International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.140

(03/2006)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

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

Testing and Test Control Notation version 3 (TTCN-3): Core language

ITU-T Recommendation Z.140

ITU-T

ITU-T Z-SERIES RECOMMENDATIONS

LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

FORMAL DESCRIPTION TECHNIQUES (FDT)	
Specification and Description Language (SDL)	Z.100–Z.109
Application of formal description techniques	Z.110–Z.119
Message Sequence Chart (MSC)	Z.120–Z.129
Extended Object Definition Language (eODL)	Z.130–Z.139
Testing and Test Control Notation (TTCN)	**Z.140–Z.149**
User Requirements Notation (URN)	Z.150–Z.159
PROGRAMMING LANGUAGES	
CHILL: The ITU-T high level language	Z.200–Z.209
MAN-MACHINE LANGUAGE	
General principles	Z.300–Z.309
Basic syntax and dialogue procedures	Z.310–Z.319
Extended MML for visual display terminals	Z.320–Z.329
Specification of the man-machine interface	Z.330–Z.349
Data-oriented human-machine interfaces	Z.350–Z.359
Human-machine interfaces for the management of telecommunications networks	Z.360–Z.379
QUALITY	
Quality of telecommunication software	Z.400–Z.409
Quality aspects of protocol-related Recommendations	Z.450–Z.459
METHODS	
Methods for validation and testing	Z.500–Z.519
MIDDLEWARE	
Distributed processing environment	Z.600–Z.609

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

ITU-T Recommendation Z.140

Testing and Test Control Notation version 3 (TTCN-3): Core language

Summary

This Recommendation defines TTCN-3 (*Testing and Test Control Notation 3*) intended for specification of test suites that are independent of platforms, test methods, protocol layers and protocols. TTCN-3 can be used for 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 and APIs. The specification of test suites for physical layer protocols is outside the scope of this Recommendation.

The core language of TTCN-3 can be expressed in a variety of presentation formats. While this Recommendation defines the Core Language, ITU-T Rec. Z.141 defines the Tabular Format for TTCN (TFT) and ITU-T Rec. Z.142 defines the Graphical Format for TTCN (GFT). The specification of these formats is outside the scope of this Recommendation. The core language serves three purposes:

1) as a generalized text-based test language;
2) as a standardized interchange format of TTCN test suites between TTCN tools;
3) as the semantic basis (and where relevant, the syntactical basis) for the 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.

Source

ITU-T Recommendation Z.140 was approved on 16 March 2006 by ITU-T Study Group 17 (2005-2008) under the ITU-T Recommendation A.8 procedure.

FOREWORD

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

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

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

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

NOTE

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

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

INTELLECTUAL PROPERTY RIGHTS

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

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

© ITU 2006

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

CONTENTS

		Page
1	Scope	1
2	References	1
3	Definitions and abbreviations	2
	3.1 Definitions	2
	3.2 Abbreviations	3
4	Introduction	4
	4.0 General	4
	4.1 The core language and presentation formats	4
	4.2 Unanimity of the specification	5
	4.3 Conformance	5
5	Basic language elements	5
	5.0 General	5
	5.1 Ordering of language elements	6
	5.2 Parameterization	7
	5.3 Scope rules	9
	5.4 Identifiers and keywords	11
6	Types and values	12
	6.0 General	12
	6.1 Basic types and values	12
	6.2 Sub-typing of basic types	14
	6.3 Structured types and values	17
	6.4 The anytype	23
	6.5 Arrays	24
	6.6 Recursive types	25
	6.7 Type compatibility	25
7	Modules	29
	7.0 General	29
	7.1 Naming of modules	29
	7.2 Module parameters	30
	7.3 Module definitions part	30
	7.4 Module control part	32
	7.5 Importing from modules	32
8	Test configurations	38
	8.0 General	38
	8.1 Port communication model	39
	8.2 Restrictions on connections	40
	8.3 Abstract test system interface	41
	8.4 Defining communication port types	42
	8.5 Defining component types	43
	8.6 Addressing entities inside the SUT	45
	8.7 Component references	46
	8.8 Defining the test system interface	47
9	Declaring constants	47
10	Declaring variables	47
	10.0 General	47
	10.1 Value variables	48
	10.2 Template variables	48
11	Declaring timers	48
	11.0 General	48
	11.1 Timers as parameters	49
12	Declaring messages	49

Page

		Page
13	Declaring procedure signatures	49
	13.0 General	49
	13.1 Signatures for blocking and non-blocking communication	49
	13.2 Parameters of procedure signatures	50
	13.3 Value returning remote procedures	50
	13.4 Specifying exceptions	50
14	Declaring templates	50
	14.0 General	50
	14.1 Declaring message templates	51
	14.2 Declaring signature templates	52
	14.3 Template matching mechanisms	53
	14.4 Parameterization of templates	57
	14.5 Void	57
	14.6 Modified templates	57
	14.7 Changing template fields	59
	14.8 Match Operation	59
	14.9 Valueof Operation	59
15	Operators	59
	15.0 General	59
	15.1 Arithmetic operators	61
	15.2 String operators	61
	15.3 Relational operators	61
	15.4 Logical operators	63
	15.5 Bitwise operators	63
	15.6 Shift operators	64
	15.7 Rotate operators	64
16	Functions and altsteps	65
	16.1 Functions	65
	16.2 Altsteps	68
	16.3 Functions and altsteps for different component types	70
17	Test cases	70
	17.0 General	70
	17.1 Parameterization of test cases	71
18	Overview of program statements and operations	71
19	Expressions and basic program statements	74
	19.0 General	74
	19.1 Expressions	74
	19.2 Assignments	74
	19.3 The Log statement	74
	19.4 The Label statement	76
	19.5 The Goto statement	76
	19.6 The If-else statement	77
	19.7 The For statement	78
	19.8 The While statement	78
	19.9 The Do-while statement	78
	19.10 The Stop execution statement	78
	19.11 The Select Case statement	79
20	Behavioural program statements	80
	20.0 General	80
	20.1 Alternative behaviour	80
	20.2 The Repeat statement	84
	20.3 Interleaved behaviour	84
	20.4 The Return statement	85

Page

		Page
21	Default handling	86
21.0	General	86
21.1	The default mechanism	86
21.2	Default references	87
21.3	The Activate operation	87
21.4	The Deactivate operation	88
22	Configuration operations	88
22.0	General	88
22.1	The Create operation	89
22.2	The Connect and Map operations	90
22.3	The Disconnect and Unmap operations	91
22.4	The MTC, System and Self operations	92
22.5	The Start test component operation	92
22.6	The Stop test behaviour operation	93
22.7	The Running operation	94
22.8	The Done operation	94
22.9	The Kill test component operation	95
22.10	The Alive operation	95
22.11	The Killed operation	96
22.12	Using component arrays	96
22.13	Summary of the use of any and all with components	97
23	Communication operations	97
23.0	General	97
23.1	General format of communication operations	98
23.2	Message-based communication	100
23.3	Procedure-based communication	103
23.4	The Check operation	112
23.5	Controlling communication ports	113
23.6	Use of any and all with ports	114
24	Timer operations	114
24.0	General	114
24.1	The Start timer operation	115
24.2	The Stop timer operation	115
24.3	The Read timer operation	116
24.4	The Running timer operation	116
24.5	The Timeout operation	116
24.6	Summary of use of any and all with timers	116
25	Test verdict operations	117
25.0	General	117
25.1	Test case verdict	117
25.2	Verdict values and overwriting rules	117
26	External actions	118
27	Module control part	118
27.0	General	118
27.1	Execution of test cases	119
27.2	Termination of test cases	119
27.3	Controlling execution of test cases	119
27.4	Selection of test cases	119
27.5	Use of timers in control	120
28	Specifying attributes	121
28.0	General	121
28.1	Display attributes	121
28.2	Encoding of values	121

Page

28.3 Extension attributes ... 123
28.4 Scope of attributes ... 123
28.5 Overwriting rules for attributes ... 124
28.6 Changing attributes of imported language elements ... 125

Annex A – BNF and static semantics ... 126
A.1 TTCN-3 BNF ... 126

Annex B – Matching incoming values ... 144
B.1 Template matching mechanisms ... 144

Annex C – Pre-defined TTCN-3 functions ... 153
C.0 General exception handling procedures ... 153
C.1 Integer to character ... 153
C.2 Character to integer ... 153
C.3 Integer to universal character ... 153
C.4 Universal character to integer ... 153
C.5 Bitstring to integer ... 153
C.6 Hexstring to integer ... 153
C.7 Octetstring to integer ... 154
C.8 Charstring to integer ... 154
C.9 Integer to bitstring ... 154
C.10 Integer to hexstring ... 154
C.11 Integer to octetstring ... 154
C.12 Integer to charstring ... 155
C.13 Length of string type ... 155
C.14 Number of elements in a structured value ... 155
C.15 The IsPresent function ... 156
C.16 The IsChosen function ... 156
C.17 The Regexp function ... 156
C.18 Bitstring to charstring ... 156
C.19 Hexstring to charstring ... 157
C.20 Octetstring to character string ... 157
C.21 Character string to octetstring ... 157
C.22 Bitstring to hexstring ... 157
C.23 Hexstring to octetstring ... 158
C.24 Bitstring to octetstring ... 158
C.25 Hexstring to bitstring ... 158
C.26 Octetstring to hexstring ... 158
C.27 Octetstring to bitstring ... 159
C.28 Integer to float ... 159
C.29 Float to integer ... 159
C.30 The random number generator function ... 159
C.31 The Substring function ... 159
C.32 Number of elements in a structured type ... 160
C.33 Character string to float ... 160
C.34 The Replace function ... 160
C.35 Octetstring to character string ... 161
C.36 Character string to octetstring ... 161

Annex D (informative) – Void ... 162

Annex E (informative) – Library of useful types ... 163
E.1 Limitations ... 163
E.2 Useful TTCN-3 types ... 163

Annex F (informative) Operations on TTCN-3 active objects ... 166
F.1 General ... 166
F.2 Test components ... 167
F.3 Timers ... 170

Page

F.4 Ports 171

Annex G (informative) – Deprecated language features 174

G.1 Group style definition of module parameters 174

G.2 Recursive import 174

G.3 Using `all` in port type definitions 174

BIBLIOGRAPHY 175

ITU-T Recommendation Z.140

Testing and Test Control Notation version 3 (TTCN-3): Core language

1 Scope

This Recommendation defines the Core Language of 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 this Recommendation.

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 (ITU-T Rec. Z.141 [1]) and a graphical presentation format (ITU-T Rec. Z.142 [2]). The specification of these formats is outside the scope of this Recommendation.

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 this Recommendation.

2 References

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

[1] ITU-T Recommendation Z.141 (2006), *Testing and Test Control Notation version 3 (TTCN-3): Tabular presentation format (TFT)*.

[2] ITU-T Recommendation Z.142 (2006), *Testing and Test Control Notation version 3 (TTCN-3): Graphical presentation format (GFT)*.

[3] ITU-T Recommendation Z.143 (2006), *Testing and Test Control Notation version 3 (TTCN-3): Operational semantics*.

[4] ITU-T Recommendation Z.144 (2006), *Testing and Test Control Notation version 3 (TTCN-3): Runtime interface (TRI)*.

[5] ITU-T Recommendation Z.145 (2006), *Testing and Test Control Notation version 3 (TTCN-3): Control interface (TCI)*.

[6] ITU-T Recommendation Z.146 (2006), *Testing and Test Control Notation version 3 (TTCN-3): Using ASN.1 with TTCN-3*.

[7] ITU-T Recommendation X.290 (1995), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – General concepts*.

ISO/IEC 9646-1:1994, *Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 1: General concepts*.

[8] ITU-T Recommendation X.292 (2002), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – The Tree and Tabular Combined Notation (TTCN)*.

ISO/IEC 9646-3:1998, *Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 3: The Tree and Tabular Combined Notation (TTCN)*.

[9] ITU-T Recommendation T.50 (1992), *International Reference Alphabet (IRA) (Formerly International Alphabet No. 5 or IA5) – Information technology – 7-bit coded character set for information interchange*.

ISO/IEC 646:1991, *Information technology – ISO 7-bit coded character set for information interchange*.

[10] ISO/IEC 10646:2003, *Information technology – Universal Multiple-Octet Coded Character Set (UCS)*.

[11] ISO/IEC 6429:1992, *Information technology – Control functions for coded character sets*.

3 Definitions and abbreviations

3.1 Definitions

For the purposes of this Recommendation, the terms and definitions given in ITU-T Rec. X.290 [7], ITU-T Rec. X.292 [8] and the following apply:

3.1.1 actual parameter: Value, expression, template or name reference (identifier) to be passed as a parameter to the invoked entity (function, test case, altstep, etc.) as defined at the place of invoking.

NOTE – The number, order and type of all actual parameters to be passed at a single invocation shall be in line with the list of formal parameters as defined in the invoked entity.

3.1.2 basic types: Set of predefined TTCN-3 types described in 6.1.0 and 6.1.1.

NOTE – Basic types are referenced by their names.

3.1.3 compatible type: TTCN-3 is not strongly typed but the language does require type compatibility.

NOTE – Variables, constants, templates, etc., have compatible types if conditions in 6.7 are met.

3.1.4 communication port: Abstract mechanism facilitating communication between test components.

NOTE – 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.

3.1.5 data types: Common name for simple basic types, basic string types, structured types, the special data type **`anytype`** and all user-defined types based on them (see Table 3).

3.1.6 defined types (defined TTCN-3 types): Set of all predefined TTCN-3 types (basic types, all structured types, the type **`anytype`**, the address, port and component types and the default type) and all user-defined types declared either in the module or imported from other TTCN-3 modules.

3.1.7 dynamic parameterization: Kind of parameterization, in which actual parameters are dependent on run-time events; e.g., the value of the actual parameter is a value received during run-time or depends on a received value by a logical relation.

3.1.8 exception: In cases of procedure-based communication, an exception (if defined) is raised by an answering entity if it cannot answer a remote procedure call with the normal expected response.

3.1.9 formal parameter: Typed name or typed template reference (identifier) not resolved at the time of the definition of an entity (function, test case, altstep, etc.) but at the time of invoking it.

NOTE – Actual values or templates (or their names) to be used at the place of formal parameters are passed from the place of invoking the entity (see also the definition of actual parameter).

3.1.10 global visibility: Attribute of an entity (module parameter, constant, template, etc.) that its identifier can be referenced anywhere within the module where it is defined including all functions, test cases and altsteps defined within the same module and the control part of that module.

3.1.11 Implementation Conformance Statement (ICS): See ITU-T Rec. X.290 [7].

3.1.12 Implementation eXtra Information for Testing (IXIT): See ITU-T Rec. X.290 [7].

3.1.13 Implementation Under Test (IUT): See ITU-T Rec. X.290 [7].

3.1.14 known types: Set of all TTCN-3 predefined types, types defined in a TTCN-3 module and types imported into that module from other TTCN-3 modules or from non-TTCN-3 modules.

3.1.15 left-hand side (of assignment): Value or template variable identifier or a field name of a structured type value or template variable (including array index if any), which stands to the left of an assignment symbol (:=).

NOTE – A constant, module parameter, timer, structured type field name or a template header (including template type, name and formal parameter list) standing to the left of an assignment symbol (:=) in declarations and/or a modified template definitions are out of the scope of this definition as not being part of an assignment.

3.1.16 local visibility: Attribute of an entity (constant, variable, etc.) whereby its identifier can be referenced only within the function, test case or **`altstep`** where it is defined.

3.1.17 Main Test Component (MTC): See ITU-T Rec. X.292 [8].

3.1.18 passing parameter by value: Way of passing parameters where the arguments are evaluated before a parameterizable entity is entered.

NOTE – Only the values of the arguments are passed and changes to the arguments within the called entity have no effect on the actual arguments as seen by the caller.

3.1.19 passing parameter by reference: Way of passing parameters where arguments are not evaluated before the function, altstep, etc., is entered and a reference to the parameter is passed by the calling procedure (function, altstep, etc.) to the called procedure.

NOTE – All changes to the arguments within the called procedure have an effect on the actual arguments as seen by the caller.

3.1.20 Parallel Test Component (PTC): See ITU-T Rec. X.292 [8].

3.1.21 right-hand side (of assignment): Expression, template reference or signature parameter identifier which stands to the right of an assignment symbol (:=).

NOTE – Expressions and template references standing to the right of an assignment symbol (:=) in constant, module parameter, timer, template or modified template declarations are out of the scope of this definition as not being part of an assignment.

3.1.22 root type: Basic type, structured type, special data type, special configuration type or special default type to which the user-defined TTCN-3 type can be traced back.

3.1.23 static parameterization: Kind of parameterization, in which actual parameters are independent of run-time events; i.e., known at compile time or, in the case of module parameters, known by the start of the test suite execution (e.g., known from the test suite specification, here counting imported definitions, or the test system is aware of its value before execution time).

NOTE – All types are known at compile time, i.e., are statically bound.

3.1.24 strong typing: Strict enforcement of type compatibility by type name equivalence with no exceptions.

3.1.25 System Under Test (SUT): See ITU-T Rec. X.290 [7].

3.1.26 template: TTCN-3 templates are specific data structures for testing; used to either transmit a set of distinct values or to check whether a set of received values matches the template specification.

3.1.27 test behaviour: (or behaviour) Test case or a function started on a test component when executing an **`execute`** or a **`start`** component statement and all functions and altsteps called recursively.

NOTE – During a test case execution each test component has its own behaviour and hence several test behaviours may run concurrently in the test system (i.e., a test case can be seen as a collection of test behaviours).

3.1.28 test case: See ITU-T Rec. X.290 [7].

3.1.29 test case error: See ITU-T Rec. X.290 [7].

3.1.30 test suite: Set of TTCN-3 modules that contains a completely defined set of test cases, optionally supplemented with one or more TTCN-3 control parts.

3.1.31 test system: See ITU-T Rec. X.290 [7].

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

3.1.33 type compatibility: Language feature that allows use of values, expressions or templates of a given type as actual values of another type (e.g., at assignments, as actual parameters at calling a function, referencing a template, etc., or as a return value of a function).

NOTE – Both the type and the current value of the value, expression or template shall be compatible with the other type.

3.1.34 value parameterization: Ability to pass a value or template as an actual parameter into a parameterized object.

NOTE – This actual value parameter then completes the specification of that object.

3.1.35 user-defined type: Type that is defined by subtyping of a basic type or declaring a structured type.

NOTE – User-defined types are referenced by their identifiers (names).

3.1.36 value notation: Notation by which an identifier is associated with a given value or range of a particular type.

NOTE – Values may be constants or variables.

3.2 Abbreviations

This Recommendation uses the following abbreviations:

API	Application Programming Interface
ATS	Abstract Test Suite
BMP	Basic Multilingual Plane
BNF	Backus-Naur Form

CORBA	Common Object Request Broker Architecture
ETS	Executable Test Suite
FIFO	First In First Out
ICS	Implementation Conformance Statement
IRV	International Reference Version
IUT	Implementation Under Test
IXIT	Implementation eXtra Information for Testing
MTC	Main Test Component
PTC	Parallel Test Component
SUT	System Under Test
TSI	Test System Interface

4 Introduction

4.0 General

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.

TTCN-3 includes the following essential characteristics:

- the ability to specify dynamic concurrent testing configurations;
- operations for procedure-based and message-based 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;
- value parameterization;
- the assignment and handling of test verdicts;
- test suite parameterization and test case selection mechanisms;
- combined use of TTCN-3 with other languages;
- 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

The TTCN-3 specification is separated into several parts. The first part, defined in this Recommendation, is the core language. The second part, defined in ITU-T Rec. Z.141 [1], is the tabular presentation format. The third part, defined in ITU-T Rec. Z.142 [2], is the graphical presentation format. The fourth part, ITU-T Rec. Z.143 [3] contains the operational semantics of the language. The fifth part, ITU-T Rec. Z.144 [4], defines the TTCN-3 Runtime Interface (TRI), the sixth part, ITU-T Rec. Z.145 [5], defines the TTCN-3 Control Interfaces (TCI) and the seventh part, ITU-T Rec. Z.146 [6], specifies the use of ASN.1 definitions with TTCN-3.

