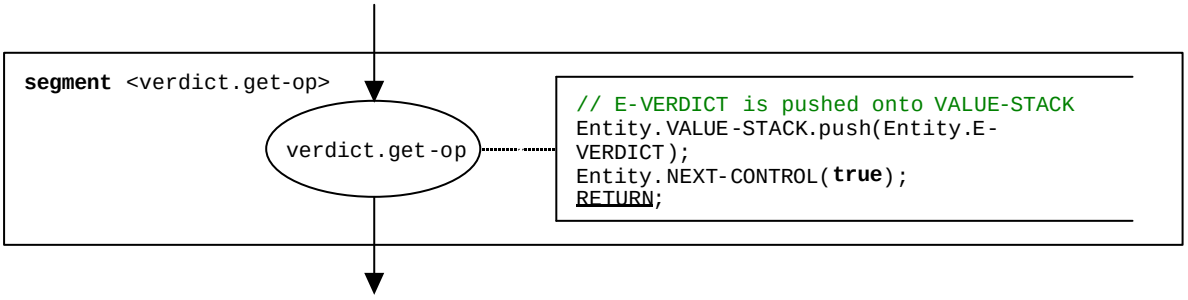


Figure B.123: Flow graph segment <verdict.get-op>

B.3.7.58 Verdict.set operation

The syntactical structure of the **verdict.set** operation is:

verdict.set(<verdicttype_expression>)

NOTE: The <verdicttype_expression> parameter of the **verdict.set** 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 **verdict.set** operation is applied.

The flow graph segment <verdict.set-op> in figure B.124 defines the execution of the **verdict.set** operation.

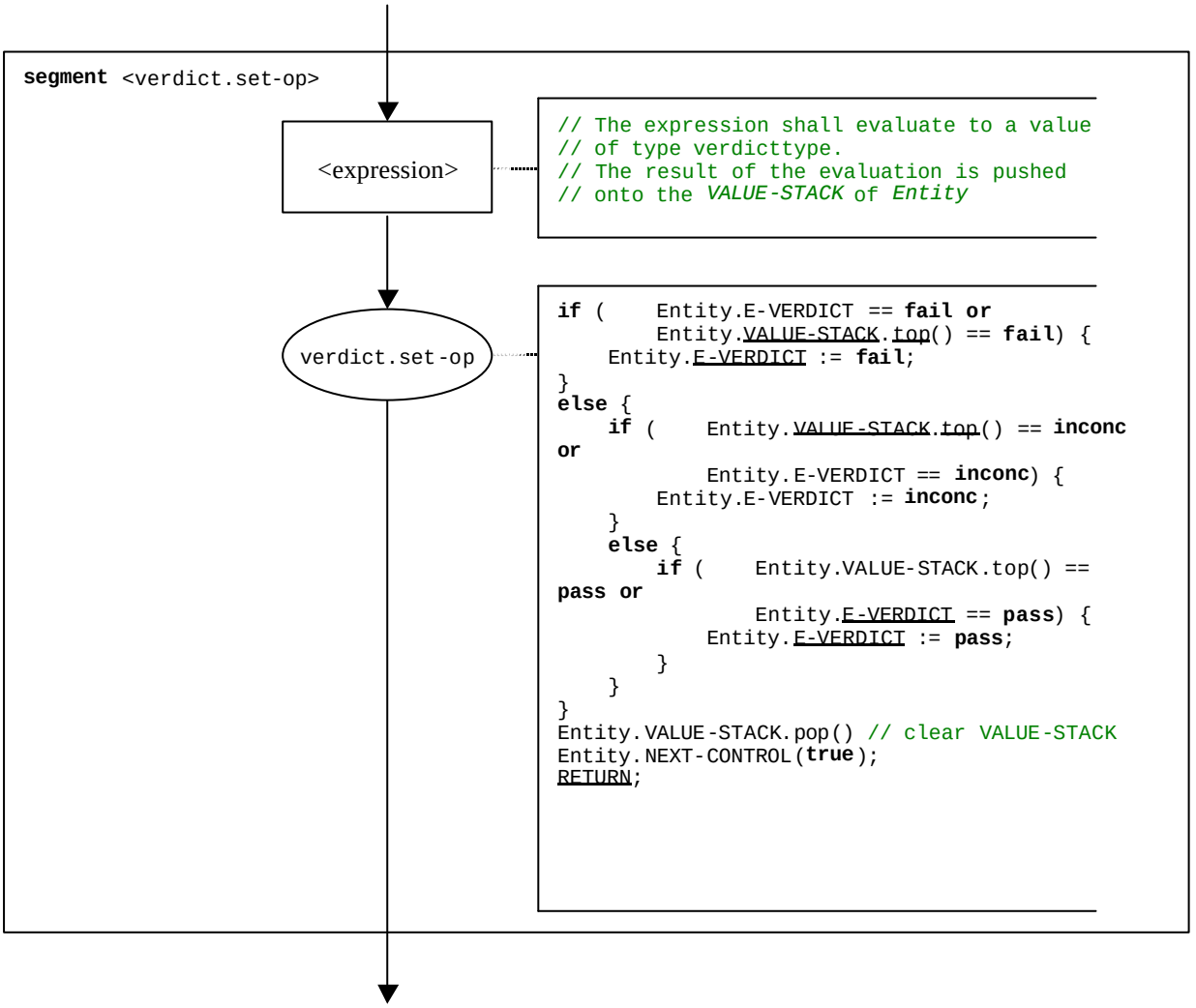


Figure B.124: Flow graph segment <verdict.set-op>

B.3.7.59 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 B.125.