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-3 test suites between TTCN-3 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 this Recommendation.

TTCN-3 may optionally be used with other type-value notations in which case definitions in other languages may be used as an alternative data type and value syntax. Other parts of this standard specify use of some other languages with TTCN-3. The support of other languages is not limited to those specified in the ITU-T Z.140 series of Recommendations but to support languages for which combined use with TTCN-3 is defined, rules given in this Recommendation shall apply.

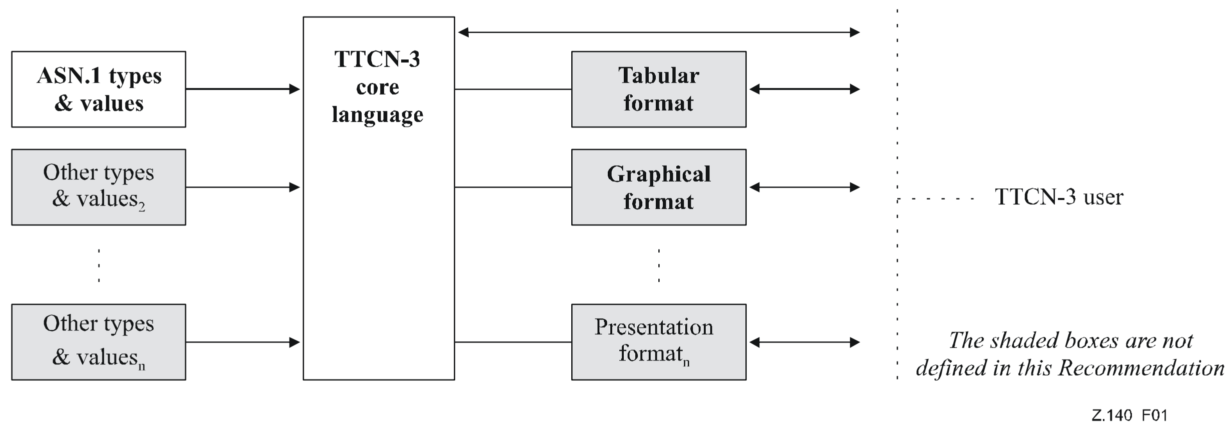


Figure 1/Z.140 – User's view of the core language and the various presentation formats

The core language is defined by a complete syntax (see Annex A) and operational semantics (ITU-T Rec. Z.143 [3]). It contains minimal static semantics (provided in the body of this Recommendation and in Annex A) which do not restrict the use of the language due to some underlying application domain or methodology.

4.2 Unanimity of the specification

The language is specified syntactically and semantically in terms of a textual description in the body of this Recommendation (clauses 5 to 28) and in a formalized way in Annex A. In each case, when the textual description is not exhaustive, the formal description completes it. If the textual and the formal specifications are contradictory, the latter shall take precedence.

4.3 Conformance

For an implementation claiming to conform to this version of the language, all features specified in this Recommendation shall be implemented consistently with the requirements given in this Recommendation and in ITU-T-Rec. Z.143 [3].

5 Basic language elements

5.0 General

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 module parameters to allow test suite parameterization.

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.

A special kind of data structure 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 message-based and procedure-based 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, 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/Z.140 – Overview of TTCN-3 language elements

Language element	Associated keyword	Specified in module definitions	Specified in module control	Specified in functions/ altsteps/test cases	Specified in test component type
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	Yes
Data/signature template definitions	**template**	Yes	Yes	Yes	Yes
Function definitions	**function**	Yes			
Altstep definitions	**altstep**	Yes			
Test case definitions	**testcase**	Yes			
Value variable declarations	**var**		Yes	Yes	Yes
Template variable declarations	**var template**		Yes	Yes	Yes
Timer declarations	**timer**		Yes	Yes	Yes
NOTE – The notions "definition" and "declaration" of variables, constants, types and other language elements are used interchangeably throughout this Recommendation. The distinction between both notions is useful only for implementation purposes, as it is the case in programming languages like C and C++. On the level of TTCN-3, the notions have equal meaning.					

5.1 Ordering of language elements

Generally, the order in which declarations can be made is arbitrary. Inside a block of statements and declarations, such as a function body or a branch of an **if-else** statement, all declarations (if any), shall be made at the beginning of the 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;
  :
}

 :
```

Declarations in the module definitions part may be made in any order. However, inside the module control part, test case definitions, functions, and altsteps, all required declarations must be given beforehand. This means in particular, local variables, local timers, and local constants shall never be used before they are declared. The only exceptions to this rule are labels. Forward references to a label may be used in **goto** statements before the label occurs (see 19.5).

5.2 Parameterization

5.2.0 Static and dynamic parameterization

TTCN-3 supports *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* value parameterization i.e., this parameterization shall be resolved at compile-time;

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

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/Z.140 – Overview of parameterizable TTCN-3 language elements

Keyword	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 (Note 1)	Static at compile-time	*Values of:* all basic types, all user-defined types and **address** type.
template	Dynamic at run-time	*Values of:* all basic types, all user-defined types, **address** type and **template**.
function	Dynamic at run-time	*Values of:* all basic types, all user-defined types, **address** type, **component** type, **port** type, **default**, **template** and **timer**.
altstep	Dynamic at run-time	*Values of:* all basic types, all user-defined types, **address** type, **component** type, **port** type, **default**, **template** and **timer**.
testcase	Dynamic at run-time	*Values of:* all basic types and of all user-defined types, **address** type and **template**.
signature	Dynamic at run-time	*Values of:* all basic types, all user-defined types and **address** type and **component** type.
NOTE 1 – **record of**, **set of**, **enumerated**, **port**, **component** and **sub-type definitions** do not allow parameterization. NOTE 2 – Examples of syntax and specific use of parameterization with the different language elements are given in the relevant clauses in this Recommendation.		

5.2.1 Parameter passing by reference and by value

5.2.1.0 General

By default, all actual 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. Timer parameters are identified by the keyword **timer**. Port parameters are identified by their port type. The keyword **inout** may optionally be used to denote passing by reference.

5.2.1.1 Parameters passed by reference

Passing parameters by reference has the following limitations:

a) only the formal parameter lists to altsteps, functions, signatures and testcases may contain pass-by-reference parameters;

   NOTE – There are further restrictions on how to use pass-by-reference parameters in signatures (see clause 23) and altsteps (see 16.2.1 and 21.3.1).

b) the actual parameters shall only be value or template variables, ports or timers.

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

5.2.1.2 Parameters passed by value

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

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

5.2.2 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);
```

5.2.3 Empty formal parameter list

If the formal parameter list of the TTCN-3 language elements **function**, **testcase**, **signature**, **altstep** 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 { … }
```

5.2.4 Nested parameter lists

All parameterized entities specified as an actual parameter shall have their own parameters resolved in the top-level 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 test case 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
{  :
  execute( TC001(MyTemplate(7)));
  :
}
```

5.2.5 Template-type formal parameters

5.2.5.1 Parameterization with templates and matching attributes

To enable templates or matching symbols to be passed as actual parameters the extra keyword **`template`** shall be added before the type field of the corresponding formal parameter. This makes the parameter a template-type and in effect extends the allowed parameters for the associated type to include the appropriate set of matching attributes (see Annex B) 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 optional,
   field2 := pattern "abc*xyz",
   field3 := true
}

// could be used as follows
pco1.receive(MyTemplate(?));
// Or as follows
pco1.receive(MyTemplate(omit));
```

5.2.5.2 Language elements using template-type parameters

Only **`function`**, **`testcase`**, **`altstep`** and **`template`** definitions can have template-type formal parameters.

EXAMPLE:

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

5.3 Scope rules

5.3.0 General

TTCN-3 provides seven basic units of scope:

a) module definitions part;

b) control part of a module;

c) component types;

d) functions;

e) altsteps;

f) test cases;

g) "blocks of statements and declarations" within compound statements.

NOTE 1 – Additional scoping rules for groups are given in 7.3.1.

NOTE 2 – Additional scoping rules for counters of **`for`** loops are given in 19.7.

Each unit of scope consists of (optional) declarations. The scope units: control part of a module, functions, test cases, altsteps and "blocks of statements and declarations" within compound statements may additionally specify some form of behaviour by using the TTCN-3 program statements and operations (see clause 18).

Definitions made in the module definitions part but outside of other scope units are globally visible, i.e., may be used elsewhere in the module, including all functions, test cases and altsteps defined within the module and the control part. Identifiers imported from other modules are also globally visible throughout the importing module.

Definitions made in the module control part have local visibility, i.e., can be used within the control part only.

Definitions made in a test component type may be used only in functions, test cases and altsteps referencing that component type or a consistent test component type (see 16.3) by a **`runs on`**-clause.

Test cases, altsteps and functions are individual scope units without any hierarchical relationship between them, i.e., declarations made at the beginning of their body have local visibility and shall only be used in the given test case, altstep or function (e.g., a declaration made in a test case is not visible in a function called by the test case or in an altstep used by the test case).

Compound statements, e.g., **`if-else`**-, **`while`**-, **`do-while`**-, or **`alt`**-statements include "blocks of statements and declarations". They may be used within the control part of a module, test cases, altsteps, functions, or may be embedded in other compound statements, e.g., an **`if-else`**-statement that is used within a **`while`**-loop.

The "blocks of statements and declarations" of compound statements and embedded compound statements have a hierarchical relationship both to the scope unit, including the given "block of statements and declarations", and to any embedded "block of statements and declarations". Declarations made within a "block of statements and declarations" have local visibility.

The hierarchy of scope units is shown in Figure 2. Declarations of a scope unit at a higher hierarchical level are visible in all units at lower levels within the same branch of the hierarchy. Declarations of a scope unit in a lower level of hierarchy are not visible to those units at a higher hierarchical level.

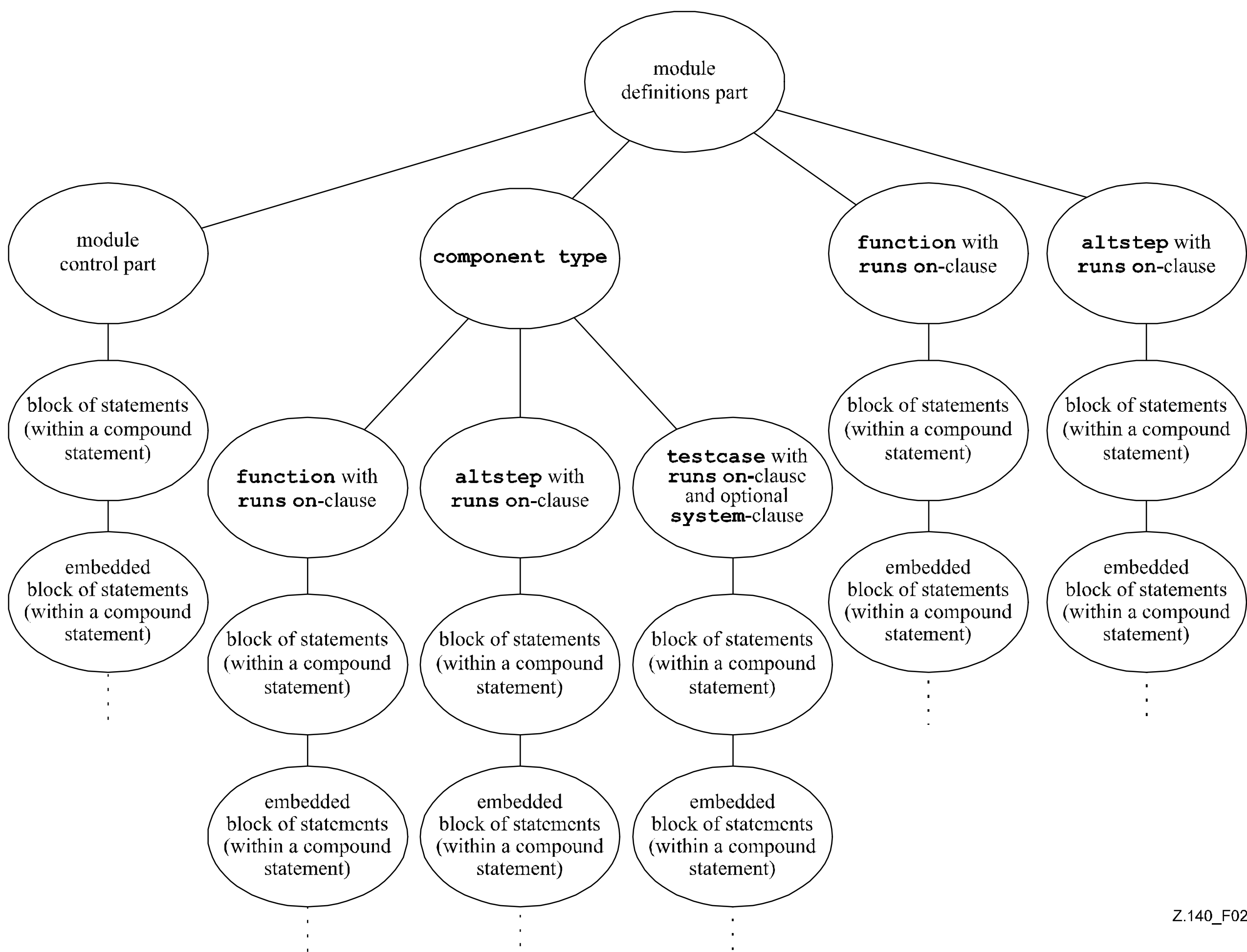


Figure 2/Z.140 – Hierarchy of scope units

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
        :

   }
}
```

5.3.1 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 5.3.0).

5.3.2 Uniqueness of identifiers

TTCN-3 requires uniqueness of identifiers, i.e., all identifiers in the same scope hierarchy shall be distinctive. 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 in the same branch of the scope hierarchy. Identifiers for fields of structured types, enumeration values and groups do not have to be globally unique; however, in the case of enumeration values the identifiers shall only be re-used for enumeration values within other enumerated types. The rules of identifier uniqueness shall also apply to identifiers of formal parameters.

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 Identifiers and keywords

TTCN-3 identifiers are case sensitive. TTCN-3 keywords shall be written in all lower-case letters (see Annex A). TTCN-3 keywords shall neither be used as identifiers of TTCN-3 objects nor as identifiers of objects imported from modules of other languages.

6 Types and values

6.0 General

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 **verdicttype**. Structured types such as **record** types, **set** types and **enumerated** types can be constructed from these basic types.

The special data type **anytype** is defined as the union of all known data types and the address type within a module.

Special types associated with test configurations such as **address**, **port** and **component** may be used to define the architecture of the test system (see clause 22).

The special type **default** may be used for the default handling (see clause 21).

The TTCN-3 types are summarized in Table 3.

Table 3/Z.140 – Overview of TTCN-3 types

Class of type	Keyword	Sub-type
Simple basic types	**integer**	range, list
	float	range, list
	boolean	list
	objid	list
	verdicttype	list
Basic string types	**bitstring**	list, length
	hexstring	list, length
	octetstring	list, length
	charstring	range, list, length, pattern
	universal charstring	range, list, length, pattern
Structured types	**record**	list (see Note)
	record of	list (see Note), length
	set	list (see Note)
	set of	list (see Note), length
	enumerated	list (see Note)
	union	list (see Note)
Special data types	**anytype**	list (see Note)
Special configuration types	**address**	
	port	
	component	
Special default types	**default**	
NOTE – List subtyping of these types is possible when defining a new constrained type from an already existing parent type but not directly at the declaration of the first parent type.		

6.1 Basic types and values

6.1.0 Simple basic types and values

TTCN-3 supports the following basic types:

a) **integer**: a type with distinguished values which are 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) **`float`**: a type to describe floating-point numbers.

In general, floating point numbers can be defined as: <*mantissa*> × <*base*>$^{<exponent>}$, where <*mantissa*> is a positive or negative integer, <*base*> a positive integer (in most cases 2, 10 or 16) and <*exponent*> a positive or negative integer.

In TTCN-3, the floating-point number value notation is restricted to a base with the value of 10. Floating point values can be expressed by using two forms of value notations:

- the decimal notation with a dot in a sequence of numbers, e.g., 1.23 (which represents 123×10^{-2}), 2.783 (i.e., 2783×10^{-3}) or –123.456789 (which represents $-123\,456\,789 \times 10^{-6}$); 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 123×10^{3}) or –12.3E-4 (which represents -123×10^{-5}).

NOTE – In contrast to the general definition of float values, the mantissa of the TTCN-3 value notation, beside integers, allows decimal numbers as well.

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

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

d) Void.

e) **`verdicttype`**: a type for use with test verdicts consisting of 5 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 1 – 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 the bit digits: 0 1, preceded by a single quote (') and followed by the pair of characters 'B.

EXAMPLE 1: `'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 (upper-case and lower-case letters can equally be used as hex digits):

0 1 2 3 4 5 6 7 8 9 a b c d e f 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.