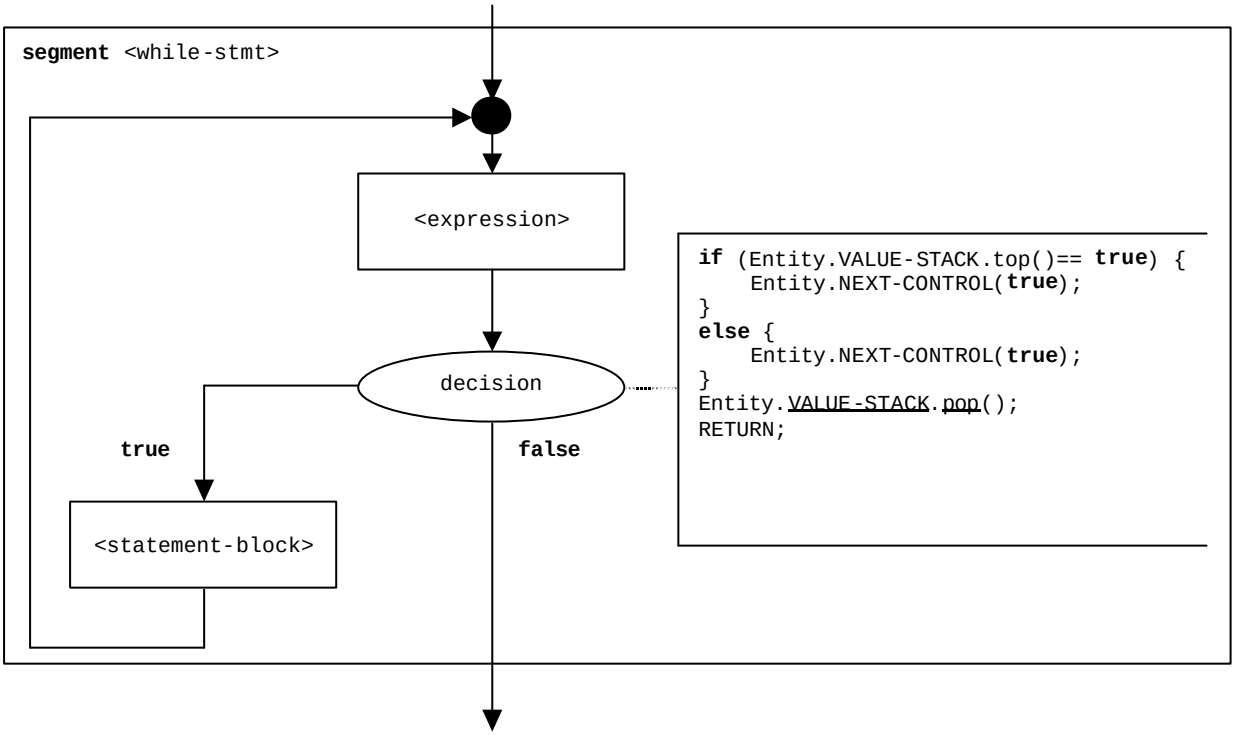


Figure B.125: Flow graph segment <while-stmt>

B.3.8 Lists of operational semantic components

B.3.8.1 Functions and states

Name	Description	Reference
NEXT	Retrieves the successor node of a given node in a flow graph.	Clause B.3.1.6
GET-FLOW-GRAPH	Retrieves the start node of a flow graph	Clause B.3.2.6
MTC	Reference to **mtc** in module state	Clause B.3.3.1.1
TC-VERDICT	Actual test case verdict in module state	Clause B.3.3.1.1
DONE	Number of terminated test components (part of module state)	Clause B.3.3.1.1
append	List operation 'append': appends an item as last element to a list	Clause B.3.3.1.1
delete	List operation 'delete': deletes an item from a list	Clause B.3.3.1.1
first	List operation 'first': returns the first element of a list	Clause B.3.3.1.1
	Queue operation 'first': returns the first element of a queue	Clause B.3.3.3.2
length	List operation 'length': returns the length of a list	Clause B.3.3.1.1
STATUS	Status (**ACTIVE** or **BLOCKED**) of module control or a test component	Clause B.3.3.2.1
	Status (**IDLE**, **RUNNING** or **TIMEOUT**) of a timer	Clause B.3.3.2.4
	Status (**STARTED** or **STOPPED**) of a port	Clause B.3.3.3.2
E-VERDICT	Local test verdict of a test component	Clause B.3.3.2.1
CONTROL-STACK	Stack of flow graph nodes denoting the actual control state of an entity	Clause B.3.3.2.1
VALUE-STACK	Stack of values for the storage of results of expressions, operands, operations and functions.	Clause B.3.3.2.1
push	Stack operation 'push': pushes an item onto a stack	Clause B.3.3.2.1
pop	Stack operation 'pop': pops an item from a stack	Clause B.3.3.2.1
top	Stack operation 'top': returns the top item from a stack	Clause B.3.3.2.1
clear	Stack operation 'clear': clears a stack	Clause B.3.3.2.1
	Queue operation 'clear': removes all elements from a queue	Clause B.3.3.3.2
clear-until	Stack operation 'clear-until': pops items until a specific item is top element in the stack.	Clause B.3.3.2.1
NEW-ENTITY	Creates a new entity state	Clause B.3.3.2.1
VAR-SET	Setting the value of a variable	Clause B.3.3.2.4
TIMER-SET	Setting values of a timer	Clause B.3.3.2.4
DEF-DURATION	Default Duration of a timer	Clause B.3.3.2.4
ACT-DURATION	Duration with which an active timer has been started	Clause B.3.3.2.4
TIME-LEFT	Time a running timer has left to run before a it times out	Clause B.3.3.2.4
INIT-VAR	Creates a new variable binding	Clause B.3.3.2.4
INIT-TIMER	Creates a new timer binding	Clause B.3.3.2.4
GET-VAR-LOC	Retrieves location of a variable	Clause B.3.3.2.4
GET-TIMER-LOC	Retrieves location of a timer	Clause B.3.3.2.4
INIT-VAR-LOC	Creates a new variable binding with an existing location	Clause B.3.3.2.4
INIT-TIMER-LOC	Creates a new timer binding with an existing location	Clause B.3.3.2.4
INIT-VAR-SCOPE	Initializes a new variable scope	Clause B.3.3.2.4
INIT-TIMER-SCOPE	Initializes a new timer scope	Clause B.3.3.2.4
DEL-VAR-SCOPE	Deletes a variable scope	Clause B.3.3.2.4
DEL-TIMER-SCOPE	Deletes a timer scope	Clause B.3.3.2.4
NEW-PORT	Creates a new port	Clause B.3.3.3.2
GET-PORT	Retrieves a port reference	Clause B.3.3.3.2
GET-REMOTE-PORT	Retrieves the reference of a remote port	Clause B.3.3.3.2
ADD-CON	Adds a connection to a port state	Clause B.3.3.3.2
DEL-CON	Deletes a connection from a port state	Clause B.3.3.3.2
enqueue	Queue operation 'enqueue': puts an item as last element into a queue	Clause B.3.3.3.2
dequeue	Queue operation 'dequeue': deletes the first element from a queue	Clause B.3.3.3.2
DEL-ENTITY	Deletes an entity from a module state	Clause B.3.3.4
EXISTING	Checks whether a test component exists or not	Clause B.3.3.4
UPDATE-REMOTE-REFERENCES	Updates timers and variables with the same location in different entities to the same value.	Clause B.3.3.4
CONSTRUCT-ITEM	Constructs an item to be sent	Clause B.3.4.3
MATCH-ITEM	Checks if a received message, call, reply or exception matches with a receiving operation	Clause B.3.4.4
RETRIEVE-INFO	Retrieves information from a received message, call, reply or exception	Clause B.3.4.4
NEW-CALL-RECORD	Creates a call record for a function call	Clause B.3.5.1
INIT-FLOW-GRAPHS	Initializes the flow graph handling	Clause B.3.6.1
GET-UNIQUE-ID	Returns a new unique identifier when it is called	Clause B.3.6.1

Name	Description	Reference
CONTINUE-COMPONENT	The actual component continues its execution	Clause B.3.6.1
RETURN	Returns the control to the module evaluation procedure defined in clause B.3.6	Clause B.3.6.1
*****DYNAMIC-ERROR*****	Describes the occurrence of a dynamic error	Clause B.3.6.1

B.3.8.2 Special keywords

Keyword	Description	Reference
MARK	Used as mark for *VALUE-STACK*	Clause B.3.3.2
ACTIVE	*STATUS* of an entity state	Clause B.3.3.2
BLOCKED	*STATUS* of an entity state	Clause B.3.3.2
NULL	Symbolic value for pointer and pointer-like types to indicate that nothing is addressed	
IDLE	*STATUS* of a timer state	Clause B.3.3.2.4
RUNNING	*STATUS* of a timer state	Clause B.3.3.2.4
TIMEOUT	*STATUS* of a timer state	Clause B.3.3.2.4
STARTED	*STATUS* of a port	Clause B.3.3.2.4
STOPPED	*STATUS* of a port	Clause B.3.3.2.4
NONE	Used to describe an undefined value	

B.3.8.3 Flow graph segments

Identifier	Related TTCN-3 construct	Reference	
		Figure	Clause
<alt-stmt>	alt statement	Figure B.25	Clause B.3.7.1
<alt-with-else>	alt statement	Figure B.26	Clause B.3.7.1
<alt-without-else>	alt statement	Figure B.27	Clause B.3.7.1
<assignment-stmt>	assignment statement	Figure B.29	Clause B.3.7.2
<b-call-with-receiver>	call	Figure B.35	Clause B.3.7.3.3
<b-call-without-receiver>	call	Figure B.36	Clause B.3.7.3.4
<b-call-with-rec-dur>	call	Figure B.37	Clause B.3.7.3.5
<b-call-without-rec-dur>	call	Figure B.38	Clause B.3.7.3.6
<blocking-call-op>	call	Figure B.31	Clause B.3.7.3
<call-op>	call	Figure B.30	Clause B.3.7.3
<catch-op>	catch	Figure B.39	Clause B.3.7.4
<catch-with-sender>	used in catch operation	Figure B.40	Clause B.3.7.4.1
<catch-without-sender>	used in catch operation	FigureB.41	Clause B.3.7.4.2
<clear-port-op>	clear port	Figure B.42	Clause B.3.7.5
<constant-declaration>	Declaration of a constant	Figure B.44	Clause B.3.7.7
<connect-op>	connect	Figure B.43	Clause B.3.7.6
<create-op>	create	Figure B.45	Clause B.3.7.8
<disconnect-op>	disconnect	Figure B.53	Clause B.3.7.12
<do-while-stmt>	do-while statement	Figure B.54	Clause B.3.7.13
<done-all-comp-op>	all component.done	Figure B.55	Clause B.3.7.14
<done-any-comp-op>	any component.done	Figure B.56	Clause B.3.7.15
<done-component-op>	done component	Figure B.57	Clause B.3.7.16
<execute-stmt>	execute	Figure B.58	Clause B.3.7.17
<execute-timeout>>	execute	FigureB.59	Clause B.3.7.17
<execute-without-timeout>>	execute	Figure B.60	Clause B.3.7.17
<expression>	Expression	Figure B.61	Clause B.3.7.18
<finalize-component-init>	Used in the behaviour of component type definitions	Figure B.66	Clause B.3.7.19
<for-stmt>>	for statement	Figure B.68	Clause B.3.7.21
<function-call>	Call of user defined functions	Figure B.69	Clause B.3.7.22
<func-op-call>	Used in <expression>	Figure B.64	Clause B.3.7.18.3
<getcall-op>	**getcall**	Figure B.74	Clause B.3.7.27
<getcall-with-sender>	used in **getcall** operation	Figure B.75	Clause B.3.7.27.1
<getcall-without-sender>	used in **getcall** operation	Figure B.76	Clause B.3.7.27.2
<getreply-op>	**getreply**	Figure B.76	Clause B.3.7.28
<getreply-with-sender>	used in **getreply** operation	Figure B.78	Clause B.3.7.28.1
<getreply-without-sender>	used in **getreply** operation	Figure B.79	Clause B.3.7.28.2
<goto-stmt>	**goto**	Figure B.80	Clause B.3.7.29
<if-else-stmt>	**if-else**	Figure B.80	Clause B.3.7.30
<if-with-else-branch>	**if-else**	Figure B.82	Clause B.3.7.30.1
<if-without-else-branch>	**if-else**	Figure B.83	Clause B.3.7.30.2
<init-component-scope>	Used in the behaviour of component type definitions	Figure B.67	Clause B.3.7.20
<label-stmt>	**label**	Figure B.84	Clause B.3.7.31
<lit-value>	Used in <expression>	Figure B.62	Clause B.3.7.18.1