EXAMPLE 2:
```
'AB01D'H
'ab01d'H
'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 (upper-case and lower-case letters can equally be used as hex digits):

0 1 2 3 4 5 6 7 8 9 a b c d e f 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.

EXAMPLE 3:
```
'FF96'O
'ff96'O
'Ff96'O
```

d) **`charstring:`** are types whose distinguished values are zero, one, or more characters of the version of ITU-T Rec. T.50 [9] complying with the International Reference Version (IRV) as specified in 8.2/T.50 [9].

NOTE 2 – The IRV version of the International Reference Alphabet (former International Alphabet No. 5 – IA5), described in ITU-T Rec. T.50 [9] is equivalent to the IRV version of ISO/IEC 646.

Values of **charstring** type shall be denoted by an arbitrary number (possibly zero) of characters from the relevant character set, preceded and followed by double quote (") or calculated using the predefined conversion function int2char with the positive integer value of their encoding as argument (see C.1).

NOTE 3 – The predefined conversion function is able to return single-character-length values only.

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.

EXAMPLE 4: `""abcd""` represents the literal string `"abcd"`.

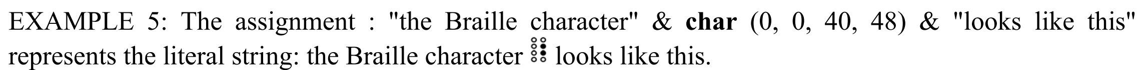
e) The character string type preceded by the keyword **universal** denotes types whose distinguished values are zero, one, or more characters from ISO/IEC 10646 [10].

universal charstring values can also be denoted by an arbitrary number (possibly zero) of characters from the relevant character set, preceded and followed by double quote ("), calculated using a predefined conversion function (see C.3) with the positive integer value of their encoding as argument or by a "quadruple".

NOTE 4 – The predefined conversion function is able to return single-character-length values only.

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.

The "quadruple" is only capable to denote a single character and denotes the character by the decimal values of its group, plane, row and cell according to ISO/IEC 10646 [10], preceded by the keyword **char** included into a pair of brackets and separated by commas (e.g., **char** (0, 0, 1, 113) denotes the Hungarian character "ű"). In cases where it is necessary to denote the character double quote (") in a string assigned according to the first method (within double quotes), the character is represented by a pair of double quotes on the same line with no intervening space characters. The two methods may be mixed within a single notation for a string value by using the concatenation operator.



EXAMPLE 5: The assignment : "the Braille character" & **char** (0, 0, 40, 48) & "looks like this" represents the literal string: the Braille character ⠰ looks like this.

NOTE 5 – Control characters can be denoted by using the predefined conversion function or the quadruple form.

By default, **universal charstring** shall conform to the UCS-4 coded representation form specified in 14.2 of ISO/IEC 10646 [10].

NOTE 6 – UCS-4 is an encoding format, which represents any UCS character on a fixed, 32 bits-length field.

This default encoding can be overridden using the defined variant attributes (see 28.2.3). The following useful character string types utf8string, bmpstring, utf16string and iso8859string using these attributes are defined in Annex E.

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).

EXAMPLE:

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

6.2 Sub-typing of basic types

6.2.0 General

User-defined types shall be denoted by the keyword **type**. With user-defined types it is possible to create sub-types (such as lists, ranges and length restrictions) on basic types, structured types and **anytype** according to Table 3.

6.2.1 Lists of values

TTCN-3 permits the specification of a list of distinguished values of basic types, structured types and **anytype** as listed in Table 3. The values in the list shall be of the root type and shall be a true subset of the values defined by the root type. The subtype defined by this list restricts the allowed values of the subtype to those values in the list.

EXAMPLE:

```
type bitstring MyListOfBitStrings ('01'B, '10'B, '11'B);
type float pi (3.1415926);
type charstring MyStringList ("abcd", "rgy", "xyz");
type universal charstring SpecialLetters (char(0, 0, 1, 111), char(0, 0, 1, 112), char(0, 0,
1,    113));
```

6.2.2 Ranges

6.2.2.0 General

TTCN-3 permits the specification of range constraints for the types **integer, charstring, universal charstring** and **float** (or derivations of these types). For **integer** and **float**, the subtype defined by the range restricts the allowed values of the subtype to the values in the range including the lower boundary and the upper boundary. In the case of **charstring** and **universal charstring** types, the range restricts the allowed values for each separate character in the strings. The boundaries shall evaluate to valid character positions according to the coded character set table(s) of the type (e.g., the given position shall not be empty). Empty positions between the lower and the upper boundaries are not considered to be valid values of the specified range.

EXAMPLE 1:

```
type integer MyIntegerRange (0 .. 255);
type float piRange (3.14 .. 3142E-3);
```

EXAMPLE 2:

```
type charstring MyCharString ("a" .. "z");
// Defines a string type of any length with each character within the specified range
type universal charstring MyUCharString1 ("a" .. "z");
// Defines a string type of any length with each character within the range from a to z
// (character codes from 97 to 122), like "abxyz";
// strings containing any other character (including control characters), like
// "abc2" are disallowed.
type universal charstring MyUCharString2 (char(0, 0, 1, 111) .. char(0, 0, 1, 113));
// Defines a string type of any length with each character within the range specified using
// the quadruple notation
```

6.2.2.1 Infinite ranges

In order to specify an infinite integer or float 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.

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

Void.

NOTE – This clause is replaced by clause 6.2.5.

6.2.3 String length restrictions

TTCN-3 permits the specification of length restrictions on string types. The length boundaries are based on different units 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).

EXAMPLE:

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

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

Table 4/Z.140 – 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.

6.2.4 Pattern sub-typing of character string types

TTCN-3 allows using character patterns specified in B.1.5 to constrain permitted values of **charstring** and **universal charstring** types. The type constraint shall use the **pattern** keyword followed by a character pattern. All values denoted by the pattern shall be a true subset of the type being sub-typed.

NOTE – Pattern sub-typing can be seen as a special form of list constraint, where members of the list are not defined by listing specific character strings but via a mechanism generating elements of the list.

EXAMPLE:

```
type charstring MyString (pattern "abc*xyz");
  // all permitted values of MyString have prefix abc and postfix xyz

type universal charstring MyUString (pattern "*\r\n")
  // all permitted values of MyUString are terminated by CR/LF

type charstring MyString2 (pattern "abc?\q{0,0,1,113}");
  // causes an error because the character denoted by the quadruple {0,0,1,113} is not a
  // legal character of the TTCN-3 charstring type

type MyString MyString3 (pattern "d*xyz");
  // causes an error because the type MyString does not contain a value starting with the
  // character d
```

6.2.5 Mixing sub-typing mechanisms

6.2.5.1 Mixing patterns, lists and ranges

Within **integer** and **float** (or derivations of these types) sub-type definitions it is allowed to mix lists and ranges. Overlapping of different constraints is not an error.

EXAMPLE 1:

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

Within **charstring** and **universal charstring** sub-type definitions it is not allowed to mix pattern, list or range constraints.

EXAMPLE 2:

```
type charstring MyCharStr0 ('gr', 'xyz');
  // contains character strings gr and xyz;

type charstring MyCharStr1 ('a'..'z');
  // contains character strings of arbitrary length containing characters a to z.

type charstring MyCharStr2 (pattern '[a-z]#(3,9)');
// contains character strings of length form 3 to 9 characters containing characters a to z
```

6.2.5.2 Using length restriction with other constraints

Within **bitstring**, **hexstring, octetstring** sub-type definitions lists and length restriction may be mixed in the same sub-type definition.

Within **charstring** and **universal charstring** sub-type definitions it is allowable to add a length restriction to constraints containing list, range or pattern sub-typing in the same sub-type definition.

When mixed with other constraints the length restriction shall be the last element of the sub-type definition. The length restriction takes effect jointly with other sub-typing mechanisms (i.e., the value set of the type consists of the common subset of the value sets identified by the list, range or pattern sub-typing and the length restriction).

EXAMPLE:

```
type charstring MyCharStr5 ('gr', 'xyz') length (1..9);
  // contains the character strings gr and xyz;

type charstring MyCharStr6 ( 'a'..'z') length (3..9);
  // contains character strings of length from 3 to 9 characters and containing characters
  // a to z

type charstring MyCharStr7 (pattern '[a-z]#(3,9)') length (1..9);
  // contains character strings of length form 3 to 9 characters containing characters a to z

type charstring MyCharStr8 (pattern '[a-z]#(3,9)') length (1..8);
  // contains character strings of length form 3 to 8 characters containing characters a to z

type charstring MyCharStr9 (pattern '[a-z]#(1,8)') length (1..9);
  // contains any character strings of length form 1 to 8 characters containing characters
  // a to z

type charstring MyCharStr10 ('gr', 'xyz') length (4);
  // contains no value (empty type).
```

6.3 Structured types and values

6.3.0 General

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 value list notation.

EXAMPLE 1:

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

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

When specifying partial values (i.e., setting the value of only a subset of the fields of a structured variable) using the assignment notation only the fields to be assigned values must be specified. Fields not mentioned are implicitly left unspecified. It is also possible to leave fields explicitly unspecified using the not used symbol "-". Using the value list notation all fields in the structure shall be specified either with a value, the not used symbol "-" or the **omit** keyword.

EXAMPLE 2:

```
var MyRecordType MyVariable:=                               //assignment notation
{
  field1 := '11001'B,
  // field2 implicitly unspecified
  field3 := "A string"
}

// Or
var MyRecordType MyVariable:=                               //assignment notation
{
  field1 := '11001'B,
  field2 := -, // field2 explicitly unspecified
  field3 := "A string"
}

// Or
var MyRecordType MyVariable:= {'11001'B, -, "A string"}      //value list notation
```

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

EXAMPLE 3:

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

In both the assignment notation and value list notation, optional fields shall be omitted by using the explicit **omit** value for the relevant field. The **omit** keyword shall not be used for mandatory fields. When re-assigning a previously initialized value, using the not used symbol or skipping a field in assignment notation will cause the relevant fields to remain unchanged.

EXAMPLE 4:

```
var MyRecordType MyVariable :=
{
  field1 := '111'B,
  field2 := false,
  field3 := -
}

MyVariable := { '10111'B, -, - };
// after this, MyVariable contains { '10111'B, false /* unchanged */, <undefined> }

MyVariable :=
{
  field2 := true
}
// after this, MyVariable contains { '10111'B, true, <undefined> }

MyVariable :=
{
  field1 := -,
  field2 := false,
  field3 := -
}
// after this, MyVariable contains { '10111'B, false, <undefined> }
```

6.3.1 Record type and values

6.3.1.0 General

TTCN-3 supports ordered structured types known as **record**. The elements of a **record** type may be any of the basic types or user-defined data 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** (but do not have to be globally unique). A constant that is of **record** type shall contain no variables or module parameters as field values, either directly or indirectly.

EXAMPLE 1:

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

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

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

EXAMPLE 2:

```
type record MyEmptyRecord { }
```

A **record** value is assigned on an individual element basis. The order of field values in the value list notation shall be the same as the order of fields in the related type definition.

EXAMPLE 3:

```
var integer MyIntegerValue := 1;

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

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

The same value specified with a value list.

EXAMPLE 4:

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

6.3.1.1 Referencing fields of a record type

Elements of a **`record`** shall be referenced by the dot notation *`TypeOrValueId.ElementId`*, where *`TypeOrValueId`* resolves to the name of a structured type or variable. ElementId shall resolve to the name of a field in a structured type.

EXAMPLE:

```
MyVar1 := MyRecord1.myElement1;
// If a record is nested within another type then the reference may look like this
MyVar2 := MyRecord1.myElement1.myElement2;
```

6.3.1.2 Optional elements in a record

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

EXAMPLE 1:

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

Optional fields shall be omitted using the omit symbol.

EXAMPLE 2:

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

// Note that this is not the same as writing,
// MyRecordValue:= {MyIntegerValue, -, "A string"};
// which would mean the value of field2 is unchanged
```

6.3.1.3 Nested type definitions for field types

TTCN-3 supports the definition of types for record fields nested within the **`record`** definition. Both the definition of new structured types (**`record`**, **`set`**, **`enumerated`**, **`set of`** and **`record of`**) and the specification of subtype constraints are possible.

EXAMPLE:

```
// record type with nested structured type definitions
type record MyNestedRecordType
{
   record
   {
      integer nestedField1,
      float   nestedField2
} outerField1,
enumerated {
      nestedEnum1,
```

```
            nestedEnum2
        } outerField2,
        record of boolean outerField3
}

// record type with nested sub-type definitions
type record MyRecordTypeWithSubtypedFields
{
        integer    field1 (1 .. 100),
        charstring field2 length ( 2 .. 255 )
}
```

6.3.2 Set type and values

6.3.2.0 General

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.

EXAMPLE:

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

The field identifiers are local to the set and shall be unique within the set (but do not have to be globally unique).

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

6.3.2.1 Referencing fields of a set type

Elements of a **set** shall be referenced by the dot notation (see 6.3.1.1).

EXAMPLE:

```
MyVar3 := MySet1.myElement1;
// If a set is nested in another type then the reference may look like this
MyVar4 := MyRecord1.myElement1.myElement2;
// Note, that the set type, of which the field with the identifier 'myElement2' is referenced,
// is embedded in a record type
```

6.3.2.2 Optional elements in a set

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

6.3.2.3 Nested type definition for field types

TTCN-3 supports the definition of types for set fields nested within the **set** definition, similar to the mechanism for record types described in 6.3.1.3.

6.3.3 Records and sets of single types

6.3.3.0 General

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**.

EXAMPLE 1:

```
type record length(10) of integer MyRecordOfType; // is a record of exactly 10 integers

type record length(0..10) of integer MyRecordOfType; // is a record of a maximum of 10 integers

type record length(10..infinity) of integer MyRecordOfType; // record of at least 10 integers

type set of boolean MySetOfType; // is an unlimited set of boolean values

type record length(0..10) of charstring StringArray length(12);
// is a record of a maximum of 10 strings each with exactly 12 characters
```

The value notation for **record of** and **set of** shall be a value list notation or an indexed notation for an individual element (the same value notation as for arrays, see 6.5). There is one exception from this general rule: in the case of defining modified templates, when the assignment notation is also allowed to be used (see 14.6.0).

When the value list notation is used, the first value in the list is assigned to the first element, the second list value is assigned to the second element, etc. No empty assignment is allowed (e.g., two commas, the second immediately following the first or only with white space between them). Elements to be left out of the assignment shall be explicitly skipped or omitted in the list.

Indexed value notations can be used on both the right-hand side and left-hand side of assignments. The index of the first element shall be zero and the index value shall not exceed the limitation placed by length subtyping. If the value of the element indicated by the index at the right hand of an assignment is undefined, this shall cause a semantical or run-time error. If an indexing operator at the left-hand side of an assignment refers to a non-existent element, the value at the right-hand side is assigned to the element and all elements with an index smaller than the actual index and without assigned value are created with an undefined value. Undefined elements are permitted only in transient states (while the value remains invisible). Sending a **record of** value with undefined elements shall cause a dynamic testcase error.

EXAMPLE 2:

```
// Given
type record of integer MyRecordOf;
var integer MyVar;
var MyRecordOf MyRecordVar := { 0, 1, 2, 3, 4 };

MyVar := MyRecordVar[0]; // the first element of the "record of" value (integer 0)
                         // is assigned to MyVar

// Indexed values are permitted on the left-hand side of assignments as well:
MyRecordVar[1] := MyVar; // MyVar is assigned to the second element
                         // value of MyRecordVar is { 0, 0, 2, 3, 4 }

// The assignment
MyRecordVar := { 0, 1, -, 2, omit };
// will change the value of MyRecordVar to{ 0, 1, 2 <unchanged>, 2};
// Note, that the 3rd element would be undefined if it had no previous assigned value.

// The assignment
MyRecordVar[6] := 6;

// will change the value of MyRecordVar to{ 0, 1, 2 , 2, <undefined>,   <undefined>, 6 };
  // Note the 5th and 6th elements (with indexs 4 and 5) had no assigned value before this
// last assignment and are therefore undefined.
```

NOTE – This makes it possible to copy **record of** values element by element in a for loop. For example, the function below reverses the elements of a **record of** value:

```
function reverse(in MyRecord src) return MyRecord
    {
    var MyRecord dest;
    var integer I;
    for(I := 0; I < sizeof(src); I:= I + 1) {
        dest[sizeof(src) - 1 - I] := src[I];
    }
    return dest;
    }
```

Embedded **record of** and **set of** types will result in a data structure similar to multidimensional arrays (see 6.5).

EXAMPLE 3:

```
// Given
type record of integer MyBasicRecordOfType;
type record of MyBasicRecordOfType  MyRecordOfType;

// Then, the variable myRecordOfArray will have similar attributes to a two-dimensional array:
var MyRecordOfType myRecordOfArray;
// and reference to a particular element would look like this
// (value of the second element of the third 'MyBasicRecordOfType' construct)
myRecordOfArray [2][1] := 1;
```

6.3.3.1 Nested type definitions

TTCN-3 supports the definition of the aggregated type nested with the **record of** or **set of** definition. Both the definition of new structured types (**record**, **set**, **enumerated**, **set of** and **record of**) and the specification of subtype constraints are possible.

EXAMPLE:

```
type record of enumerated { red, green, blue } ColorList;
type record length (10) of record length (10) of integer Matrix;
type set of record { charstring id, charstring val } GenericParameters;
```

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. Such distinct values are called enumerations. Each enumeration shall have an identifier. Operations on enumerated types shall only use these identifiers and are restricted to assignment, equivalence and ordering operators. Enumeration identifiers shall be unique within the enumerated type (but do not have to be globally unique) and are consequently visible within the context of the given type only. Enumeration identifiers shall only be re-used within other structured type definitions and shall not be used for identifiers of local or global visibility at the same or a lower level of the same branch of the scope hierarchy (see scope hierarchy in 5.3.0).

EXAMPLE 1:

```
type enumerated MyFirstEnumType {
  Monday, Tuesday, Wednesday, Thursday, Friday
};

type integer Monday;
// This definition is illegal, as the name of the type has local or global visibility

type enumerated MySecondEnumType {
  Saturday, Sunday, Monday
};
// This definition is legal as it reuses the Monday enumeration identifier within
// a different enumerated type

type record MyRecordType {
  integer      Monday
};
// This definition is legal as it reuses the Monday enumeration identifier within
// a distinct structured type as identifier of a given field of this type

type record MyNewRecordType {
  MyFirstEnumType firstField,
  integer      secondField
};

var MyNewRecordType newRecordValue := { Monday, 0 }
// MyFirstEnumType is implicitly referenced via the firstField element of MyNewRecordType

const integer Monday := 7
// This definition is illegal as it reuses the Monday enumeration identifier for a
// different TTCN-3 object within the same scope unit
```

Each enumeration may optionally have an assigned integer value, which is defined after the name of the enumeration in parenthesis. Each assigned integer number shall be distinct within a single **enumerated** type. For each enumeration without an assigned integer value, the system successively associates an integer number in the textual order of the enumerations, starting at the left-hand side, beginning with zero, by step 1 and skipping any number occupied in any of the enumerations with a manually assigned value. These values are only used by the system to allow the use of relational operators.

NOTE 1 – The integer value may also be used by the system to encode/decode enumerated values. This, however, is outside of the scope of this Recommendation (with the exception that TTCN-3 allows the association of encoding attributes to TTCN-3 items).

For any instantiation or value reference of an **enumerated** type, the given type shall be implicitly or explicitly referenced.

NOTE 2 – If the enumerated type is an element of a user-defined structured type, the enumerated type is implicitly referenced via the given element (i.e., by the identifier of the element or the position of the value in a value list notation) at value assignment, instantiation, etc.

EXAMPLE 2:

```
// Valid instantiations of MyFirstEnumType and MySecondEnumType would be
var MyFirstEnumType Today := Tuesday;
var MySecondEnumType Tomorrow := Monday;

// But the following statement is illegal because the two enumeration types are not compatible
Today := Tomorrow
```

6.3.5 Unions

6.3.5.0 General

TTCN-3 supports the **union** type. The **union** type is a collection of fields, each one identified by an identifier. 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.

EXAMPLE:

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

// A valid instantiation of MyUnionType would be
var MyUnionType age, oneYearOlder;
var integer ageInMonths;

age.number := 34;       // value notation by referencing the field. Note, that this
                        // notation makes the given field to be the chosen one
oneYearOlder := {number := age.number+1};

ageInMonths := age.number * 12;
```

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

6.3.5.1 Referencing fields of a union type

Fields of a **union** type shall be referenced by the dot notation (see 6.3.1.1).

EXAMPLE:

```
MyVar5 := MyUnion1.myChoice1;
// If a union type is nested in another type then the reference may look like this
MyVar6 := MyRecord1.myElement1.myChoice2;
// Note, that the union type, of which the field with the identifier 'myChoice2' is referenced,
// is embedded in a record type
```

6.3.5.2 Optionality and union

Optional fields are not allowed for the **union** type, which means that the **optional** keyword shall not be used with **union** types.

6.3.5.3 Nested type definition for field types

TTCN-3 supports the definition of types for union fields nested within the union definition, similar to the mechanism for record types described in 6.3.1.3.

6.4 The anytype

The special type **anytype** is defined as a shorthand for the union of all known data types and the address type in a TTCN-3 module. The definition of the term "known types" is given in 3.1, i.e., the **anytype** shall comprise all the known data types but not the **port**, **component**, and **default** types. The **address** type shall be included if it has been explicitly defined within that module.

The fieldnames of the **anytype** shall be uniquely identified by the corresponding type names.

NOTE 1 – As a result of this requirement, imported types with clashing names (either with an identifier of a definition in the importing module or with an identifier imported from a third module) cannot be reached via the **anytype** of the importing module.

EXAMPLE:

```
// A valid usage of anytype would be
var anytype MyVarOne, MyVarTwo;
var integer MyVarThree;

MyVarOne.integer := 34;
MyVarTwo := {integer := MyVarOne.integer + 1};

MyVarThree := MyVarOne.integer * 12;
```

The **anytype** is defined locally for each module and (like the other predefined types) cannot be directly imported by another module. However, a user-defined type of the type **anytype** can be imported by another module. The effect of this is that all types of that module are imported.

NOTE 2 – The user-defined type of **anytype** "contains" all types imported into the module where it is declared. Importing such a user-defined type into a module may cause side effects and hence due caution should be given to such cases.

6.5 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. Arrays may be declared as single- or multi-dimensional. Array dimensions shall be specified using constant expressions, which shall evaluate to a positive **integer** values.