Identifier	Related TTCN-3 construct	Reference	
		Figure	Clause
<log-stmt>	**log**	FigureB.85	Clause B.3.7.32
<map-op>	**map** operation	Figure B.86	Clause B.3.7.33
<mtc-op>	**mtc**	Figure B.87	Clause B.3.7.34
<nb-call-with-receiver>	**call**	Figure B.33	Clause B.3.7.3.1
<nb-call-without-receiver>	**call**	Figure B.34	Clause B.3.7.3.2
<non-blocking-call-op>	**call**	Figure B.32	Clause B.3.7.3
<operator-appl>	used in <expression>	Figure B.65	Clause B.3.7.18.4
<parameter-handling>	creation of entities, function calls	Figure B.73	Clause B.3.7.26
<port-declaration>	Declaration of a port	Figure B.46	Clause B.3.7.9
<raise-op>	**raise**	Figure B.88	Clause B.3.7.35
<raise-with-receiver-op>	**raise**	Figure B.89	Clause B.3.7.35.1
<raise-without-receiver-op>	**raise**	Figure B.90	Clause B.3.7.35.2
<read-timer-op>	**read** timer	Figure B.91	Clause B.3.7.36
<receive-assignment>	used in **receive** operation	Figure B.95	Clause B.3.7.37.3
<receive-op>	**receive**	Figure B.92	Clause B.3.7.37
<receive-with-sender>	used in **receive** operation	Figure B.93	Clause B.3.7.37.1
<receive-without-sender>	used in **receive** operation	Figure B.94	Clause B.3.7.37.2
<receiving-branch>	**alt** statement	Figure B.28	Clause B.3.7.1.1
<reply-op>	**reply**	Figure B.96	Clause B.3.7.38
<reply-with-receiver-op>	**reply**	FigureB.97	Clause B.3.7.38.1
<reply-without-receiver-op>	**reply**	Figure B.98	Clause B.3.7.38.2
<ref-par-var-calc>	creation of entities, function calls	Figure B.71	Clause B.3.7.24
<ref-par-timer-calc>	creation of entities, function calls	Figure B.72	Clause B.3.7.25
<return-stmt>	**return**	Figure B.99	Clause B.3.7.39
<return-with-value>	**return**	Figure B.100	Clause B.3.7.39.1
<return-without-value>	**return**	Figure B.101	Clause B.3.7.39.2
<running-all comp-op>	**all component.running**	Figure B.102	Clause B.3.7.40
<running-any comp-op>	**any component.running**	Figure B.103	Clause B.3.7.41
<running-component-op>	**running** component	Figure B.104	Clause B.3.7.42
<running-timer-op>	**running** timer	Figure B.105	Clause B.3.7.43
<self-op>	**self**	Figure B.109	Clause B.3.7.45
<send-op>	**send**	Figure B.106	Clause B.3.7.44
<send-with-receiver-op>	**send**	Figure B.107	Clause B.3.7.44.1
<send-without-receiver-op>	**send**	Figure B.108	Clause B.3.7.44.2
<start-component-op>	**start** component	Figure B.110	Clause B.3.7.46
<start-port-op>	**start** port	Figure B.111	Clause B.3.7.47
<start-timer-op>	**start** timer	Figure B.112	Clause B.3.7.48
<start-timer-op-default>	**start** timer	Figure B.113	Clause B.3.7.48.1
<start-timer-op-duration>	**start** timer	Figure B.114	Clause B.3.7.48.2
<stop-entity-op>	**stop** execution of module control, mtc or a test component	Figure B.116	Clause B.3.7.50
<stop-port-op>	**stop** port	Figure B.117	Clause B.3.7.51
<statement-block>	Statement block	Figure B.115	Clause B.3.7.49
<stop-timer-op>	**stop** timer	Figure B.118	Clause B.3.7.52

Identifier	Related TTCN-3 construct	Reference	
		Figure	Clause
<sut.action-op>	**sut.action-op**	Figure B.119	Clause B.3.7.53
<system-op>	**system**	Figure B.120	Clause B.3.7.54
<timeout-timer-op>	**timeout** timer	Figure B.121	Clause B.3.7.55
<timer-declaration>	Declaration of a timer	Figure B.47	Clause B.3.7.10
<timer-decl-default>	Declaration of a timer with a default duration	Figure B.48	Clause B.3.7.10.1
<timer-decl-no-def>	Declaration of a timer without default duration	Figure B.49	Clause B.3.7.10.2
<unmap-op>	**unmap** operation	FigureB.122	Clause B.3.7.56
<value-par-calculation>	creation of entities, function calls	Figure B.70	Clause B.3.7.23
<variable-declaration>	Declaration of a variable	Figure B.50	Clause B.3.7.11
<variable-declaration-init>	Declaration of a variable with an initial values	FigureB.51	Clause B.3.7.11.1
<variable-declaration-undef>	Declaration of a variable without an initial value	FigureB.52	Clause B.3.7.11.2
<var-value>	Used in <expression>	Figure B.63	Clause B.3.7.18.2
<verdit.get-op>	**verdict.get**	Figure B.123	Clause B.3.57
<verdit.set-op>	**verdict.set**	Figure B.124	Clause B.3.7.58
<while-stmt>	**while** statement	Figure B.125	Clause B.3.7.59

Annex C (normative):
Matching incoming values

C.1 Template matching mechanisms

This annex specifies the matching mechanisms that may be used in TTCN-3 templates (and only in templates).

C.1.1 Matching specific values

Specific values are the basic matching mechanism of TTCN-3 templates. Specific values in templates are expressions which do not contain any matching mechanisms or wildcards. Unless otherwise specified, a template field matches the corresponding incoming field value if, and only if, the incoming field value has exactly the same value as the value to which the expression in the template evaluates. For example:

```
// Given the message type definition
type record MyMessageType
{
    integer    field1,
    charstring field2,
    boolean    field3 optional,
    integer[4] field4
}

// A message template using specific values could be
template MyMessageType MyTemplate:=
{
    field1 := 3+2,        // specific value of integer type
    field2 := "My string", // specific value of charstring type
    field3 := true,       // specific value of boolean type
    field4 := {1,2,3}     // specific value of integer array
}
```

C.1.2 Matching mechanisms instead of values

C.1.2.1 Value list

Value lists specify lists of acceptable incoming values. It can be used on values of all types. A template field that uses a value list matches the corresponding incoming field if, and only if, the incoming field value matches any one of the values in the value list. Each value in the value list shall be of the type declared for the template field in which this mechanism is used. For example:

```
template Mymessage MyTemplate:=
{
    field1 := (2,4,6),              // list of integer values
    field2 := ("String1", "String2"),// list of charstring values
    :
    :
}
```

C.1.2.2 Complemented value list

The keyword **`complement`** denotes a list of values that will not be accepted as incoming values (i.e., it is the complement of a value list). It can be used on all values of all types.

Each value in the list shall be of the type declared for the template field in which the complement is used. A template field that uses complement matches the corresponding incoming field if and only if the incoming field does not match any of the values listed in the value list. The value list may be a single value, of course.

EXAMPLE:

```
template Mymessage MyTemplate:=
{
    complement (1,3,5),// list of unacceptable integer values
    :
    field3 not(true)      // will match false
    :
}
```

C.1.2.3 Omitting values

The keyword **omit** denotes that an optional template field shall be absent. It can be used on values of all types, provided that the template field is optional. For example:

```
template Mymessage:MyTemplate:=
{   :
    :
    field3 := omit,    // omit this field
    :
}
```

C.1.2.4 Any value

The matching symbol "?" (*AnyValue*) is used to indicate that any valid incoming value is acceptable. It can be used on values of all types. A template field that uses the any value mechanism matches the corresponding incoming field if, and only if, the incoming field evaluates to a single element of the specified type. For example:

```
template Mymessage:MyTemplate:=
{
    field1 := ?,   // will match any integer
    field2 := ?,   // will match any non-empty charstring value
    field3 := ?,   // will match true or false
    field4 := ?    // will match any sequence of integers
}
```

C.1.2.5 Any value or none

The matching symbol "*" (*AnyValueOrNone*) is used to indicate that any valid incoming value, including omission of that value, is acceptable. It can be used on values of all types, provided that the template field is declared as optional.