EXAMPLE 1:

```
var integer MyArray1[3];    // Instantiates an integer array of 3 elements with the index 0 to 2
var integer MyArray2[2][3]; // Instantiates a two-dimensional integer array of 2 × 3 elements with
                            // indexes from (0,0) to (1,2)
```

Array elements are accessed by means of the index notation (`[]`), which must specify a valid index within the array's range. Individual elements of multi-dimensional arrays can access by repeated use of the index notation. Accessing elements outside the array's range will cause a compile-time or test case error.

EXAMPLE 2:

```
MyArray1[1] := 5;
MyArray2[1][2] := 12;

MyArray1[4] := 12;    // ERROR: index must be between 0 and 2
MyArray2[3][2] := 15; // ERROR: first index must be 0 or 1
```

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.

EXAMPLE 3:

```
var integer MyArray3[1 .. 5]; // Instantiates an integer array of 5 elements
                              // with the index 1 to 5
MyArray3[1] := 10; // Lowest index
MyArray3[5] := 50; // Highest index

var integer MyArray4[1 .. 5][2 .. 3 ];  // Instantiates a two-dimensional integer array of
                                        // 5 × 2 elements with indexes from (1,2) to (5,3)
```

The values of array elements shall be compatible with the corresponding variable declaration. Values may be assigned individually by a value list notation or indexed notation or more than one or all at once by a value list notation. When the value list notation is used, the first value of the list is assigned to the first element of the array (the element with index 0), the second value to the second element, etc. Elements to be left out from the assignment shall be explicitly skipped or omitted in the list. For assigning values to multi-dimensional arrays, each dimension that is assigned shall resolve to a set of values enclosed in curly braces. When specifying values for multi-dimensional arrays, the leftmost dimension corresponds to the outermost structure of the value, and the rightmost dimension to the innermost structure. The use of array slices of multi-dimensional arrays, i.e., when the number of indexes of the array value is less than the number of dimensions in the corresponding array definition, is allowed. Indexes of array slices shall correspond to the dimensions of the array definition from left to right (i.e., the first index of the slice corresponds to the first dimension of the definition). Slice indexes shall conform to the related array definition dimensions.

EXAMPLE 4:

```
MyArray1[0]:= 10;
MyArray1[1]:= 20;
MyArray1[3]:= 30;

// or using a value list
MyArray1:= {10, 20, -, 30};

MyArray4:= {{1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10}};
  // The array value is completely defined
```

```
var integer MyArray5[2][3][4] :=
{
  {
    {1, 2, 3, 4}, // assigns a value to MyArray5 slice [0][0]
    {5, 6, 7, 8}, // assigns a value to MyArray5 slice [0][1]
    {9, 10, 11, 12} // assigns a value to MyArray5 slice [0][2]
  }, // end assignments to MyArray5 slice [0]
  {
    {13, 14, 15, 16}, {17, 18, 19, 20}, {21, 22, 23, 24}
  } // assigns a value to MyArray5 slice [1]
};

MyArray4[2] := {20, 20};
  // yields {{1, 2}, {3, 4}, {20, 20}, {7, 8}, {9, 10}};
MyArray5[1] := { {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}};
  // yields {{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}},
  //         {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}}};

MyArray5[0][2] := {3, 3, 3, 3};
  // yields {{{1, 2, 3, 4}, {5, 6, 7, 8}, {3, 3, 3, 3}},
  //         {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}}};

var integer MyArrayInvalid[2][2];
MyArrayInvalid := { 1, 2, 3, 4 }
  // invalid as the dimension of the value notation does not corresponds to the dimensions
  // of the definition
MyArrayInvalid[2] := { 1, 2 }
  // invalid as the index of the slice should be 0 or 1
```

NOTE – An alternative way to use multi-dimensional data structures is via employing the record, record of, set or set of types.

EXAMPLE 5:

```
// 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;
```

6.6 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.

6.7 Type compatibility

6.7.0 General

Generally, TTCN-3 requires type compatibility of values at assignments, instantiations and comparison.

For the purpose of this clause the actual value to be assigned, passed as parameter, etc., is called value "b". The type of value "b" is called type "B". The type of the formal parameter, which is to obtain the actual value of value "b" is called type "A".

6.7.1 Type compatibility of non-structured types

For variables, constants, templates, etc., of simple basic types and bitstring, hexstring and octetstring types the value "b" is compatible to type "A" if type "B" resolves to the same root type as type "A" (e.g., **integer**) and it does not violate subtyping (e.g., ranges, length restrictions) of type "A".

EXAMPLE:

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

```
// Then
y := 5; // is a valid assignment

x := y;
// is a valid assignment, because y has the same root type as x and no subtyping is violated

x := 20; // is a valid assignment
y := x;
// is NOT a valid assignment, because the value of x is out of the range of MyInteger

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

//Given
type charstring MyChar length (1);
type charstring MySingleChar length (1);
var MyChar myCharacter;
var charstring myCharString;
var MySingleChar mySingleCharString := "B";

//Then
myCharString := mySingleCharString;
//is a valid assignment as charstring restricted to length 1 is compatible with charstring.
myCharacter := mySingleCharString;
//is a valid assignment as two single-character-length charstrings are compatible.

//Given
myCharString := "abcd";

//Then
myCharacter := myCharString[1];
//is valid as the r.h.s. notation addresses a single element from the string

//Given
var charstring myCharacterArray [5] := {"A", "B", "C", "D", "E"}

//Then
myCharString := myCharacterArray[1];
//is valid and assigns the value "B" to myCharString;
```

For variables, constants, templates, etc., of **`charstring`** type, value 'b' is compatible with a **`universal charstring`** type 'A' unless it violates any type constraint specification (range, list or length) of type "A".

For variables, constants, templates, etc., of **`universal charstring`** type, value 'b' is compatible with a **`charstring`** type 'A' if all characters used in value 'b' have their corresponding characters (i.e., the same control or graphical character using the same character code) in the type **`charstring`** and it does not violate any type constraint specification (range, list or length) of type "A".

6.7.2 Type compatibility of structured types

6.7.2.0 General

In the case of structured types (except the **`enumerated`** type), a value "b" of type "B" is compatible with type "A", if the effective value structures of type "B" and type "A" are compatible, in which case assignments, instantiations and comparisons are allowed.

6.7.2.1 Type compatibility of enumerated types

Enumerated types are never compatible with other basic or structured types (i.e., for enumerated types, strong typing is required).

6.7.2.2 Type compatibility of record and record of types

For **`record`** types the effective value structures are compatible if: the number, and optional aspect of the fields in the textual order of definition, are identical; the types of each field are compatible; and, the value of each existing field of the value "b" is compatible with the type of its corresponding field in type "A". The value of each field in the value "b" are assigned to the corresponding field in the value of type "A".

EXAMPLE 1:

```
// Given
type record AType {
      integer     a(0..10)    optional,
      integer     b(0..10)    optional,
```

```
      boolean        c
}

type record BType {
      integer        a       optional,
      integer     b(0..10)   optional,
      boolean        c
}

type record CType {     // type with different field names
      integer        d  optional,
      integer        e  optional,
      boolean        f

}

type record DType {     // type with field c optional
      integer        a  optional,
      integer        b  optional,
      boolean        c  optional
}

type record EType {     // type with an extra field d
      integer        a  optional,
      integer        b  optional,
      boolean        c,
        float        d  optional
}

      var AType MyVarA := { -, 1, true};
      var BType MyVarB := { omit, 2, true};
      var CType MyVarC := { 3, omit,  true};
      var DType MyVarD := { 4, 4, true};
      var EType MyVarE := { 5, 5, true, omit};

      // Then

      MyVarA := MyVarB;  // is a valid assignment,
                         // value of MyVarA is ( a := <undefined>, b:= 2, c:= true)
      MyVarC := MyVarB;  // is a valid assignment
                         // value of MyVarC is ( d := <undefined>, e:= 2, f:= true)
      MyVarA := MyVarD;  // is NOT a valid assignment because the optionality of fields does not
                         // match
      MyVarA := MyVarE;  // is NOT a valid assignment because the number of fields does not match

      MyVarC := { d:= 20 };// actual value of MyVarC is { d:=20, e:=2,f:= true }
      MyVarA := MyVarC   // is NOT a valid assignment because field 'd' of MyVarC violates subtyping
                         // of field 'a' of AType
```

For **record of** types and arrays, the effective value structures are compatible if their component types are compatible and value "b" of type "B" does not violate any length subtyping of the **record of** type or dimension of the array of type "A". Values of elements of the value "b" shall be assigned sequentially to the instance of type "A", including undefined elements.

record of types and single-dimension arrays are compatible with **record** types if their effective value structures are compatible and the number of elements of value "b" of the **record of** type "B" or the dimension of array "b" is exactly the same as the number of elements of the **record** type "A". Optionality of the **record** type fields has no importance when determining compatibility, i.e., it does not affect the counting of fields (which means that optional fields shall always be included in the count). Assignment of the element values of the **record of** type or array to the instance of a **record** type shall be in the textual order of the corresponding **record** type definition, including undefined elements. If an element with an undefined value is assigned to an optional element of the **record**, this will cause the optional element to be omitted. An attempt to assign an element with undefined value to a mandatory element of the **record** shall cause an error.

NOTE – If the **record of** type has no length restriction or the length restriction exceeds the number of elements of the compared **record** type and the index of any defined element of the **record of** value is less or equal than the number of elements of the **record** type minus one, then the compatibility requirement is always fulfilled.

Values of a **record** type can also be assigned to an instance of a **record of** type or a single-dimension array if no length restriction of the **record of** type is violated, or the dimension of the array is more than or equal to the number of elements of the **record** type. Optional elements missing in the **record** value shall be assigned as elements with undefined values.

EXAMPLE 2:

```
// Given
type record HType {
  integer a,
  integer b optional,
  integer c
}

type record of integer IType

var HType MyVarH := { 1, omit, 2};
var IType MyVarI;
var integer MyArrayVar[2];

// Then

MyArrayVar := MyVarH;
// is a valid assignment as type of MyArrayVar and HType are compatible

MyVarI := MyVarH;
// is a valid assignment as the types are compatible and no subtyping is violated

MyVarI := { 3, 4 };
MyVarH := MyVarI;
// is NOT a valid assignment as the mandatory field 'c' of Htype receives no value
```

6.7.2.3 Type compatibility of set and set of types

set types are only type compatible with other **set** types and **set of** types. For **set** types and for **set of** types the same compatibility rules shall apply as for **record** and **record of** types.

NOTE 1 – This implies that, though the order of elements at sending and receipt is unknown, when determining type compatibility for **set** types, the textual order of the fields in the type definition is decisive.

NOTE 2 – In **set** values, the order of fields may be arbitrary; however, this does not affect type compatibility as field names unambiguously identify which fields of the related **set** type correspond to which **set** value fields.

EXAMPLE:

```
// Given
type set FType {
  integer aoptional,
  integer boptional,
  boolean c
}

type set GType {
  integer d   optional,
  integer e   optional,
  boolean f
}

var FType MyVarF := { a:=1, c:=true };
var GType MyVarG := { f:=true, d:=7};

// Then

MyVarF := MyVarG; // is a valid assignment as types FType and GType are compatible

MyVarF := MyVarA; // is NOT a valid assignment as MyVarA is a record type
```

6.7.2.4 Compatibility between sub-structures

The rules defined in this clause for structured types compatibility are also valid for the sub-structure of such types.

EXAMPLE:

```
// Given
type record JType {
      HType H,
      integer  b optional,
      integer  c
}
```

```
var JType MyVarJ

// If considering the declarations above, then


MyVarJ.H := MyVarH;
// is a valid assignment as the type of field H of JType and HType are compatible

MyVarI := MyVarJ.H;
// is a valid assignment as IType and the type of field H of JType are compatible
```

6.7.3 Type compatibility of component types

Type compatibility of component types has to be considered in two different conditions.

1) Compatibility of a component reference value with a component type (e.g., when passing a component reference as an actual parameter to a function or an altstep or when assigning a component reference value to a variable of different component type): a component reference "b" of component type "B" is compatible with component type "A" if all definitions of "A" have identical definitions in "B".
2) **`Runs on`** compatibility: a function or altsteps referring to component type "A" in its **`runs on`** clause may be called or started on a component instance of type 'B' if all the definitions of "A" have identical definitions in "B".

Identity of definitions in 'A' with definitions of 'B' is determined based on the following rules:

- For port instances, both the type and the identifier shall be identical.
- For timer instances, identifiers shall be identical, and either both shall have identical initial durations or both shall have no initial duration.
- For variable instances and constant definitions, the identifiers, the types and initialization values shall be identical (in case of variables, this means that either missing in both definitions or be the same).
- For local template definitions the identifiers, the types, the formal parameter lists and the assigned template or template field values shall be identical.

6.7.4 Type compatibility of communication operations

The communication operations (see clause 23) **`send`**, **`receive`**, **`trigger`**, **`call`**, **`getcall`**, **`reply`**, **`getreply`** and **`raise`** are exceptions to the weaker rule of type compatibility and require strong typing. The types of values or templates directly used as parameters to these operations must also be explicitly defined in the associated port type definition. Strong typing also applies to storing the received value, address or component reference during a **`receive`** or **`trigger`** operation.

6.7.5 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 root type, then either one of the predefined conversion functions defined in Annex C or a user-defined function shall be used.

EXAMPLE:

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

7 Modules

7.0 General

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 definitions part (optional), and a module control part (optional).

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. In addition, a module specification can carry an optional attribute identified by the **`language`** keyword that identifies the edition of the TTCN-3 language in which the module is specified. Currently the following language strings are supported: "TTCN-3:2001" for a module specification complying with the TTCN-3 2001 edition, "TTCN-3:2003" for the 2003 edition, "TTCN-3:2005" for the 2006 edition.

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

EXAMPLE:

```
module SIPTestSuite language "TTCN-3:2003"
{ … }
```

7.2 Module parameters

7.2.0 General

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. Module parameters are declared by specifying the type and listing their identifiers following the keyword **`modulepar`**. Module parameters shall not be of port type, default type or component type. A module parameter shall only be of type address if the address type is explicitly defined within the associated module. Module parameters shall be declared within the module definition part only. More than one occurrence of module parameters declaration is allowed but each parameter shall be declared only once (i.e., redefinition of the module parameter is not allowed).

EXAMPLE:

```
module MyModulewithParameters
{
  modulepar integer TS_Par0, TS_Par1;
  modulepar boolean TS_Par2;
  modulepar hexstring TS_Par3;
}
```

7.2.1 Default values for module parameters

It is allowed to specify default values for module parameters. This shall be done by an assignment in the module parameter list. A default value can be a literal value only and can merely be assigned at the place of the declaration of the parameter. If the test system does not provide an actual run-time value for the given parameter, the default value shall be used during test execution; otherwise, the actual value provided by the test system shall be used.

EXAMPLE:

```
module MyModuleDefaultParameter
{
  modulepar integer TS_Par0 := 0, TS_Par1;
  modulepar boolean TS_Par2 := true;
  :
}
```

7.3 Module definitions part

7.3.0 General

The module definitions part specifies the top-level definitions of the module and may import identifiers from other modules. Scope rules for declarations made in the module definitions part and imported declarations are given in 5.3. 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, functions, altsteps or component types.

NOTE – TTCN-3 does not support the declaration of variables in the module definitions part. This means that global variables cannot be defined in TTCN-3. However, variables defined in a test component may be used by all test cases, functions, etc., running on that component and variables defined in the control part provide the ability to keep their values independently of test case execution.

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 module if required. Groups and nested groups have no scoping *except* in the context of group identifiers and attributes given to a group by an associated **`with`** statement. This means:

- Group identifiers across the whole module need not necessarily be unique. However, all group identifiers of subgroups of a single group shall be unique. If necessary, the dot notation shall be used to identify sub-groups within the group hierarchy uniquely, e.g., for the import of a specific sub-group.
- Overriding rules for attributes are given in 28.4.

EXAMPLE:

```
module MyModule {
   :
   // A collection of definitions
   group MyGroup {
       const integer MyConst:= 1;
        :
       type record MyMessageType { … };
       group MyGroup1 { // Sub-group with definitions
            type record AnotherMessageType { … };
            const boolean MyBoolean := false
       }
   }

   // A group of altsteps
   group MyStepLibrary {
       group MyGroup1 { // Sub-group with the same name as the sub-group with definitions
            altstep MyStep11() { … }
            altstep MyStep12() { … }
             :
            altstep MyStep1n() { … }
       }
       group MyGroup2 {
            altstep MyStep21() { … }
            altstep MyStep22() { … }
            :
            altstep MyStep2n() { … }
       }
   }
   :
}

// An import statement that imports MyGroup1 within MyStepLibrary
import from MyModule {
   group MyStepLibrary.MyGroup1
}
```

7.4 Module control part

The module control part may contain local definitions and describes the execution order (possibly repetitive) 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
           :
           execute( MyTestCase1()); // sequential execution of test cases
           execute( MyTestCase2());
           :
      }
}
```

7.5 Importing from modules

7.5.0 General

It is possible to reuse 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. The mechanism to change attributes of imported definitions is explained in 28.6.

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

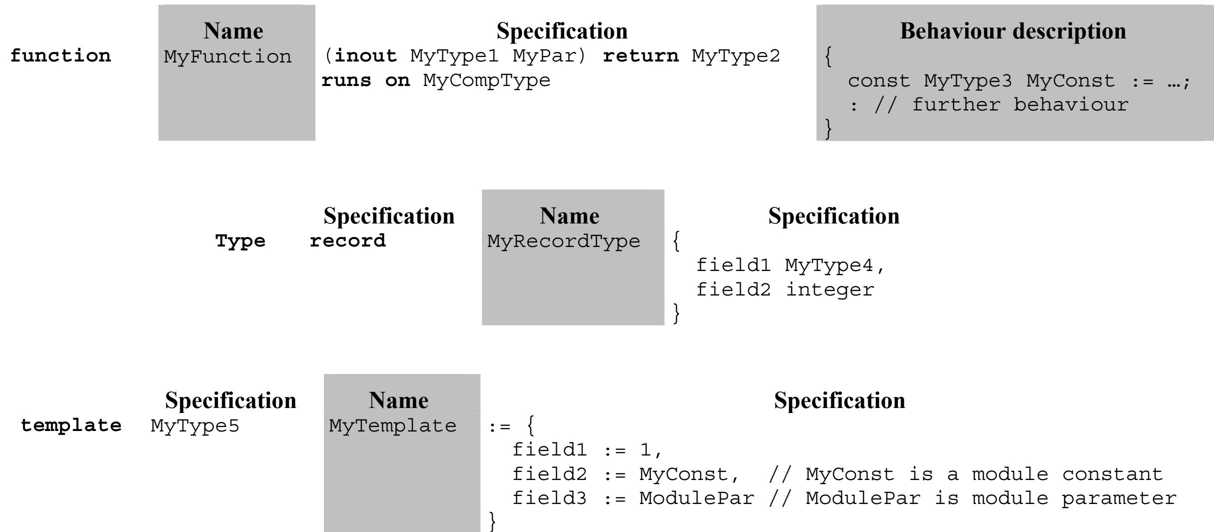
EXAMPLE:

```
module MyModuleA
{     // This module contains definitions and imported definitions
      :
      const integer MyConstant := 1;
      import from MyModuleB all;    // Scope of the imported definitions is global to MyModuleA
      import from MyModuleC {
           type MyType1, MyType2;
           template all
      }
      type record MyMessageType { … }
      :
      function MyBehaviourC()
      {
           const integer MyConstant := 2;
           // import cannot be used here
             :
      }
      :
      control
      {    // import cannot be used here
           :
      }
}
```

7.5.1 Structure of importable definitions

TTCN-3 supports the import of the following definitions: module parameters, user-defined types, signatures, constants, external constants, data templates, signature templates, functions, external functions, altsteps and test cases. Each definition has a *name* (defines the identifier of the definition, e.g., a function name), a *specification* (e.g., a type specification or a signature of a function) and in the case of functions, altsteps and test cases an associated *behaviour description*.

EXAMPLE:



Behaviour descriptions have no effect on the import mechanism, because their internals are considered to be invisible to the importer when the corresponding functions, altsteps or test cases are imported. Thus, they are not considered in the following descriptions.

The specification part of an importable definition contains *local definitions* (e.g., field names of structured type definitions or values of enumerated types) and *referenced definitions* (e.g., references to type definitions, templates, constants or module parameters). For the examples above, this means:

	Name	Local definitions	Referenced definitions
function	MyFunction	MyPar	MyType1, MyType2, MyCompType
type	MyRecordType	field1, field2	MyType4, integer
template	MyTemplate		MyType5, field1, field2, field3, MyConst, ModulePar

NOTE 1 – The local definitions column refers to identifiers only that are newly defined in the importable definition. Values assigned to individual fields of importable definitions, e.g., in template definitions, may also be considered as local definitions, but they are not important for the explanation of the import mechanism.

NOTE 2 – The referenced definitions field1, field2 and field3 of template MyTemplate are the field names of MyType5, i.e., they are referenced via MyType5.

Referenced definitions are also importable definitions, i.e., the source of a referenced definition can again be structured into a name and a specification part, and the specification part also contains local and referenced definitions. In other words, an importable definition may be built up recursively from other importable definitions.

The TTCN-3 import mechanism is related to the local and referenced definitions used in the specification part of the importable definitions. Therefore, Table 5 specifies the possible local and referenced definitions of importable definitions.

Table 5/Z.140 – Possible local and referenced definitions of importable definitions

Importable definition	Possible local definitions	Possible referenced definitions
Module parameter		Module parameter type
User-defined type (for all)	Parameter names	Parameter type
• enumerated type	Concrete values	
• structured type	Field names	Field types
• port type		Message types, signatures
• component type	Constant names, variable names, timer names and port names	Constant types, variable types, port types
Signature	Parameter names	Parameter types, return type, types of exceptions
Constant		Constant type
External constant		Constant type
Data template	Parameter names	Template type, parameter types, constants, module parameters, functions
Signature template		Signature definition, constants, module parameters functions
Function	Parameter names	Parameter types, return type, component type (**runs on**-clause)
External function	Parameter names	Parameter types, return type
Altstep	Parameter names	Parameter types, component type (**runs on**-clause)
Test case	Parameter names	Parameter types, component types (**runs on**- and **system**-clause)

The TTCN-3 import mechanism distinguishes between the *identifier of a referenced definition* and the *information necessary for the usage of a referenced definition* within the imported definition. For the usage, the identifier of a referenced definition is not required and, therefore, is not imported automatically.

7.5.2 Rules on using import

On using import, the following rules shall be applied:

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

b) Only direct importing from the source module of a definition (i.e., the module where the actual definition for the identifier referenced in the **import** statement resides) is allowed.

c) A definition is imported together with its name and all local definitions.

NOTE 1 – A local definition, e.g., a field name of a user-defined record type, only has meaning in the context of the definitions in which it is defined, e.g., a field name of a record type can only be used to access a field of the record type and not outside this context.

d) A definition is imported together with all information of referenced definitions that are necessary for the usage of the referenced definition.

NOTE 2 – Import statements are transitive, e.g., if a module A imports a definition from module B that uses a type reference defined in module C, the corresponding information necessary for the usage of that type is automatically imported into module A.

e) Identifiers of referenced definitions are not automatically imported.

NOTE 3 – If the referenced definitions are wished to be used in the importing module, they shall be explicitly imported from its source module.

f) When importing a function, altstep or test case, the corresponding behaviour specifications and all definitions used inside the behaviour specifications remain invisible for the importing module.

g) Cyclic imports are forbidden.

EXAMPLE:

```
module ModuleONE {

  modulepar integer ModPar1, ModPar2 := 7

  type record RecordType_T1 {
     integer  Field1_T1,
     boolean  Field2_T1
  }
```

```
   type record RecordType_T2 {
      RecordType_T1    Field1_T2,  // Use of RecordType_T1
      RecordType_T1    Field2_T2,
      integer      Field3_T2
   }

   const integer MyConst := 13;

   template RecordType_T2 Template_T2 (RecordType_T1 TempPar_T2):= { // parameterized template
      Field1_T2 := TempPar_T2,      // Reference to template parameter
      Field2_T2 := {MyConst, true}, // Reference to module constant
      Field3_T2 := ModPar1          // Reference to a module parameter
   }

} // end module ModuleONE


module ModuleTWO {

   import from ModuleONE {
      template Template_T2
   }

   // Only the names Template_T2 and TempPar_T2 will be visible in ModuleTWO. Please note, that
   // the identifier TempPar_T2 can only be used in the context of Template_T2, e.g. when
   // providing an actual parameter value. All information
   // necessary for the usage of Template_T2, e.g. for type checking purposes, are imported
   // for the referenced definitions RecordType_T2, RecordType_T1, Field1_T2, Field2_T2,
   // Field3_T3, MyConst and ModPar1, but their identifiers are not visible in ModuleTWO.
   // This means, e.g. it is not possible to use the constant MyConst or to declare a
   // variable of type RecordType_T1 or RecordType_T2 in ModuleTWO without explicitly importing
   // these types

   import from ModuleONE {
      modulepar ModPar2
   }

   // The module parameter ModPar2 of ModuleONE is imported from ModuleONE and
   // can be used like an integer constant

} // end module ModuleTWO


module ModuleTHREE {

   import from ModuleONE all; // imports all definitions from ModuleONE

   type port MyPortType {
      inout RecordType_T2
   }

   type component MyCompType {
      var integer MyComponentVar := ModPar2; // Reference to a module parameter of ModuleONE
      port MyPortType MyPort
   }

   function MyFunction () return integer {
      return MyConst    // Returns a module constant defined in ModuleONE
   }

   testcase MyTestCase (out RecordType_T2 MyPar) runs on MyCompType {

      var integer MyTCVar := ModPar2; // Reference to a module parameter of ModuleONE

      MyPort.send(Template_T2); // Sending a template defined in ModuleONE
      MyPort.receive(RecordType_T2 : ?) -> value MyPar; // The received value is assigned
                                                        // to the out parameter MyPar.

   } // end testcase MyTestCase

} // end ModuleTHREE


module ModuleFOUR {

   import from ModuleTHREE {
      testcase MyTestCase
   }
```

```
    // Only the names MyTestCase and MyPar will be visible and usable in ModuleFOUR.
    // Type information for RecordType_T2 is imported via ModuleTHREE from ModuleONE and
    // type information for MyCompType is imported from ModuleTHREE. All definitions
    // used in the behaviour part of MyTestCase remain hidden for the user of ModuleFOUR.

} // end ModuleFOUR
```

7.5.3 Void

7.5.4 Importing single definitions

Single definitions may be imported.

EXAMPLE:

```
import from MyModuleA {
  type MyType1          // imports one type definition from MyModuleA
}

import from MyModuleB {
  type MyType2, Mytype3, MyType4;  // imports three types
  template MyTemplate1;            // imports one template
  const MyConst1, MyConst2         // imports two constants
}
```

7.5.5 Importing all definitions of a module

All definitions of a module definitions part may be imported using the **all** keyword next to the module name. If all definitions of a module are imported by using the **all** keyword, no other form of import (import of single definitions, import of the same kind, etc.) shall be used for the same **import** statement.

EXAMPLE 1:

```
import from MyModule all;
```

If some declarations are wished not to be imported, their kinds and identifiers shall be listed in the exception list within a pair of curly brackets following the **except** keyword.

EXAMPLE 2:

```
import from MyModule all except {
  type MyType3, MyType5
     // excludes type declarations MyType3 and MyType5 from the import statement
     // but imports all other declarations of MyModule
}
```

The **all** keyword is also allowed to be used in the exception list; this will exclude all declarations of the same kind from the import statement.

EXAMPLE 3:

```
import from MyModule all except {
  type MyType3, MyType5;   // excludes the two types from the import statement
  template all // excludes all templates declared in MyModule from the import statement
}
```

7.5.6 Importing groups

Groups of definitions may be imported.

EXAMPLE 1:

```
import from MyModule {
  group MyGroup
}
```

The effect of importing a group is identical to an **import** statement that lists all importable definitions (including sub-groups) of this group.

TTCN-3 groups are only used for structuring purposes and are not scope units. Therefore, it is allowed to import sub-groups (i.e., a group which is defined within another group) directly, i.e., without the groups in which the sub-group is embedded. If the name of a sub-group that should be imported is identical to the name of another sub-group in the same module (see 7.3.1), the dot notation shall be used to identify the sub-group to be imported uniquely.

If some definitions of a group are wished not to be imported, their kinds and identifiers shall be listed in the exception list within a pair of curly brackets following the **except** keyword.

EXAMPLE 2:

```
import from MyModule {
  group MyGroup except {
      type MyType3, MyType5
           // excludes type definitions MyType3 and MyType5 from the import statement
           // but imports all other definitions of MyGroup
  }
}
```

The **all** keyword is also allowed to be used in the exception list; this will exclude all definitions of the same kind from the import statement.

EXAMPLE 3:

```
import from MyModule {
  group MyGroup except {
      type MyType3, MyType5;  // excludes the two types from the import statement and
      template all // excludes all templates defined in MyGroup from the import statement
  }
}
```

7.5.7 Importing definitions of the same kind

The **all** keyword may be used to import all definitions of the same kind of a module. The **all** keyword used with the **constant** keyword identifies all constants, as well as all external constants declared in the definitions part of the module that the import statement refers to. Similarly, the **all** keyword used with the **function** keyword identifies all functions and all external functions defined in the module the import statement denotes.

EXAMPLE 1:

```
import from MyModule {
  type all;           // imports all types of MyModule
  template all        // imports all templates of MyModule
}
```

If some declarations of a kind are wished to be excluded from the given import statement, their identifiers shall be listed following the **except** keyword.

EXAMPLE 2:

```
import from MyModule {
  type all except MyType3, MyType5;     // imports all types except MyType3 and MyType5
  template all                          // imports all templates defined in Mymodule
}
```

7.5.8 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. 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 (but may) be present when the imported definitions are used. When the definition is referenced in the same module where it is defined, the module identifier of the module (the current module) also may be used for prefixing the identifier of the definition.

EXAMPLE:

```
module MyModuleA  {
    :
  type bitstring MyTypeA;
  import from SomeModuleC {
      type MyTypeA,     // Where MyTypeA is of type character string
           MyTypeB      // 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 that is already defined locally, even with the same name, would lead to two different types being available in the module.

7.5.9 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. Such cases shall be resolved by the system and definitions 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 this Recommendation and should be provided by TTCN-3 tools.

All **`import`** statements and definitions within import statements are considered to be treated independently one after the other in the order of their appearance. It is important to point out that the **`except`** statement does not exclude the definitions listed from being imported in general; all statements importing definitions of the same kind can be seen as a shorthand notation for an equivalent list of identifiers of single definitions. The **`except`** statement excludes definitions from this single list only.

EXAMPLE:

```
import from MyModule {
    type all except MyType3;     // imports all types of MyModule except MyType3
    type MyType3                 // imports MyType3 explicitly
}
```

7.5.10 Import definitions from non-TTCN-3 modules

In cases when definitions are imported from other sources than TTCN-3 modules, the language specification shall be used to denote the language (may be together with a version number) of the source (e.g., module, package, library or even file) from which definitions are imported. It consists of the **`language`** keyword and a subsequent textual declaration of the denoted language. The use of the language specification is optional when importing from a TTCN-3 module of the same edition as the importing module. When incompatibility is discovered between the language identification (including implicit identification by omitting the language specification) and the syntax of the module from which definitions are imported, tools shall provide reasonable effort to resolve the conflict.

The following TTCN-3 language identifiers are defined:

- 'TTCN-3:2001' – to be used with modules complying with the 2001 version of this Recommendation (see the Bibliography).
- 'TTCN-3:2003' – to be used with modules complying with the 2003 version of this Recommendation (see the Bibliography).
- 'TTCN-3:2005' – to be used with modules complying with this Recommendation.

EXAMPLE:

```
import from MyModule language "TTCN-3:2003" {
  type MyType
}
```

NOTE – The import mechanism is designed to allow the reuse of definitions from other TTCN-3 or other language modules. The rules for importing definitions from specifications written in other languages, e.g., SDL packages, may follow the TTCN-3 rules or may have to be defined separately.

8 Test configurations

8.0 General

TTCN-3 allows the (dynamic) specification of concurrent test configurations (or configurations 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.

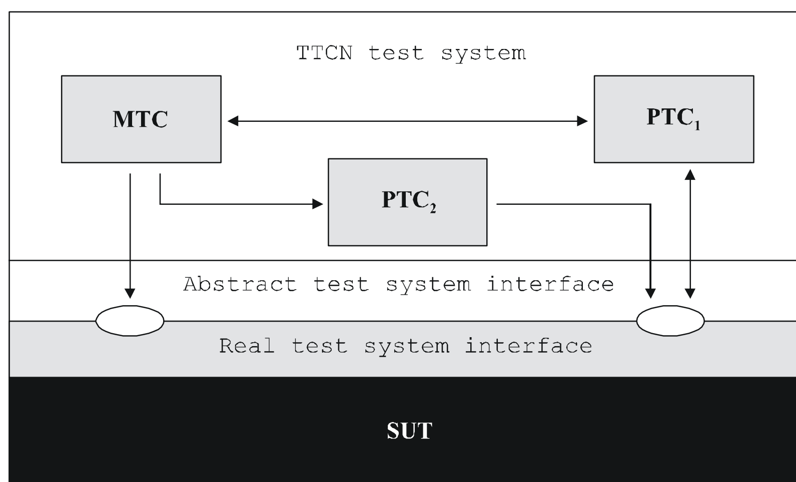


Figure 3/Z.140 – Conceptual view of a typical TTCN-3 test configuration

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 by the system 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. When the MTC terminates, the test system has to stop all PTCs not terminated by the moment when the test case execution is ended.

Communication between test components and between the components and the test system interface is achieved via communication ports (see 8.1).

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 22.2).

8.1 Port communication model

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 25.2.1).

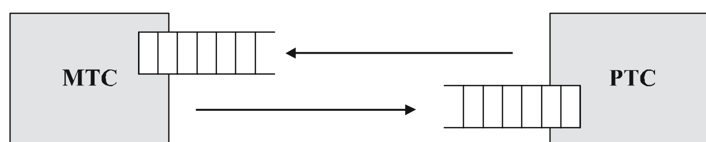


Figure 4/Z.140 – The TTCN-3 communication port model

8.2 Restrictions on connections

TTCN-3 connections are port-to-port and port-to-test system interface connections (see Figure 5). There are no restrictions on the number of connections a component may maintain. One-to-many connections are also allowed (e.g., Figure 5(g) or Figure 5(h)).

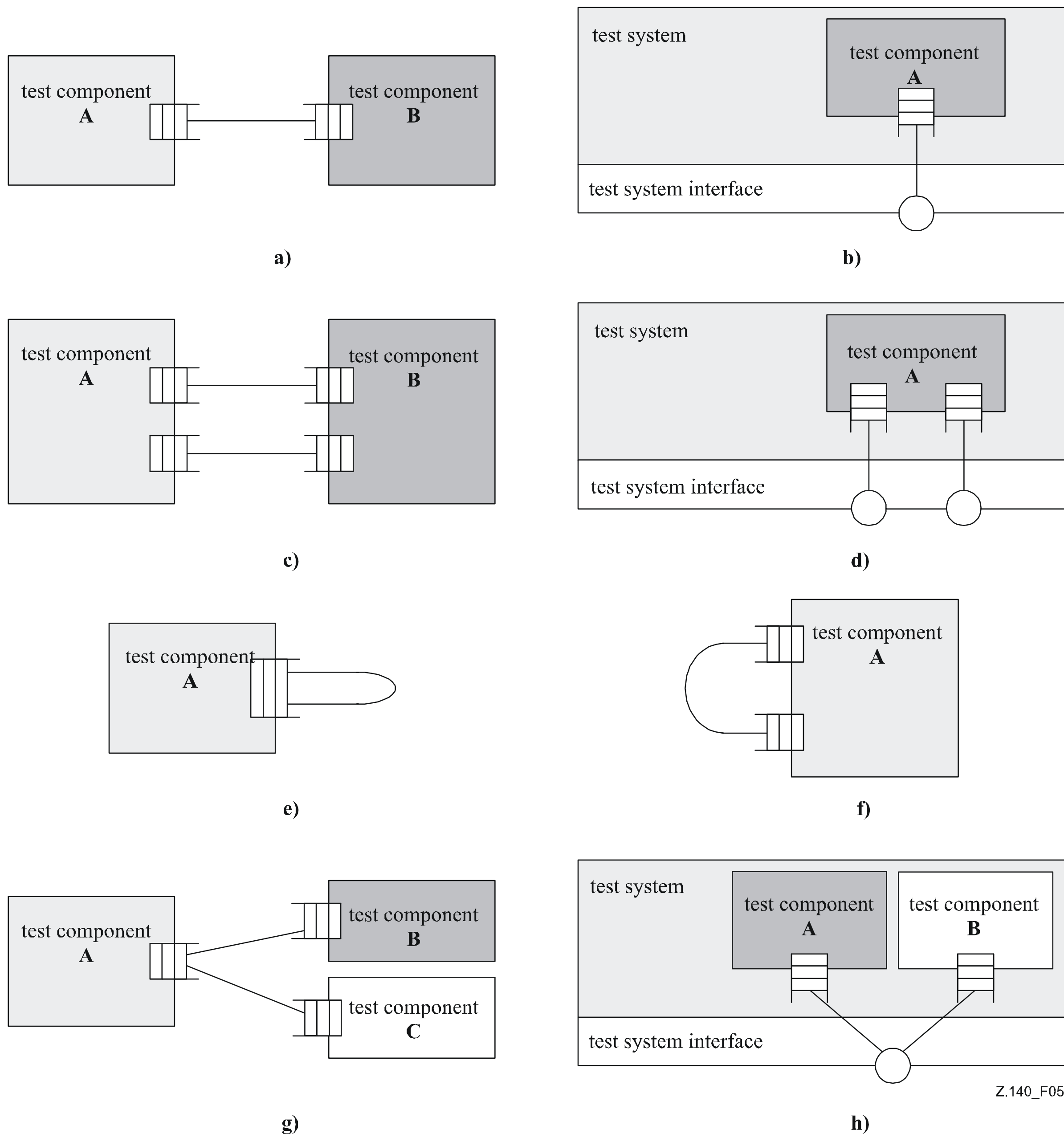


Figure 5/Z.140 – Allowed connections

The following connections are not allowed:

- A port owned by a component A shall not be connected with two or more ports owned by the same component (Figures 6(a) and 6(e)).
- A port owned by a component A shall not be connected with two or more ports owned by a component B (see Figure 6(c)).
- A port owned by a component A can only have a one-to-one connection with the test system interface. This means, connections as shown in Figures 6(b) and 6(d) are not allowed.
- Connections within the test system interface are not allowed (see Figure 6(f)).
- A port that is connected shall not be mapped and a port that is mapped shall not be connected (see Figure 6(g)).

Since TTCN-3 allows dynamic configurations and addresses, the restrictions on connections cannot always be checked at compile-time. The checks shall be made at run-time and shall lead to a test case error when failing.

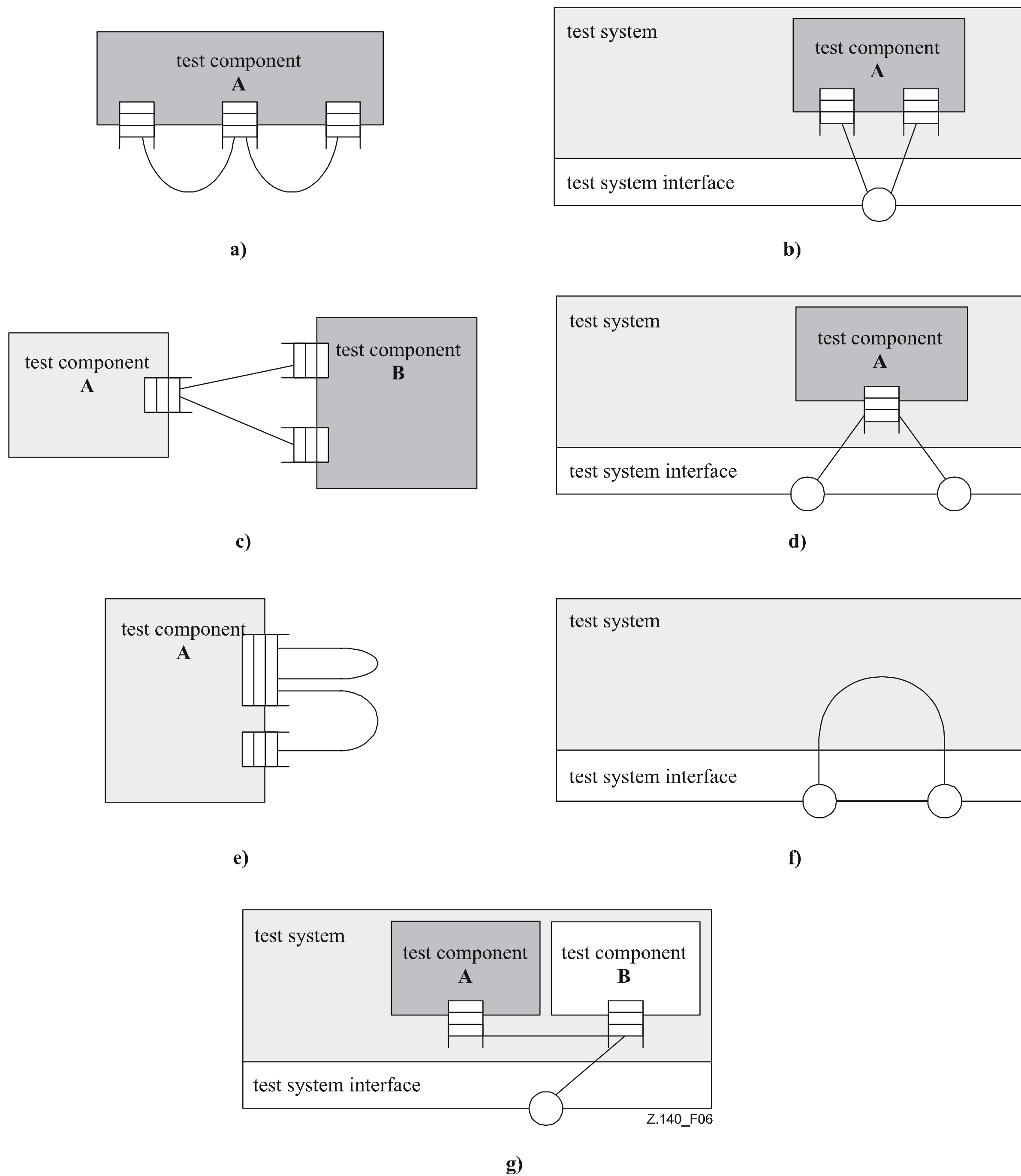


Figure 6/Z.140 – Not allowed connections

8.3 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 this Recommendation 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 shall be 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.

The test system interface statically defines the number and type of the port connections to the SUT during a test run. However, the connections between the test system interface and the TTCN-3 test components are dynamic in nature and may be modified during a test run by using **map** and **unmap** operations (see 22.2 and 22.3).

8.4 Defining communication port types

8.4.0 General

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 message-based or procedure-based (or both at the same time as described in 8.4.1). Message-based ports shall be identified by the keyword **message** and procedure-based ports shall be identified by the keyword **procedure** within the associated port type definition.

Ports are bidirectional. 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.

Whenever a signature (see also clause 13) is defined in the "out" direction of a procedure-based port, the types of all its **inout** and **out** parameters, its return type and its exception types are automatically part of the "in" direction of this port. Whenever a signature is defined in the "in" direction for a procedure-based port, the types of all its **inout** and **out** parameters, its return type and its exception types are automatically part of the "out" direction of this port.

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 procedures 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.

8.4.1 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;
}
```

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 23.5) 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.

8.5 Defining component types

8.5.0 General

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. Definition of a component alone does not mean that there is any connection between the components over these ports.

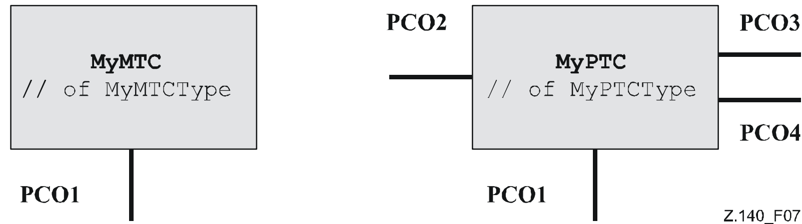


Figure 7/Z.140 – Typical components

EXAMPLE:

```
type component MyMTCType
{
  port MyMessagePortType   PCO1
}

type component MyPTCType
{
  port MyMessagePortType     PCO1, PCO4;
  port MyProcedurePortType PCO2;
  port MyAllMesssagesPortType PCO3
}
```

8.5.1 Declaring local variables, constants and timers in a component

It is possible to declare constants, variables and timers local to a particular component.

EXAMPLE:

```
type component MyMTCType
{
  var integer MyLocalInteger;
  timer MyLocalTimer;
  port MyMessagePortType   PCO1
}
```

These declarations are visible to all testcases, functions and altsteps 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 5.3. 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).

NOTE – When used as test system interfaces (see 8.8), components cannot make use of any constants, variables and timers declared in the component.

8.5.2 Defining components with arrays of ports

It is possible to define arrays of ports in component type definitions (also see 22.12).

EXAMPLE:

```
type component My3pcoCompType
{
  port MyMessageInterfaceType PCO[3]
  port MyProcedureInterfaceType PCOm[3][3]
  // Defines a component type which has an array of 3 message ports and a two-dimensional
  // array of 9 procedure ports.
}
```

8.5.3 Extension of component types

It is possible to define component types as the extension of other component types, using the **extends** keyword.

EXAMPLE 1:

```
type component MyExtendedMTCType extends MyMTCType
{
  var float MyLocalFloat;
  timer MyOtherLocalTimer;
  port MyMessagePortType PCO2;
}
```

In such a definition, the new type definition is referred to as the *extended type*, and the type definition following the **extends** keyword is referred to as the *parent type*.

The effect of this definition is that the extended type will implicitly also contain all definitions from the parent type. So, the definition above is equivalent to writing (and hence called the *effective type definition*):

EXAMPLE 2:

```
// effectively, the definition from Example 1 is equivalent to this one:
type component MyExtendedMTCType
{
  /* the definitions from MyMTCType */
  var integer MyLocalInteger;
  timer MyLocalTimer;
  port MyMessagePortType PCO1

  /* the additional definitions */
  var float MyLocalFloat;
  timer MyOtherLocalTimer;
  port MyMessagePortType PCO2;
}
```

It is allowed to extend component types that are defined by means of extension, as long as no cyclic chain of definition is created.

EXAMPLE 3:

```
type component MTCTypeA extends MTCTypeB { /* … */ };
type component MTCTypeB extends MTCTypeC { /* … */ };
type component MTCTypeC extends MTCTypeA { /* … */ };  // ERROR - cyclic extension
type component MTCTypeD extends MTCTypeD { /* … */ };  // ERROR - cyclic extension
```

When defining component types by extension, there shall be no name clash between the definitions being taken from the parent type and the definitions being added in the extended type; i.e., there shall not be a port, variable, constant, timer, or template identifier that is declared both in the parent type (directly or by means of extension) and the extended type.

EXAMPLE 4:

```
type component MyExtendedMTCType extends MyMTCType
{
  var integer MyLocalInteger; // ERROR - already defined in MyMTCType (see example 2)
  var float MyLocalTimer;     // ERROR - timer with that name exists in MyMTCType
  port MyOtherMessagePortType PCO1; // ERROR - port with that name exists in MyMTCType
}

type component MyBaseComponent { timer MyLocalTimer };
type component MyInterimComponent extends MyBaseComponent { timer MyOtherTimer };
type component MyExtendedComponent extends MyInterimComponent
{
  timer MyLocalTimer; // ERROR - already defined in MyInterimComponent via extension

   }
```

It is allowed to have one component type extending several parent types in one definition, which have to be specified as a comma-separated list of types in the definition:

EXAMPLE 5:

```
type component MyCompA extends MyCompB, MyCompC, MyCompD {
  /* additional definitions for MyCompA */
}
```

Any of the parent types may also be defined by means of extension.

The effective component type definition of the extended type is obtained as the collection of all constant, variable, timer, port and template definitions contributed by the parent types (determined recursively if a parent type is also defined by means of an extension) and the definitions declared in the extended type directly. The effective component

type definition shall be name clash free. To fulfil this condition, within the set of parent types used in the definition of the extended type, all definitions shall have unique names and these names shall differ from any of the names of the definitions declared directly in the extended type.

NOTE 1 – It is not considered to be a different declaration and hence causes no error if the same definition is contributed to the extended type by different parent types (via different extension paths).

EXAMPLE 6:

```
type component MyCompB { timer T };
type component MyCompC { var integer T };
type component MyCompD extends MyCompB, MyCompC {}
  // ERROR - name clash between MyCompB and MyCompC

// MyCompB is defined above
type component MyCompE extends MyCompB {
  var integer MyVar1 := 10;
}

type component MyCompF extends MyCompB {
  var float MyVar2 := 1.0;
}

type component MyCompG extends MyCompB, MyCompE, MyCompF {
  // No name clash.
  // All three parent types of MyCompG have a timer T, either directly or via extension of
  // MyCompB; as all these stem (directly or via extension) from timer T declared in MyCompB,
  // which make this form of collision legal.
  /* additional definitions here */
}
```

The semantics of component types with extensions are defined by simply replacing each component type definition by its effective component type definition as a pre-processing step prior to using it.

NOTE 2 – For component type compatibility, this means that a component reference c of type CT1, which extends CT2, is compatible with CT2, and test cases, functions and altsteps specifying CT2 in their **runs on** clauses can be executed on c (see 6.7.3).

8.6 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. When used with **to**, **from** and **sender** the address data type shall only be used in receive and send operations of ports mapped to the test system interface. 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;
```

8.7 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 22.1). In addition, component references are returned by the predefined operations **`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 22) to set up test configurations and in the **`from`**, **`to`** and **`sender`** parts of communication operations of ports connected to test components other than the test system **`interface`** for addressing purposes (see clause 23 and Figure 5).

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 must use the corresponding component type name in its declaration.

EXAMPLE:

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

// Declaring one 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
// always 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 MyInst1 -> value MyMessage sender MyInst1 {}
  [] PCO1.receive(M1:?) from MyInst2 -> value MyMessage sender MyInst2 {}
  [] PCO1.receive(M1:?) from MyInst3 -> 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};
 :
```

8.8 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).

NOTE – Variables, timers and constants declared in component types, which are used as test system interfaces, will have no effect.

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

Generally, a component type reference defining the test system interface shall be associated with every test case using more than one test component. The ports of the test system interface shall automatically be instantiated by the system together with the MTC when the test case execution starts.

The operation returning the component reference of the test system interface is **system**. This shall be used to address the ports of the test system.

EXAMPLE:

```
map(MyMTCComponent: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 the module definitions part, component type definitions, the module control part, test cases, functions and altsteps. Constant definitions are denoted by the keyword **const**. Constants shall not be of port type. The value of the constant shall be assigned at the point of declaration.

NOTE – The only value that can be assigned to constants of default and component types is the special value **null**.

EXAMPLE 1:

```
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**.

EXAMPLE 2:

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

An external constant may have an arbitrary type, except for port type, default type, or component type, and the type has to be known in the module; i.e., shall be a root type or a user-defined type defined in the module, or imported from another module. The mapping of the type to the external representation of an external constant and the mechanism of how the value of an external constant is passed into a module are outside the scope of this Recommendation.

10 Declaring variables

10.0 General

Variables can be of simple basic types, basic string types, structured types, special data types (including subtypes derived from these types) as well as address, component or default types.

NOTE – Structured and component type variables can be declared based on user-defined types only.

Variables can be declared and used in the module control part, test cases, functions and altsteps. Additionally, variables can be declared in component type definitions. These variables can be used in test cases, altsteps and functions which are running on the given component type. Variables shall not be declared or used in a module definitions part (i.e., global variables are not supported in TTCN-3).

Use of uninitialized or not completely initialized variables at other places than the left-hand side of assignments or as actual parameters passed to out formal parameters shall cause an error.

10.1 Value variables

A value variable is declared by the **var** keyword followed by a type identifier and a variable identifier. An initial value can be assigned at declaration. Value variables shall store values only and may be used at the right-hand side as well as at the left-hand side of assignments, in expressions, following the **return** keyword in bodies of functions with a return clause in their headers, and may be passed to both value and template-type formal parameters.

EXAMPLE:

```
var integer MyVar0;
var integer MyVar1 := 1;
var boolean MyVar2 := true, MyVar3 := false;
```

10.2 Template variables

Template variables are declared by the **var template** keyword followed by a type identifier and a variable identifier. An initial content can be assigned at declaration. In excess to values, template variables may also store matching mechanisms (see 14.3). They may be used on the right-hand side as well as on the left-hand side of assignments, following the **return** keyword in bodies of functions defining a template-type return value in their headers, and may be passed as actual parameters to template-type formal parameters. When used on the right-hand side of assignments they shall not be operands of TTCN-3 operators (see clause 15) and the variable on the left-hand side shall be a template variable too. It is also allowed to assign a template instance to a template variable or a template variable field.

NOTE – Template variables, similarly to global and local templates, shall be fully specified in order to be used in sending and receiving operations.

EXAMPLE:

```
template MyRecord MyTempl ( template boolean par_bool ) :=
  { field1 := par_bool, field2 := * }
:
function Myfunc () return template MyRecord {
  var template integer MyVarTemp1 := ?;
  var template MyRecord MyVarTemp2 := { field1 := true, field2 := * },
                MyVarTemp3 := { field1 := ?, field2 := MyVarTemp1 };
  MyVarTemp2 := MyTempl (?);
:
  return MyVarTemp2
}
```

While it is not allowed to directly apply TTCN-3 operations to template variables, it is allowed to use the dot notation and the index notation to inspect and modify template variable fields. Rules to apply when these notations attempt to reach fields beyond a matching mechanism are given in 14.3.1.

11 Declaring timers

11.0 General

Timers can be declared and used in the module control part, test cases, functions and altsteps. Additionally, timers can be declared in component type definitions. These timers can be used in test cases, functions and altsteps which are running on the given component type. 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 a non-negative **float** value (i.e., greater than or equal to 0.0) where the base unit is seconds.

EXAMPLE 1:

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

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

In addition to single timer instances, timer arrays can also be declared. Default duration(s) of the elements of a timer array shall be assigned using a value array. Default duration(s) assignment shall use the array value notation as specified in 6.5. If the default duration assignment is wished to be skipped for some element(s) of the timer array, it shall explicitly be declared by using the not used symbol ("-").

EXAMPLE 2:

```
timer t_Mytimer1[5] := { 1.0, 2.0, 3.0, 4.0, 5.0 }
  // all elements of the timer array get a default duration.

timer t_Mytimer2[5] := { 1.0, -, 3.0, 4.0, 5.0 }
  // the second timer (t_Mytimer2[1]) is left without a default duration.
```

11.1 Timers as parameters

Timers can only be passed by reference to functions and altsteps. Timers passed into a function or altstep are known inside the behaviour definition of the function or altstep.

Timers passed in as parameters by reference can be used like any other timer; i.e., they need not to be declared. A started timer can also be passed into a function or altstep. The timer continues to run; i.e., it is not stopped implicitly. Thereby, possible timeout events can be handled inside the function or altstep 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 coordination 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 coordination 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.

13 Declaring procedure signatures

13.0 General

Procedure signatures (or signatures for short) are needed for procedure-based communication. Procedure-based communication may be used for the communication within the test system, i.e., among test components, or for the communication between the test system and the SUT. In the latter case, a procedure may either be invoked in the SUT (i.e., the test system performs the call) or invoked in the test system (i.e., the SUT performs the call). For all used procedures, i.e., procedures used for the communication among test components, procedures called from the SUT and procedures called from the test system, complete procedure **signature** shall be defined in the TTCN-3 module.

13.1 Signatures for blocking and non-blocking communication

TTCN-3 supports *blocking* and *non-blocking* procedure-based communication. Signature definitions for non-blocking communication shall use the **noblock** keyword, shall only have **in** parameters (see 13.2) and shall have no return value (see 13.3), but may raise exceptions (see 13.4). By default, signature definitions without the **noblock** keyword are assumed to be used for blocking procedure-based communication.

EXAMPLE:

```
signature MyRemoteProcOne ();           // MyRemoteProcOne will be used for blocking
                                        // procedure-based communication. It has neither
                                        // parameters nor a return value.

signature MyRemoteProcTwo () noblock;   // MyRemoteProcTwo will be used for non blocking
                                        // procedure-based communication. It has neither
                                        // parameters nor a return value.
```

13.2 Parameters of procedure signatures

Signature definitions may have parameters. Within a **signature** definition the parameter list may include parameter identifiers, parameter types and their direction, i.e., **in**, **out**, or **inout**. The direction **inout** and **out** indicate that these parameters are used to retrieve information from the remote procedure. Note that the direction of the parameters is as seen by the *called* party rather than the *calling* party.

EXAMPLE:

```
signature MyRemoteProcThree (in integer Par1, out float Par2, inout integer Par3);
// MyRemoteProcThree will be used for blocking procedure-based communication. The procedure
// has three parameters: Par1 an in parameter of type integer, Par2 an out parameter of
// type float and Par3 an inout parameter of type integer.
```

13.3 Value returning remote procedures

A remote procedure may return a value after its termination. The type of the return value shall be specified by means of a **return** clause in the corresponding signature definition.

EXAMPLE:

```
signature MyRemoteProcFour (in integer Par1) return integer;
// MyRemoteProcFour will be used for blocking procedure-based communication. The procedure
// has the in parameter Par1 of type integer and returns a value of type integer after its
// termination
```

13.4 Specifying exceptions

Exceptions that may be raised by remote procedures are represented in TTCN-3 as values of a specific type. Therefore, templates and matching mechanisms can be used to specify or to check return values of remote procedures.

NOTE – The conversion of exceptions generated by, or sent to, the SUT into the corresponding TTCN-3 type or SUT representation is tool- and system-specific and is, therefore, beyond the scope of this Recommendation.

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 specific values of these types).

EXAMPLE:

```
signature MyRemoteProcFive (inout float Par1) return integer
      exception (ExceptionType1, ExceptionType2);
// MyRemoteProcFive will be used for blocking procedure-based communication. It returns a
// float value in the inout parameter Par1 and an integer value, or may raise exceptions of
// type ExceptionType1 or ExceptionType2

signature MyRemoteProcSix (in integer Par1) noblock
      exception (integer, float);
// MyRemoteProcSix will be used for non-blocking procedure-based communication. In case of
// an unsuccessful termination, MyRemoteProcSix raises exceptions of type integer or float.
```

14 Declaring templates

14.0 General

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 can be defined globally in a module definitions part, locally in a testcase, function, altstep or statement block or in-line as arguments of a communication operation or actual parameter of a testcase, function or altstep call.

Templates provide the following possibilities:

a) they are a way to organize and to reuse 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.

A template declaration must specify a set of base values or matching symbols for each and every field defined in the appropriate type or signature definition; i.e., it is fully specified. A modified template declaration (see 14.6) specifies only the fields to be changed from the base template; i.e., it is a partial specification. The NotUsedSymbol shall only be used in signature templates for parameters which are not relevant and in modified template declarations and modified in-line templates to indicate no change for the specified field or element.

There exist a number of restrictions on the functions used in expressions when specifying templates or template fields; these are specified in 16.1.4.

14.1 Declaring message templates

14.1.0 General

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 **`port`** and **`default`** types.

EXAMPLE:

```
// 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);
```

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.

NOTE – For sending templates, omitting an optional field is considered to be a value notation rather than a matching mechanism.

EXAMPLE:

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

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

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

14.1.2 Templates for receiving messages

A template used in a **`receive`**, **`trigger`** or **`check`** operation defines a data template against which an incoming message is to be matched. Matching mechanisms, as defined in Annex B, 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 optional,
  charstring  field2,
  boolean  field3
}

// a message template might be
template MyMessageType MyTemplate:=
```

```
{
  field1 := ?,
  field2 := pattern "abc*xyz",
  field3 := true
}

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

14.2 Declaring signature templates

14.2.0 General

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. A signature template defines the values and matching mechanisms of the procedure parameters only, but not the return value. The values or matching mechanisms for a return has to be defined within the **`reply`** or **`getreply`** operation (see 23.3.3 and 23.3.4, respectively).

EXAMPLE:

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

// example templates associated 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 invoking 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 B).

EXAMPLE:

```
// Given the examples in clause 14.2.0

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

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

// Invalid invocation because the inout parameter Par3 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 operation must be retrieved using an assignment clause at the end of the call statement
```

14.2.2 Templates for accepting procedure invocations

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 B, may be used in any templates used by this operation. No binding of incoming values to the template shall occur. Any **`out`** parameters shall be ignored in the matching process.

EXAMPLE:

```
// Given the examples in clause 14.2.0

// Valid getcall, it will match if Par1 == 1 and Par3 == 3
MyPCO.getcall(RemoteProc:Template1);

// Valid getcall, it will match if Par1 == 1 and Par3 == 3
MyPCO.getcall(RemoteProc:Template2);

// Valid getcall, it will match on Par1 == 1 and Any value of Par3
MyPCO.getcall(RemoteProc:Template3);
```

14.3 Template matching mechanisms

14.3.0 General

Generally, matching mechanisms are 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, trigger, getcall`**, **`getreply`** and **`catch`** operations). They may appear in explicit values.

EXAMPLE 1:

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

The type identifier may be omitted when the value unambiguously identifies the type.

EXAMPLE 2:

```
MyPCO.receive("AAAA"O);
```

NOTE – The following types may be omitted: integer, float, Boolean, bitstring, hexstring, octetstring.

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 between the listed type and the type of the value provided (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:
   - an expression that evaluates to a specific value;
   - **omit**: value is omitted;
b) special symbols that can be used *instead* of values:
   - **(…)**: a list of values;
   - **complement (…)**: complement of a list of **values**;
   - **?**: wildcard for any value;
   - *****: wildcard for any value or no **value** at all (i.e., an omitted value);
   - **(***lowerBound* **`..`** *upperBound***)**: a range of **integer** or float values between and including the lower- and upper bounds;
   - **superset:** at least all of the elements listed, i.e., possibly more;
   - **subset:** at most the elements listed, i.e., possibly less;
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);
   - **permutation**: all of the elements listed but in an arbitrary order (note, that ? and * are also allowed as elements of the permutation list);

d) special symbols which describe *attributes* of values:
   - **length**: restrictions for string length for string types and the number of elements for **`record of`**, **`set of`** 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 6. The left-hand column of this table lists all the TTCN-3 types to which these matching mechanisms apply. A full description of each matching mechanism can be found in Annex B.

Table 6/Z.140 – TTCN-3 matching mechanisms

Used with values of	Value		Instead of values							Inside values			Attributes	
	Specific value	Omit value	Complemented list	Value list	Any value(?)	Any value Or None(*)	Range	Superset	Subset	Any element(?)	Any elements Or None(*)	Per mutation	Length restriction	If present
`boolean`	Yes	Yes	Yes	Yes	Yes	Yes[a]								Yes[b]
`integer`	Yes	Yes	Yes	Yes	Yes	Yes[a]	Yes							Yes[b]
`float`	Yes	Yes	Yes	Yes	Yes	Yes[a]	Yes							Yes[b]
`bitstring`	Yes	Yes	Yes	Yes	Yes	Yes[a]				Yes	Yes		Yes	Yes[b]
`octetstring`	Yes	Yes	Yes	Yes	Yes	Yes[a]				Yes	Yes		Yes	Yes[b]
`hexstring`	Yes	Yes	Yes	Yes	Yes	Yes[a]				Yes	Yes		Yes	Yes[b]
`character strings`	Yes	Yes	Yes	Yes	Yes	Yes[a]	Yes			Yes	Yes		Yes	Yes[b]
`record`	Yes	Yes	Yes	Yes	Yes	Yes[a]								Yes[b]
`record of`	Yes	Yes	Yes	Yes	Yes	Yes[a]				Yes	Yes	Yes	Yes	Yes[b]
`array`	Yes	Yes	Yes	Yes	Yes	Yes[a]				Yes	Yes		Yes	Yes[b]
`set`	Yes	Yes	Yes	Yes	Yes	Yes[a]								Yes[b]
`set of`	Yes	Yes	Yes	Yes	Yes	Yes		Yes	Yes	Yes	Yes		Yes	Yes[b]
`enumerated`	Yes	Yes	Yes	Yes	Yes	Yes[a]								Yes[b]
`union`	Yes	Yes	Yes	Yes	Yes	Yes[a]								Yes[b]
`anytype`	Yes	Yes	Yes	Yes	Yes	Yes[a]								Yes[b]

[a] – When used, shall be applied to optional fields of record and set types only (without restriction on the type of that field).

[b] – When used, shall be applied to record and set fields only (without restriction on the type of that field).

14.3.1 Referencing elements of templates or template fields

14.3.1.1 Referencing individual string elements

It is not allowed to reference individual string elements inside templates or template fields.

EXAMPLE:

```
var template charstring t_Char1 := 'MYCHAR';
var template charstring t_Char2;

t_Char2 := t_Char1[1];
  // shall cause an error as referencing individual string elements is not allowed;
```

14.3.1.2 Referencing **`record`** and **`set`** fields

Both templates and template variables allow referencing sub-fields inside a template definition using the dot notation. However, the referenced field may be a subfield of a structured field to which a matching mechanism is assigned. This clause provides rules for such cases.

- Omit, AnyValueOrNone, value lists and complemented lists: referencing a subfield within a structured field to which **`omit`**, AnyValueOrNone (*), a value list or a complemented list is assigned, shall cause an error.

EXAMPLE 1:

```
type record R1 {
  integer f1 optional,
  R2      f2 optional
}
type record R2 {
  integer g1,
  R2      g2 optional
}

:
var template R1 t_R1 := {
  f1 := 5,
  f2 := omit
}
var template R2 t_R2 := t_R1.f2.g2;
  // causes an error as omit is assigned to t_R1.f2
t_R1. f2 := *
t_R2 := t_R1.f2.g2;
  // causes an error as * is assigned to t_R1.f2

t_R1 := ({f1:=omit, f2:={g1:=0, g2:=omit}},{f1:=5, f2:={g1:=1, g2:={g1:=2, g2:=omit}}});

t_R2 := t_R1.f2;
t_R2 := t_R1.f2.g2;
t_R2 := t_R1.f2.g2.g2;
  // all these assignments cause error as a value list is assigned to t_R1

t_R1 :=
  complement({f1:=omit, f2:={g1:=0, g2:=omit}},{f1:=5, f2:={g1:=1, g2:={g1:=2, g2:=omit}}})

t_R2 := t_R1.f2;
t_R2 := t_R1.f2.g2;
t_R2 := t_R1.f2.g2.g2;
  // all these assignments cause error as a complemented list is assigned to t_R1
```

- AnyValue: when referencing a subfield within a structured field to which AnyValue (?) is assigned, at the right-hand side of an assignment, AnyValue (?) shall be returned for mandatory subfields and AnyValueOrNone shall be returned for optional subfields.

  When referencing a subfield within a structured field to which AnyValue (?) is assigned, at the left-hand side of an assignment, the structured field has to be expanded recursively up to the depth of the referenced subfield. During this expansion an AnyValue (?) shall be assigned to mandatory subfields and AnyValueOrNone shall be assigned to optional subfields. After this expansion the value or matching mechanism at the right-hand side of the assignment shall be assigned to the referenced subfield.

EXAMPLE 2:

```
t_R1 := {f1:=0, f2:=?}
t_R2 := t_R1.f2.g2;
  // after the assignment t_R2 will be {g1:=?, g2:=*}
t_R1.f2.g2.g2 := ({g1:=1, g2:=omit},{g1:=2, g2:=omit});
  // first the field t_R1.f2 has hypothetically be expanded to {g1:=?,g2:={g1:=?,g2:=*}}
  // thus after the assignment t_R1 will be:
  //   {f1:=0, f2:={g1:=?,g2:={g1:=?,g2:=({g1:=1, g2:=omit},{g1:=2, g2:=omit})}}}
```

- Ifpresent attribute: referencing a subfield within a structured field to which the **`ifpresent`** attribute is attached, shall cause an error (irrespective the value or the matching mechanism to which **`ifpresent`** is appended).

14.3.1.3 Referencing `record of` and `set of` elements

Both templates and template variables allow referencing elements of a **`record of`** or **`set of`** template or field using the index notation. However, a matching mechanism may be assigned to the template or field within which the element is referenced. This clause provides rules on handling such cases.

- Omit, AnyValueOrNone, value lists, complemented lists, subset and superset: referencing an element within a record of or set of field to which **`omit`**, AnyValueOrNone (*) with or without a length attribute, a value list, a complemented list, a subset or a superset is assigned, shall cause an error.

EXAMPLE 1:

```
type record of integer RoI;
type record of RoI RoRoI;

:
var template RoI t_RoI;
var template RoRoI t_RoRoI;
var template integer t_Int;
:
t_RoRoI := ({},{0},{0,0},{0,0,0});
t_RoI := t_RoRoI[0];
  // shall cause an error as value list is assigned to t_RoRoI;
```

- AnyValue: when referencing an element of a **`record of`** or **`set of`** template or field to which AnyValue (?) is assigned (without a length attribute), at the right-hand side of an assignment, AnyValue (?) shall be returned. If a length attribute is attached to the AnyValue (?), the index of the reference shall not violate the length attribute.

  When referencing an element within a **`record of`** or **`set of`** template or field to which AnyValue (?) is assigned (without a length attribute), at the left-hand side of an assignment, the value or matching mechanism at the right-hand side of the assignment shall be assigned to the referenced element, AnyElement(?) shall be assigned to all elements before the referenced one (if any), and a single AnyElementsOrNone(*) shall be added at the end. When a length attribute is attached to AnyValue(?) the attribute shall be conveyed to the new template or field transparently. The index shall not violate type restrictions in any of the above cases.

EXAMPLE 2:

```
type record of integer RoI;
type record of RoI RoRoI;

:
var template RoI t_RoI;
var template RoRoI t_RoRoI;
var template integer t_Int;
:
t_RoI := ?;
t_Int := t_RoI[5];
  // after the assignment t_Int will be AnyValue(?);

t_RoRoI := ?;
t_RoI := t_RoRoI[5];
  // after the assignment t_RoI will be AnyValue(?);
t_Int := t_RoRoI[5].[3];
  // after the assignment t_Int will be AnyValue(?);

t_RoI := ? length (2..5);
t_Int := t_RoI[3];
  // after the assignment t_Int will be AnyValue(?);
t_Int := t_RoI[5];
  // shall cause an error as the referenced index is outside the length attribute
  // (note that index 5 would refer to the 6th element);

t_RoRoI[2] := {0,0};
  // after the assignment t_RoRoI will be {?,?,{0,0},*};
t_RoRoI[4] := {1,1};
  // after the assignment t_RoRoI will be {?,?,{0,0},?,{1,1},*};
t_RoI[0] := -5;
  // after the assignment t_RoI will be {-5,*}length(2..5);
t_RoI := ? length (2..5);
t_RoI[1] := 1;
  // after the assignment t_RoI will be {?,1,*}length(2..5);
t_RoI[3] := ?
  // after the assignment t_RoI will be {?,1,?,?,*}length(2..5);
t_RoI[5] := 5
  // after the assignment t_RoI will be {?,1,?,?,?,5,*}length(2..5); note that t_RoI
  // becomes an empty set but that shall cause no error;
```

- Permutation: when referencing an element of a **`record of`** template or field, which is located inside a permutation (based on its index), this shall cause an error. Indexes of elements sheltered by a permutation shall be determined based on the number of permutation elements. AnyValueOrNone as a permutation element causes the permutation to shelter all **`record of`** element indexes.

EXAMPLE 3:

```
t_RoI := {permutation(0,1,3,?),2,?}
t_Int := t_RoI[5];
  // after the assignment t_Int will be AnyValue(?)

t_RoI := {permutation(0,1,3,?),2,*}
t_Int := t_RoI[5];
  // after the assignment t_Int will be * (AnyValueOrNone)
t_Int := t_RoI[2];
  // causes error as the third element (with index 2) is inside permutation

t_RoI := {permutation(0,1,3,*),2,?}
t_Int := t_RoI[5];
  // causes error as the permutation contains AnyValueOrNone(*) that is able to
  // cover any record of indexes
```

- Ifpresent attribute: referencing an element within a **record of** or **set of** field to which the **ifpresent** attribute is attached, shall cause an error (irrespective of the value or the matching mechanism to which **ifpresent** is appended).

14.4 Parameterization of templates

14.4.0 General

Templates for both sending and receiving operations can be parameterized. The actual parameters of a template can include values and templates, functions and special matching symbols. The rules for formal and actual parameter lists shall be followed as defined in 5.2.

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));
```

14.5 Void

14.6 Modified templates

14.6.0 General

Normally, a template specifies a set of base or default values or matching symbols for each and every field defined in the appropriate type or signature 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 an 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. When the field to be modified is nested within a template field which is a structured field itself, no other field of the structured field is changed apart from the explicitly denoted one(s).

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

EXAMPLE 1:

```
// Given
type record MyRecordType
{
  integer field,
```

```
  charstring field2,
  boolean field3
}
template MyRecordType MyTemplate1 :=
{
  field1 := 123,
  field2 := "A string",
  field3 := true
}
// then writing
template MyRecordType MyTemplate2 modifies MyTemplate1 :=
{
  field1 := omit,          // field1 is optional but present in MyTemplate1
  field2 := "A modified string"
                           // field3 is unchanged
}
// is the same as writing
template MyRecordType MyTemplate2 :=
{
  field1 := omit,
  field2 := "A modified string",
  field3 := true
}
```

When individual values of a modified template or a modified template field of **record of** type wished to be changed, and only in these cases, the value assignment notation may also be used, where the left-hand side of the assignment is the index of the element to be altered.

EXAMPLE 2:

```
template MyRecordOfType MyBaseTemplate := { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
template MyRecordOfType MyModifTemplate modifies MyBaseTemplate := { [2] := 3, [3] := 2 };
// MyModifTemplate shall match the sequence of values { 0, 1, 3, 2, 4, 5, 6, 7, 8, 9 }
```

14.6.1 Parameterization of modified templates

If a base template 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 defined at any of the modification steps between the base template and the actual modified template;

b) a derived template can have additional (appended) parameters if wished;

c) the formal parameter list shall follow the template name for every 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 and remains also parameterized in Template2
  field2 := "A modified string",
}
```

14.6.2 In-line modified templates

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

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

In communication operations (e.g., **send**, **receive**, **call**, **getcall**, etc.), it is allowed to change template fields via parameterization or by in-line derived templates only. 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 dot notation *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 23).

14.8 Match Operation

The **match** operation allows the value of a variable or parameter to be compared with a template. The operation returns a Boolean value. If the types of the template and variable are not compatible (see 6.7) 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.

EXAMPLE:

```
template integer LessThan10 := (-infinity..9);

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

  if( match( RxValue, LessThan10)) { … }
// true if the actual value of Rxvalue is less than 10 and false otherwise
   :
}
```

14.9 Valueof Operation

The **valueof** operation allows the value specified within a template to be assigned to 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.

EXAMPLE:

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

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

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

15 Operators

15.0 General

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 7.

Table 7/Z.140 – 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 8. 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 8/Z.140 – Precedence of operators

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

15.1 Arithmetic operators

The arithmetic operators represent the operations of addition, subtraction, multiplication, division, modulo and remainder. Operands of these operators shall be of type **integer** (including derivations of **integer**) or **float** (including derivations of **float**), except for **mod** and **rem** 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** part of the 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|        when  x >= 0
      = 0                     when  x < 0 and   x rem |y| = 0
      = |y| + x rem |y|       when  x < 0 and   x rem |y| < 0
```

Table 9 illustrates the difference between the **mod** and **rem** operator:

Table 9/Z.140 – 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 string operators perform concatenation of values of compatible string types. The operation is a simple concatenation from left to right. No form of arithmetic addition is implied. The result type is the root type of the operands.

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 (>), non-equality to (!=), greater than or equal to (>=) and less than or equal to (<=). Operands of equality and non-equality may be of arbitrary but compatible types with the exception of the **enumerated** type, in which case operands shall be instances of the same type. All other relational operators shall have only operands of type **integer** (including derivatives of **integer), float** (including derivations of **float**) or instances of the same **enumerated** types. The result type of these operations is **boolean**.

Two **charstring** or **universal charstring** values are equal only if they have equal lengths and the characters at all positions are the same. For values of **bitstring, hexstring** or **octetstring** types, the same equality rule applies with the exception that the elements which shall be equal at all positions are bits, hexadecimal digits or pairs of hexadecimal digits, respectively.

Two **record** values, **set** values, **record of** values or **set of** values are equal if, and only if, their effective value structures are compatible (see 6.7) and the values of all corresponding fields are equal. Record values may also be

compared to record of values and set values to set of values. In these cases the same rule applies as for comparing two **record** or **set** values.

NOTE – "All fields" means that optional fields not present in the actual value of a **record** type shall be taken as an undefined value. Such a field can be equal only to a missing optional field (also considered to be an undefined value) when compared with a value of another **record** type or to an element with undefined value when compared with a value of a **record of** type. This principle also applies when values of two **set** types or a **set** and a **set of** type are compared.

Two values of **union** types are equal if, and only if, in both values the types of the chosen fields are compatible and the actual values of the chosen fields are equal.