A template field that uses this symbol matches the corresponding incoming field if, and only if, either the incoming field evaluates to any element of the specified type, or if the incoming field is absent. For example:

```
template Mymessage:MyTemplate:=
{   :
    field3 := *,   // will match true or false or omitted field
    :
}
```

C.1.2.6 Value range

Ranges indicate a bounded range of acceptable values. It shall be used only on values of **integer** types (and integer sub-types). A boundary value shall be either:

a) infinity or -infinity;

b) an expression that evaluates to a specific integer value.

The lower boundary shall be put on the left side of the range, the upper boundary at the right side. The lower boundary shall be less than the upper boundary. A template field that uses a range matches the corresponding incoming field if, and only if, the incoming field value is equal to one of the values in the range. For example:

```
template Mymessage:MyTemplate:=
{
    field1 := (1 .. 6),// range of integer type
    :
    :
    :
}
// other entries for field1 might be (-infinity to 8) or (12 to infinity)
```

C.1.3 Matching mechanisms inside values

C.1.3.1 Any element

The matching symbol "?" (*AnyElement*) is used to indicate that it replaces single elements of a string (except character strings), a **record of**, a **set of** or an array. It shall be used only within values of string types, **record of** types, **set of** types and arrays. For example:

```
template Mymessage MyTemplate:=
{   :
    field2 := "abcxyz",
    field3 := '10???'B,   // where each "?" may either be 0 or 1
    field4 := {1, ?, 3}// where ? may be any integer value
}
```

NOTE: The "?" in `field4` can be interpreted as *AnyValue* as an integer value, or *AnyElement* inside a **record of**, **set of** or array. Since both interpretations lead to the same match no problem arises.

C.1.3.1.1 Using single character wildcards

If it is required to express the "?" wildcard in character strings it shall be done using character patterns (see clause C.1.5).. For example "abcdxyz", "abccxyz" "abcxxyz" etc. will all match **pattern** "abc?xyz". However, "abcxyz", abcdefxyz", etc. will not.

C.1.3.2 Any number of elements or no element

The matching symbol "*" (*AnyElementsOrNone*) is used to indicate that it replaces none or any number of consecutive elements of a string (except character strings), a **record of**, a **set of** or an array. It shall be used only within values of string types or arrays. The "*" symbol matches the longest sequence of elements possible, according to the pattern as specified by the symbols surrounding the "*". For example:

```
template Mymessage MyTemplate:=
{   :
    field2 := "abcxyz",
    field3 := '10*11'B,   //  where "*" may be any sequence of bits (possibly empty)
    field4 := {*, 2, 3}    // where the first element may be any integer value or omitted
}

var charstring MyStrings[4];
MyPCO.receive(MyStrings:{"abyz", *, "abc" });
```

If a "*" appears at the highest level inside a string, a **record of**, **set of** or array, it shall be interpreted as *AnyElementsOrNone*.

NOTE: This rule prevents the otherwise possible interpretation of "*" as *AnyValueOrNone* that replaces an element inside a string, **record of**, **set of** or array.

C.1.3.2.1 Using multiple character wildcards

If it is required to expressed the "*" wildcard in character strings it shall be done using character patterns (see clause C.1.5).For example: "abcxyz", "abcdefxyz" "abcabcxyz" etc. will all match **pattern** "abc*xyz"

C.1.4 Matching attributes of values

C.1.4.1 Length restrictions

The length restriction attribute is used to restrict the length of string values and the number of elements in a **set of** or **record of** structure. It shall be used only as an attribute of the following mechanisms: Complement, *AnyValue*, *AnyValueOrNone*, *AnyElement* and *AnyElementsOrNone*. It can also be used in conjunction with the **ifpresent** attribute. The syntax for **length** can be found in clause 6.2.3 and 6.3.3.

The units of length are to be interpreted according to table 4 in the main body of the present document in the case of string values. For **set of** and **record of** types the unit of length is the replicated type. The boundaries shall be denoted by expressions which resolve to specific non-negative **integer** values. Alternatively, the keyword **infinity** can be used as a value for the upper boundary in order to indicate that there is no upper limit of length.

The length specifications for the template shall not conflict with the length for restrictions (if any) of the corresponding type. A template field that uses Length as an attribute of a symbol matches the corresponding incoming field if, and only if, the incoming field matches both the symbol and its associated attribute. The length attribute matches if the length of the incoming field is greater than or equal to the specified lower bound and less than or equal to the upper bound. In the case of a single length value the length attribute matches only if the length of the received field is exactly the specified value.

In the case of an omitted field, the length attribute is always considered as matching (i.e., with **omit** it is redundant). With *AnyValueOrNone* and **ifpresent** it places a restriction on the incoming value, if any. For example:

```
template Mymessage MyTemplate:=
{
    field1 := complement (4,5) length (1 .. 6),// is the same as (1,2,3,6)
    field2 := "ab*ab" length(13) // max length of the AnyElementsOrNone string is 9 characters
    :
}
```

C.1.4.2 The IfPresent indicator

The **ifpresent** indicates that a match may be made if an optional field is present (i.e., not omitted). This attribute may be used with all the matching mechanisms, provided the type is declared as optional.

A template field that uses **ifpresent** matches the corresponding incoming field if, and only if, the incoming field matches according to the associated matching mechanism, or if the incoming field is absent. For example:

```
template Mymessage:MyTemplate:=
{   :
    field2 := "abcd" ifpresent, // matches "abcd" if not omitted
    :
    :
}
```

NOTE: *AnyValueOrNone* has exactly the same meaning as **? ifpresent**

C.1.5 Matching Character Pattern

Character patterns can be used in templates to define the format of a required character string to be received. Character patterns can be used to match **charstring** and **universal charstring** values. In addition to literal characters, character patterns allow the use of meta characters? and * to mean any character and any number of any character respectively. For example:

```
template charstring MyTemplate:= pattern "ab??xyz*";
```

This template would match any character string that consists of the characters 'ab', followed by any two characters, followed by the characters 'xyz', followed by any number of any characters.

If it is required to interpret any metacharacter literally it should be preceded with the metachacter '\'. For example:

```
template charstring MyTemplate:= pattern "ab?\?xyz*";
```

This template would match any character string which consists of the characters 'ab', followed by any characters, followed by the characters '?xyz', followed by any number of any characters.

In addition to direct string values it is also possible within the pattern statement to use references to existing templates, constants or variables. The reference shall resolve to one of the character string types and more than one. For example:

```
const charstring MyString:= "ab?";

template charstring MyTemplate:= pattern MyString;
```

This template would match any character string that consists of the characters 'ab', followed by any characters. In effect any character string following the **pattern** keyword either explicitly or by reference will be interpreted following the rules defined in this clause.

The pattern statement also allows the use of the concatenate operator and in the case of universal charstring the use of the Quadruple production to specify a single character. For example:

```
const charstring MyString:= "ab?";

template universal charstring MyTemplate:= pattern MyString & "de" & (1, 1, 13, 7);
```

This template would match any character string which consists of the characters 'ab', followed by any characters, followed by the characters 'de', followed by the character in ISO10646 with group=1, plane=1, row=65 and cell=7.

Annex D (normative):
Pre-defined TTCN-3 functions

D.1 Pre-defined TTCN-3 functions

This annex defines the TTCN-3 predefined functions.

D.1.1 Integer to character

```
int2char(integer value) return char
```

This function converts an **integer** value in the range of 0 … 127 (8-bit encoding) into a character value of ISO/IEC 646 [5]. The integer value describes the 8-bit encoding of the character.

The function returns –1 if the value of the argument is a negative or greater than 127.

D.1.2 Character to integer

```
char2int(char value) return integer
```

This function converts a **char** value of ISO/IEC 646 [5] into an integer value in the range of 0 … 127. The integer value describes the 8-bit encoding of the character.

D.1.3 Integer to universal character

```
int2unichar(integer value) return universal char
```

This function converts an **integer** value in the range of 0 … 268435455 (32-bit encoding) into a character value of ISO/IEC 10646 [6]. The integer value describes the 32-bit encoding of the character.

The function returns –1 if the value of the argument is a negative or greater than 268435455.

D.1.4 Univeral character to integer

```
unichar2int(univeral char value) return integer
```

This function converts a **univeral char** value of ISO/IEC 10646 [6] into an integer value in the range of 0 … 268435455. The integer value describes the 32-bit encoding of the character.

D.1.5 Bitstring to integer

```
bit2int(bitstring value) return integer
```

This function converts a single **bitstring** value to a single **integer** value.

For the purposes of this conversion, a **bitstring** shall be interpreted as a positive base 2 **integer** value. The rightmost bit is least significant, the leftmost bit is the most significant. The bits 0 and 1 represent the decimal values 0 and 1 respectively.

D.1.6 Hexstring to integer

```
hex2int(hexstring value) return integer
```

This function converts a single **hexstring** value to a single **integer** value.

For the purposes of this conversion, a **hexstring** shall be interpreted as a positive base 16 **integer** value. The rightmost hexadecimal digit is least significant, the leftmost hexadecimal digit is the most significant. The hexadecimal digits 0 .. F represent the decimal values 0 .. 15 respectively.

D.1.7 Octetstring to integer

```
oct2int(octetstring value) return integer
```

This function converts a single **octetstring** value to a single **integer** value.

For the purposes of this conversion, a **hexstring** shall be interpreted as a positive base 16 **integer** value. The rightmost hexadecimal digit is least significant, the leftmost hexadecimal digit is the most significant. The number of hexadecimal digits provided shall be multiples of 2 since one octet is composed of two hexadecimal digits. The hexadecimal digits 0 .. F represent the decimal values 0 .. 15 respectively.

D.1.8 Charstring to integer

```
str2int(charstring value) return integer
```

This function converts a **charstring** representing an **integer** value to the equivalent **integer**. If the string does not represent a valid integer value the function returns the value zero (0).

EXAMPLES:

str2int("66") will return the **integer** value 66

str2int("-66") will return the **integer** value -66

str2int("abc") will return the **integer** value 0

str2int("0") will return the **integer** value 0

D.1.9 Integer to bitstring

```
int2bit(integer value, length) return bitstring
```

This function converts a single **integer** value to a single **bitstring** value. The resulting string is length bits long.

For the purposes of this conversion, a **bitstring** shall be interpreted as a positive base 2 **integer** value. The rightmost bit is least significant, the leftmost bit is the most significant. The bits 0 and 1 represent the decimal values 0 and 1 respectively. If the conversion yields a value with fewer bits than specified in the length parameter, then the **bitstring** shall be padded on the left with zeros. A test case error shall occur if the value is negative or if the resulting **bitstring** contains more bits than specified in the length parameter.

D.1.10 Integer to hexstring

```
int2hex(integer value, length) return hexstring
```

This function converts a single **integer** value to a single **hexstring** value. The resulting string is length hexadecimal digits long.

For the purposes of this conversion, a **hexstring** shall be interpreted as a positive base 16 **integer** value. The rightmost hexadecimal digit is least significant, the leftmost hexadecimal digit is the most significant. The hexadecimal digits 0 ... F represent the decimal values 0 ... 15 respectively. If the conversion yields a value with fewer hexadecimal digits than specified in the length parameter, then the **hexstring** shall be padded on the left with zeros. A test case error shall occur if the value is negative or if the resulting **hexstring** contains more hexadecimal digits than specified in the length parameter.

D.1.11 Integer to octetstring

```
int2oct(integer value, length) return octetstring
```

This function converts a single **integer** value to a single **octetstring** value. The resulting string is `length` octets long.

For the purposes of this conversion, an **octetstring** shall be interpreted as a positive base 16 **integer** value. The rightmost hexadecimal digit is least significant, the leftmost hexadecimal digit is the most significant. The number of hexadecimal digits provided shall be multiples of 2 since one octet is composed of two hexadecimal digits. The hexadecimal digits 0 .. F represent the decimal values 0 .. 15 respectively. If the conversion yields a value with fewer hexadecimal digits than specified in the `length` parameter, then the **hexstring** shall be padded on the left with zeros. A test case error shall occur if the `value` is negative or if the resulting **hexstring** contains more hexadecimal digits than specified in the `length` parameter.

D.1.12 Integer to charstring

```
int2str(integer value) return charstring
```

This function converts the integer value into its string equivalent (the base of the return string is always decimal).

EXAMPLES:

int2str(66) will return the **charstring** value "66"

int2str(-66) will return the **charstring** value "-66"

int2str(0) will return the **integer** value "0"

D.1.13 Length of string type

```
lengthof(any_string_type value) return integer
```

This function returns the length of a value that is of type **bitstring**, **hexstring**, **octetstring**, or any character string. The units of length for each string type are defined in table 4 in the main body of the present document.

EXAMPLE:

```
lengthof('010'B) // returns 3

lengthof('F3'H) // returns 2

lengthof('F2'O) // returns 1

lengthof ("Length_of _Example") // returns 17
```

D.1.14 Number of elements in a structured type

```
sizeof(structured_type value) return integer
```

This function returns the actual number of elements of a **record**, **record of**, **set**, **set of**, **template** or array.

```
// Given
type record MyPDU
   {   boolean field1,
       integer field2
   }

// then
sizeof(MyPDU)
// returns 2
```

D.1.15 The IsPresent function

```
ispresent(any_type value) return boolean
```

This function returns the value **true** if and only if the value of the referenced field is present in the actual instance of the referenced data object. The argument to **ispresent** shall be a reference to a field within a data object that is defined as being **optional**.

```
// Given
type record MyRecord
    {   boolean field1 optional,
        integer field2
    }
// and given that MyPDU is a template of MyRecord type
// and received_PDU is also of MyRecord type
// then
MyPort.receive(MyPDU) -> value received_PDU
ispresent(received_PDU.field1)
// returns true if field1 in the actual instance of MyPDU is present
```

D.1.16 The IsChosen function

```
ischosen(any_type value) return boolean
```

This function returns the value **true** if and only if the data object reference specifies the variant of the **union** type that is actually selected for a given data object.

EXAMPLE:

```
// Given
type union MyUnion
    {   PDU_type1  p1,
        PDU_type2  p2,
        PDU_type   p3
    }

// and given that MyPDU is a template of MyUnion type
// and received_PDU is also of MyUnion type
// then
MyPort.receive(MyPDU) -> value received_PDU
ischosen(received_PDU.p2)
// returns true if the actual instance of MyPDU carries a PDU of the type PDU_type2
```

Annex E (normative):
Using other data types with TTCN-3

E.1 Using ASN.1 with TTCN-3

This annex defines the optional use of ASN.1 with TTCN-3.

TTCN-3 provides a clean interface for using ASN.1 version 1997 (as defined in the X.680 series [7], [8], [9], [10]) in TTCN-3 modules. When imported into a TTCN-3 module the language identifier for ASN.1 version 1997 shall be "ASN.1:1997".

When ASN.1 is used with TTCN-3 the keywords listed in table E.1 shall not be used as identifiers in a TTCN-3 module. ASN.1 keywords shall follow the requirements of X.680 [7].

Table E.1: List of ASN.1 keywords

ABSENT	EMBEDDED	INTERSECTION	SEQUENCE
ABSTRACT-SYNTAX	END	Iso10646string	SET
ALL	ENUMERATED	MAX	SIZE
APPLICATION	EXCEPT	MIN	STRING
AUTOMATIC	EXPLICIT	MINUS-INFINITY	SYNTAX
BEGIN	EXPORTS	NULL	T61String
BIT	EXTERNAL	NumericString	TAGS
BMPSTRING	FALSE	OBJECT	TeletexString
BOOLEAN	FROM	ObjectDescriptor	TRUE
BY	GeneralizedTime	OCTET	TYPE-IDENTIFIER
CHARACTER	GeneralString	OF	UNION
CHOICE	IA5String	OPTIONAL	UNIQUE
CLASS	IDENTIFIER	PDV	UNIVERSAL
COMPONENT	IMPLICIT	PLUS-INFINITY	UniversalString
COMPONENTS	IMPORTS	PRESENT	UTCTime
CONSTRAINED	INCLUDES	PrintableString	VideotexString
DEFAULT	INSTANCE	PRIVATE	VisibleString
DEFINITIONS	INTEGER	REAL	WITH

E.1.1 ASN.1 and TTCN-3 type equivalents

The ASN.1 types listed in table E.2 are considered to be equivalent to their TTCN-3 counterparts.

Table E.2: List of ASN.1 and TTCN-3 equivalents

ASN.1 type	Maps to TTCN-3 equivalent
BOOLEAN	**boolean**
INTEGER	**integer**
REAL	**float**
OBJECT IDENTIFIER	**objid**
BIT STRING	**bitstring**
OCTET STRING	**octetstring**
SEQUENCE	**record**
SEQUENCE OF	**record of**
SET	**set**
SET OF	**set of**
ENUMERATED	**enumerated**
CHOICE	**union**

All TTCN-3 operators, functions, matching mechanisms, value notation etc. that can be used with a TTCN-3 type given in table E.E.2 may also be used with the corresponding ASN.1 type.

E.1.2 ASN.1 data types and values

ASN.1 types and values may be used in TTCN-3 modules. ASN.1 definitions are made using a separate ASN.1 module.

EXAMPLE:

```
MyASN1module DEFINITIONS ::=
BEGIN
     Z::=   INTEGER           -- Simple type definition


     BMessage::= SET          -- ASN.1 type definition
     {
         name   Name,
         title  VisibleString,
         date   Date
     }

     johnValues Bmessage ::=  -- ASN.1 value definition
     {
         name   "John Doe",
         title  "Mr",
         date   "April 12th"
     }
END
```

The ASN.1 module shall be written according to the syntax of the ITU-T Recommendation X.680 series [7], [8], [9] and [10]. Once declared, ASN.1 types and values may be used within TTCN-3 modules in exactly the same way that ordinary TTCN-3 types and values from other TTCN-3 modules are used (i.e. the required definitions shall be imported).

EXAMPLE:

```
module MyTTCNModule
{
     import all from MyASN1module language "ASN.1:1997";

     const Bmessage MyTTCNConst:= johnValues;
}
```

NOTE: ASN.1 definitions other than types and values (i.e. information object classes or information object sets) are not directly accessible from the TTCN-3 notation. Such definitions shall be resolved to a type or value within the ASN.1 module before they can be referenced from within the TTCN-3 module.

E.1.2.1 Scope of ASN.1 identifiers

Imported ASN.1 identifiers follow the same scope rules as imported TTCN-3 types and values (see clause 5.4).

E.1.3 Parameterization in ASN.1

It is permitted to reference parameterized ASN.1 type and value definitions from with the TTCN-3 module. However, all ASN.1 parameterized definitions used in a TTCN-3 module shall be provided with actual parameters (open types or values are not permitted) and the actual parameters provided shall be resolvable at compile-time.

The TTCN-3 core language does not support parameterization of uniquely ASN.1 specific objects. ASN.1 specific parameterization which involves objects which cannot be defined directly in the TTCN-3 core language shall therefore be resolved in the ASN.1 part before use within the TTCN-3. The ASN.1 specific objects are:

a) Value sets;

b) Information Object classes;

c) Information Objects;

d) Information Object Sets.

For example the following is not legal because it defines a TTCN-3 type which takes an ASN.1 object set as an actual parameter.

```
MyASN1module DEFINITIONS ::=
BEGIN
    -- ASN.1 Module definition

    -- Information object class definition
    MESSAGE ::= CLASS { &msgTypeValueINTEGER UNIQUE,
                                    &MsgFields}

    -- Information object definition
    setupMessage MESSAGE ::= {&msgTypeValue     1,
                              &MsgFields        OCTET STRING}

    setupAckMessage MESSAGE ::= {&msgTypeValue  2,
                                 &MsgFields     BOOLEAN}

    -- Information object set definition
    MyProtocol MESSAGE ::= { setupMessage | setupAckMessage}

    -- ASN.1 type constrained by object set
    MyMessage{ MESSAGE : MsgSet} ::= SEQUENCE
    {
        code   MESSAGE.&msgTypeValue({ MsgSet}),
        Type  MESSAGE.&MsgFields({ MsgSet})
}
END

module MyTTCNModule
{
    // TTCN-3 module definition
    import all from MyASN1module language "ASN.1:1997";

    // Illegal TTCN-3 type with object set as parameter
    type record Q(MESSAGE MyMsgSet) ::= {Z                 field1,
                                         MyMessage(MyMsgSet)field2}
}
```

To make this a legal definition the extra ASN.1 type My Message1 has to be defined as shown below. This resolves the information object set parameterization and can therefore be directly used in the TTCN-3 module.

```
MyASN1module DEFINITIONS ::=
BEGIN
    -- ASN.1 Module definition

    …

    MyProtocol MESSAGE ::= { setupMessage | setupAckMessage}

    -- Extra ASN.1 type to remove object set parametrization
    MyMessage1 ::= MyMessage{ MyProtocol}
END

module MyTTCNModule
{
    // TTCN-3 module definition
    import all from MyASN1module language "ASN.1:1997";

    // Legal TTCN-3 type with no object set as parameter
    type record Q := {Z             field1,
                      MyMessage1    field2}
}
```

E.1.4 Defining message types with ASN.1

In ASN.1 messages are defined using SEQUENCE (or possibly SET).

EXAMPLE:

```
MyASN1module DEFINITIONS ::=
BEGIN
    -- ASN.1 Module definition

    MyMessageType ::= SEQUENCE
    {
        field1     Field1Type,
        field2     Field2Type OPTIONAL, -- This field may be omitted
        :
        fieldN     FieldNType
    }
END
```

Messages defined using ASN.1 may also, of course, be sub-structured using SEQUENCE, SET etc.

E.1.5 Defining ASN.1 message templates

If messages are defined in ASN.1 using, for example: SEQUENCE (or possibly SET) then actual messages, for both **send** and **receive** events, can be specified using the ASN.1 value syntax.

EXAMPLE:

```
MyASN1module DEFINITIONS ::=
BEGIN
    -- ASN.1 Module definition

    -- The message definition
    MyMessageType::= SEQUENCE
    {   field1 [1] IA5STRING,        // Like TTCN-3 character string
        field2 [2] INTEGER OPTIONAL, // like TTCN-3 integer
        field3 [4] Field3Type,       // Like TTCN-3 record
        field4 [5] Field4Type        // Like TTCN-3 array
    }

    Field3Type::= SEQUENCE {field31 BIT STRING, field32 INTEGER, field33 OCTET STRING},
    Field4Type::= SEQUENCE OF BOOLEAN


    -- may have the following value
    myValue MyMessageType::=
    {
        field1     "A string",
        field2     123,
        field3     {field31 '11011'B, field32 456789, field33 'FF'O},
        field4     {true, false}
    }
END
```

E.1.5.1 ASN.1 receive messages using the TTCN-3 template syntax

Matching mechanisms are not supported in the standard ASN.1 syntax. Thus, if it is wished to use matching mechanisms with an ASN.1 receive message then the TTCN-3 syntax for receive templates shall be used instead. Note that this syntax includes component references in order to be able to reference the individual components in ASN.1 SEQUENCE, SET etc.

EXAMPLE:

```
import type myMessageType from MyASN1module language "ASN.1:1997";

// a message template using matching mechanisms within TTCN-3 might be
template myMessageTypeMyValue:=
{
    field1 :=          "A"<?>"tr"<*>"g",
    field2 :=          *,
    field3.field31 := '110??'B,
    field3.field32 := ?,
    field3.field33 := 'F?'O,
    field4.[0] :=      true,
    field4.[1] :=      false
}

// the following syntax is equally valid
template myMessageTypeMyValue:=
{
    field1 := "A"<?>"tr"<*>"g",      // string with wildcards
    field2 := *,                     // any integer or none at all
    field3 := {'110??'B, ?, 'F?'O},
    field4 := {?, false}
}
```

E.1.5.2 Ordering of template fields

When TTCN-3 templates are used for ASN.1 types the significance of the order of the fields in the template will depend on the type of ASN.1 construct used to define the message type. For example: if SEQUENCE or SEQUENCE OF is used then the message fields shall be sent or matched in the order specified in the template. If SET or SET OF is used then the message fields may be sent or matched in any order.

E.1.6 Encoding information

TTCN-3 allows references to encoding rules and variations within encoding rules to be associated with various TTCN-3 language elements. It is also possible to define invalid encodings. This encoding information is specified using the **with** statement according to the following syntax:

EXAMPLE:

```
module MyModule
{   :
    import type myMessageType from MyASN1module language "ASN.1:1997"with {encode:=
"PER:1997"}
                        // All instances of MyMessageType should be encoded using PER:1997

} with {encode "BER:1997"} // Default encoding for the entire module (test suite) is BER:1997
```

E.1.6.1 ASN.1 encoding attributes

The following strings are the predefined (standardized) encoding attributes for ASN.1:

a) "BER:1997" means encoded according to ITU-T Recommendation X.690 (BER) [11];

b) "CER:1997" means encoded according to ITU-T Recommendation X.690 (CER) [11];

c) "DER:1997" means encoded according to ITU-T Recommendation X.690 (DER) [11].

d) "PER-BASIC-UNALIGNED:1997" means encoded according to (Unaligned PER) ITU-T Recommendation X.691 [12];

e) "PER-BASICALIGNED:1997" means encoded according to ITU-T Recommendation X.691 (Aligned PER) [12];

f) "PER-CANONICAL-UNALIGNED:1997" means encoded according to (Canonical Unaligned PER) ITU-T Recommendation X.691 [12];

g) "PER-CANONICAL-ALIGNED:1997" means encoded according to ITU-T Recommendation X.691 (Canonical Aligned PER) [12].

_____