EXAMPLE:

```
// Given
type  set  SetA{
       integer    a1   optional,
       integer    a2   optional,
       integer    a3   optional
       };

type  set  SetB{
       integer    b1   optional,
       integer    b2   optional,
       integer    b3   optional
       };

type  set  SetC{
       integer    c1   optional,
       integer    c2   optional,
       };

type  set of integer   SetOf;

type  union UniD  {
       integer    d1,
       integer    d2,
       };

type  union UniE  {
       integer    e1,
       integer    e2,
       };

type  union UniF  {
       integer    f1,
       integer    f2,
       boolean    f3,
       };

// And
const      Set A  conSetA1  := { a1 := 0, a2 := omit, a3 := 2 };
             // Notice that the order of defining values of the fields does not matter
const      SetB   conSetB1  := { b1 := 0, b3 := 2, b2 := omit };
const      SetB   conSetB2  := { b2 := 0, b3 := 2, b1 := omit };
const      SetC   conSetC1  := { c1 := 0, c2 :=2 };
const      SetOf  conSetOf1 := { 0, omit, 2 };
const      SetOf  conSetOf2 := { 0, 2 };
const      UniD   conUniD1  := { d1:= 0 };
const      UniE   conUniE1  := { e1:= 0 };
const      UniE   conUniE2; := { e2:= 0 };
const      UniF   conUniF1; := { f1:= 0 };

// Then
conSetA1 == conSetB1;
  // returns true
conSetA1 == conSetB2;
  // returns false, because neither a1 nor a2 are equal to their counterparts
  // ( the corresponding element is not omitted )
conSetA1 == conSetC1;
  // returns false, because the effective value structures of SetA and SetC are not compatible
conSetA1 == conSetOf1;
  // returns true
conSetA1 == conSetOf2;
  // returns false, as the counterpart of the omitted a2 is 2,
  // but the counterpart of a3 is undefined
conSetC1 == conSetOf2;
  // returns true
conUniD1 == conUniE1;
  // returns true
```

```
conUniD1 == conUniE2;
  // returns false, as the chosen field e2 is not the counterpart of the field d1 of UniD1
conUniD1 == conUniF1;
  // returns false, as the effective value structures of UniD1 and UniF are not compatible
```

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 logical operations 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**.

Short circuit evaluation for Boolean expressions is used; i.e., the evaluation of operands of logical operators is stopped once the overall result is known: in the case of the **and** operator, if the left argument evaluates to **false**, then the right argument is not evaluated and the whole expression evaluates to **false**. In the case of the **or** operator, if the left argument evaluates to **true**, then the right argument is not evaluated and the whole expression evaluates to **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** or **octetstring**. In the case of **and4b**, **or4b** and **xor4b**, the operands shall be of compatible types. The result type of the bitwise operators shall be the root type 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 1:

```
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 of equal length. 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 2:

```
'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 of equal length. 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 3:

```
'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 of equal length. 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 1. 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 4:

```
'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 (**>>**) operations. Their left-hand operand shall be of type **bitstring**, **hexstring** or **octetstring.** 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**, 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.

EXAMPLE 1:

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

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.

EXAMPLE 2:

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

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 1:

```
'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 2:

```
'100001'B  @> 2 gives  '011000'B
'12345'H   @> 2 gives  '45123'H
'1122334455'O  @> (1+2)  gives  '3344551122'O
"abcdefg" @> 3 gives "efgabcd"
```

16 Functions and altsteps

In TTCN-3, functions and altsteps are used to specify and structure test behaviour, define default behaviour and to structure computation in a module, etc., as described in the following clauses.

16.1 Functions

16.1.0 General

Functions are used in TTCN-3 to express test behaviour, to organize test execution 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 or a template. Value return is denoted by the **return** keyword followed by a type identifier. Template return is denoted by the **return template** keywords followed by a type identifier.

The keyword **return**, when used in the body of the function with a value return defined in its header, shall always be followed by an expression representing the return value. The type of the return value shall be compatible with the return type. The keyword **return**, when used in the body of the function with a template return defined in its header, shall always be followed by an expression or a template instance representing the return template. The type of the return template shall be compatible with the return template type.

The return statement in the body of the function causes the function to terminate and to return the return value to the location of the call of the function.

EXAMPLE 1:

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

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

// Definition of functions which may return matching symbols or templates
function MyFunction2() return template integer
{
:
  return ?;    // returns the matching mechanism AnyValue
}
function MyFunction3() return template octetstring
{
:
  return "FF??FF"O;    // returns an octetstring with AnyElement inside it
}
```

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 this Recommendation. External functions are not allowed to contain port operations. External functions are not allowed to return templates.

```
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 described in clause 18. If a function uses variables, constants, timers and ports that are declared in a component type definition, the component type shall be referenced using the **runs on** keywords in the function header. The one exception to this rule is when all the necessary component-wide information is passed in the function as parameters.

EXAMPLE 2:

```
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 referencing a component type
```

A function without **runs on** clause shall never invoke a function or altstep or activate an altstep as default with a **runs on** clause locally.

Functions started by using the **start** test component operation shall always have a **runs on** clause (see 22.5) and are considered to be invoked in the component to be started, i.e., not locally. However, the **start** test component operation may be invoked in functions without a **runs on** clause.

NOTE – The restrictions concerning the **runs on** clause are only related to functions and altsteps and not to test cases.

Functions used in the control part of a TTCN-3 module shall have no **runs on** clause. Nevertheless, they are allowed to execute test cases.

16.1.1 Parameterization of functions

Functions may be parameterized. The rules for formal parameter lists shall be followed as defined in 5.2.

EXAMPLE:

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

16.1.2 Invoking functions

A function is invoked by referring to its name and providing the actual list of parameters. Functions that do not return values shall be invoked directly. Functions that return values may be invoked directly or inside expressions. The rules for actual parameter lists shall be followed as defined in 5.2.

EXAMPLE:

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

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** test component operation. These restrictions are described in 22.5.

16.1.3 Predefined functions

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

Table 10/Z.140 – List of TTCN-3 predefined functions

Category	Function	Keyword
Conversion functions	Convert `integer` value to `charstring` value	`int2char`
	Convert `integer` value to `universalcharstring` value	`int2unichar`
	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`
	Convert `integer` value to `float` value	`int2float`
	Convert `float` value to `integer` value	`float2int`
	Convert `charstring` value to `integer` value	`char2int`
	Convert `charstring` value to `octetstring` value	`char2oct`
	Convert `universal charstring` value to `integer` value	`unichar2int`
	Convert `bitstring` value to `integer` value	`bit2int`
	Convert `bitstring` value to `hexstring` value	`bit2hex`
	Convert `bitstring` value to `octetstring` value	`bit2oct`
	Convert `bitstring` value to `charstring` value	`bit2str`
	Convert `hexstring` value to `integer` value	`hex2int`
	Convert `hexstring` value to `bitstring` value	`hex2bit`
	Convert `hexstring` value to `octetstring` value	`hex2oct`
	Convert `hexstring` value to `charstring` value	`hex2str`
	Convert `octetstring` value to `integer` value	`oct2int`
	Convert `octetstring` value to `bitstring` value	`oct2bit`
	Convert `octetstring` value to `hexstring` value	`oct2hex`
	Convert `octetstring` value to `charstring` value	`oct2str`
	Convert `octetstring` value to `charstring` value	`oct2char`
	Convert `charstring` value to `integer` value	`str2int`
	Convert `charstring` value to `octetstring` value	`str2oct`
	Convert `charstring` value to `float` value	`str2float`
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`
	Return the number of elements in a structured type	`sizeoftype`
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`
String handling functions	Returns part of the input string matching the specified pattern description	`regexp`
	Returns the specified portion of the input string	`substr`
	Replaces a substring of a string with or inserts the input string into a string	`replace`
Other functions	Generate a random float number	`rnd`

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 actual parameter list shall be bound.

The full description of predefined functions is given in Annex C.

16.1.4 Restrictions for functions called from specific places

Value returning functions can be called during communication operations (in templates, template fields or in-line templates) or during snapshot evaluation (in Boolean guards of alt statements or altsteps (see 20.1.1) and in initialization of altstep local definitions (see 16.2.2)). To avoid side effects that cause changing the state of the component or the actual snapshot and to prevent different results of subsequent evaluations on an unchanged snapshot, the following operations shall not be used in functions called in the cases specified above:

- All component operations, i.e., **create**, **start** (component), **stop** (component), **kill**, **running** (component), **alive**, **done** and **killed** (see Notes 1, 3, 4 and 6).
- All port operations, i.e., **start** (port), **stop** (port), **halt**, **clear**, **send**, **receive**, **trigger**, **call**, **getcall**, **reply**, **getreply**, **raise**, **catch**, **check**, **connect**, **map** (see Notes 1, 2, 3 and 6).
- The **action** operation (see Notes 2 and 6).
- All timer operations, i.e., **start** (timer), **stop** (timer), **running** (timer), **read**, **timeout** (see Notes 4 and 6).
- Calling external functions (see Notes 4 and 6).
- Calling the **rnd** predefined function (see Notes 4 and 6).
- Changing of component variables, i.e., using component variables on the right-hand side of assignments, and in the instantiation of **out** and **inout** parameters (see Notes 4 and 6).
- Calling the **setverdict** operation (see Notes 4 and 6).
- Activation and deactivation of defaults, i.e., the **activate** and **deactivate** statements (see Notes 5 and 6).
- Calling functions with **out** or **inout** parameters (see Notes 7 and 8).

NOTE 1 – The execution of the operations **start**, **stop**, **done**, **killed**, **halt**, **clear**, **receive**, **trigger**, **getcall**, **getreply**, **catch** and **check** can cause changes to the current snapshot.

NOTE 2 – The operations **send**, **call**, **reply**, **raise**, and **action** shall be avoided for readability purposes; i.e., all communication shall be made explicit and not as a side-effect of another communication operation or the evaluation of a snapshot.

NOTE 3 – The operations **map**, **unmap**, **connect**, **disconnect**, **create** shall be avoided for readability purposes; i.e., all configuration operations shall be made explicit, and not as a side-effect of a communication operation or the evaluation of a snapshot.

NOTE 4 – Calling of external functions, **rnd**, **running**, **alive**, **read**, **setverdict**, and writing to component variables shall be avoided because it may lead to different results of subsequent evaluations of the same snapshot, thus, e.g., rendering deadlock detection impossible.

NOTE 5 – The operations **activate** and **deactivate** shall be avoided because they modify the set of defaults that is considered during the evaluation of the current snapshot.

NOTE 6 – Restrictions, except the limitation on the use of **out** or **inout** parameterization, shall apply recursively; i.e., it is disallowed to use them directly, or via an arbitrary long chain of function invocations.

NOTE 7 – The restriction of calling functions with **out** or **inout** parameters does not apply recursively; i.e., calling functions that themselves call functions with **out** or **inout** parameters is legal.

NOTE 8 – Using **out** or **inout** parameters shall be avoided because it may also lead to different results of subsequent evaluations of the same snapshot.

16.2 Altsteps

16.2.0 General

TTCN-3 uses altsteps to specify default behaviour or to structure the alternatives of an **alt** statement. Altsteps are scope units similar to functions. The altstep body defines an optional set of local definitions and a set of alternatives, the so-called *top alternatives*, that form the altstep body. The syntax rules of the top alternatives are identical to the syntax rules of the alternatives of **alt** statements.

The behaviour of an altstep can be defined by using the program statements and operations summarized in clause 18. If an altstep includes port operations or uses component variables, constants or timers, the associated component type shall be referenced using the **runs on** keywords in the altstep header. The one exception to this rule is if all ports, variables, constants and timers used within the altstep are passed in as parameters.

EXAMPLE:

```
// Given
type component MyComponentType {
  var integer MyIntVar := 0;
  timer MyTimer;
  port MyPortTypeOne PCO1, PCO2;
  port MyPortTypeTwo PCO3;
}

// Altstep definition using PCO1, PCO2, MyIntVar and MyTimer of MyComponentType

altstep AltSet_A(in integer MyPar1) runs on MyComponentType {
  [] PCO1.receive(MyTemplate(MyPar1, MyIntVar) {
      setverdict(inconc);
      }
  [] PCO2.receive {
          repeat
      }
  [] MyTimer.timeout {
      setverdict(fail);
          stop
      }
}
```

Altsteps may invoke functions and altsteps or activate altsteps as defaults. An altstep without a **runs on** clause shall never invoke a function or altstep or activate an altstep as default with a **runs on** clause locally.

16.2.1 Parameterization of altsteps

Altsteps may be parameterized. An altstep that is activated as a default shall only have **in** parameters, port parameters, and timer parameters. An altstep that is only invoked as an alternative in an **alt** statement or as stand-alone statement in a TTCN-3 behaviour description may have **in**, **out** and **inout** parameters. The rules for formal parameter lists shall be followed as defined in 5.2.

16.2.2 Local definitions in altsteps

16.2.2.0 General

Altsteps may define local definitions of constants, variables and timers. The local definitions shall be defined before the set of alternatives.

EXAMPLE:

```
altstep AnotherAltStep(in integer MyPar1) runs on MyComponentType {
  var integer MyLocalVar := MyFunction();       // local variable
  const float MyFloat := 3.41;            // local constant
  [] PCO1.receive(MyTemplate(MyPar1, MyLocalVar) {
      setverdict(inconc);
      }
  [] PCO2.receive {
       repeat
      }
}
```

16.2.2.1 Restrictions for the initialization of local definitions in altsteps

The initialization of local definitions by calling value returning functions may have side effects. To avoid side effects that cause an inconsistency between the actual snapshot and the state of the component, and to prevent different results of subsequent evaluations on an unchanged snapshot, restrictions given in 16.1.4 shall apply to the initialization of local definitions.

16.2.3 Invocation of altsteps

The invocation of an altstep is always related to an **alt** statement. The invocation may be done either implicitly by the default mechanism (see clause 21) or explicitly by a direct call within an **alt** statement (see 20.1.6). The invocation of an altstep causes no new snapshot and the evaluation of the top alternatives of an altstep is done by using the actual snapshot of the **alt** statement from which the altstep was called.

NOTE – A new snapshot within an altstep will of course be taken if, within a selected top alternative, a new **alt** statement is specified and entered.

For an implicit invocation of an altstep by means of the default mechanism, the altstep shall be activated as a default by means of an **activate** statement before the place of the invocation is reached.

EXAMPLE 1:

```
 :
var default MyDefVarTwo := activate(MySecondAltStep()); // Activation of an altstep as default
 :
```

An explicit call of an altstep within an **alt** statement looks like a function call as an alternative.

EXAMPLE 2:

```
 :
alt {
  [] PCO3.receive {
      …
     }
  [] AnotherAltStep(); // explicit call of altstep AnotherAltStep as an alternative
                       // of an alt statement
  [] MyTimer.timeout {}
}
```

When an altstep is called explicitly within an **alt** statement, the next alternative to be checked is the first alternative of the **altstep**. The alternatives of the **altstep** are checked and executed the same way as alternatives of an **alt** statement (see 20.1) with the exception that no new snapshot is taken when entering the **altstep**. An unsuccessful termination of the altstep (i.e., all top alternatives of the **altstep** have been checked and no matching branch is found) causes the evaluation of the next alternative or invocation of the default mechanism (if the explicit call is the last alternative of the **alt** statement). A successful termination may cause either the termination of the test component, i.e., the **altstep** ends with a **stop** statement, or a new snapshot and re-evaluation of the **alt** statement, i.e., the **altstep** ends with **repeat** (see 20.2) or a continuation immediately after the **alt** statement, i.e., the selected top alternative of the altstep ends without explicit **repeat** or **stop**.

An **altstep** can also be called as a stand-alone statement in a TTCN-3 behaviour description. In this case, the call of the **altstep** can be interpreted as shorthand for an **alt** statement with only one alternative describing the explicit call of the **altstep**.

EXAMPLE 3:

```
// The statement
AnotherAltStep(); // AnotherAltStep is assumed to be a correctly defined altstep

//is a shorthand for

alt {
  [] AnotherAltStep();
}
```

16.3 Functions and altsteps for different component types

See 6.7.3.

17 Test cases

17.0 General

Test cases are a special kind of function. In the module control part the **execute** statement is used to start test cases (see 27.1). 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 MTC is created, the ports of the MTC and the test system interface are instantiated, 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 header 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
}
```

17.1 Parameterization of test cases

Test cases may be parameterized. The rules for formal parameter lists shall be followed as defined in 5.2.

18 Overview of program statements and operations

The fundamental program elements of test cases, functions, altsteps and the control part of TTCN-3 modules are expressions, basic program statements such as 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 and blocks of statements and declarations).

Statements shall be executed in the order of their appearance, i.e., sequentially, as illustrated in Figure 8.

Figure 8/Z.140 – Illustration of sequential behaviour

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

EXAMPLE:

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

The specification of an empty block of statements and declarations, i.e., { }, may be found in compound statements, e.g., a branch in an **`alt`** statement, and implies that no actions are taken.

Table 11/Z.140 – Overview of TTCN-3 expressions, statements and operations

Statement	Associated keyword or symbol	Can be used in module control	Can be used in functions, test cases and altsteps	Can be used in functions called from templates, Boolean guards, or from initialization of altstep local definitions
Expressions	(…)	Yes	Yes	Yes
Basic program statements				
Assignments	`:=`	Yes	Yes	Yes (see Note 3)
Logging	`log`	Yes	Yes	Yes
Label and Goto	`label / goto`	Yes	Yes	Yes
If-else	`if (…) {…} else {…}`	Yes	Yes	Yes
For loop	`for (…) {…}`	Yes	Yes	Yes
While loop	`while (…) {…}`	Yes	Yes	Yes
Do while loop	`do {…} while (…)`	Yes	Yes	Yes
Stop execution	`stop`	Yes	Yes	
Select case	`select case (…) { case (…) {…} case else {…}}`	Yes	Yes	Yes
Behavioural program statements				
Alternative behaviour	`alt {…}`	Yes (see Note 1)	Yes	
Re-evaluation of alternative behaviour	`repeat`	Yes (see Note 1)	Yes	
Interleaved behaviour	`interleave {…}`	Yes (see Note 1)	Yes	
Returning control	`return`		Yes (see Note 4)	Yes
Statements for default handling				
Activate a default	`activate`	Yes (see Note 1)	Yes	
Deactivate a default	`deactivate`	Yes (see Note 1)	Yes	
Configuration operations				
Create parallel test component	`create`		Yes	
Connect component port to component port	`connect`		Yes	
Disconnect two component ports	`disconnect`		Yes	
Map port to test interface	`map`		Yes	
Unmap port from test system interface	`unmap`		Yes	
Get MTC component reference value	`mtc`		Yes	Yes
Get test system interface component reference value	`system`		Yes	Yes
Get own component reference value	`self`		Yes	Yes
Start execution of test component behaviour	`start`		Yes	
Stop execution of test component behaviour	`stop`		Yes	
Remove a test component from the system	`kill`		Yes	
Check termination of a PTC behaviour	`running`		Yes	
Check if a PTC exists in the test system	`alive`		Yes	
Wait for termination of a PTC behaviour	`done`		Yes	

Statement	Associated keyword or symbol	Can be used in module control	Can be used in functions, test cases and altsteps	Can be used in functions called from templates, Boolean guards, or from initialization of altstep local definitions
Wait a PTC cease to exist	`killed`		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 queue	`clear`		Yes	
Clear queue and enable sending & receiving at a port	`start`		Yes	
Disable sending and disallow receiving operations to match at a port	`stop`		Yes	
Disable sending and disallow receiving operations to match new messages/calls	`halt`		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	`setverdict`		Yes	
Get local verdict	`getverdict`		Yes	Yes
External actions				
Stimulate an (SUT) action externally	`action`	Yes	Yes	
Execution of test cases				
Execute test case	`execute`	Yes	Yes (see Note 2)	

NOTE 1 – Can be used to control timer operations only.
NOTE 2 – Can only be used in functions and altsteps that are used in module control.
NOTE 3 – Changing of component variables is disallowed.
NOTE 4 – Can be used in functions and altsteps but not in test cases.

19 Expressions and basic program statements

19.0 General

Expressions and basic program statements can be used in the control part of a module and in TTCN-3 functions, altsteps and test cases.

Table 12a/Z.140 – Overview of TTCN-3 basic program statements

Basic program statements	
Statement	**Associated keyword or symbol**
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**
Select case	**select case (…) { case (…) {…}** **case else {…} }**

19.1 Expressions

19.1.0 General

TTCN-3 allows the specification of expressions using the operators defined in clause 15. Expressions are built from other (simple) expressions. Expressions may use value returning functions only. 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.

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**.

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 a value being compatible to the type of the left-hand side. The effect of an assignment is to bind the variable 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.

EXAMPLE:

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

19.3 The Log statement

The **log** statement provides the means to write one or more log items to some logging device associated with the test control or the test component in which the statement is used. Items to be logged shall be identified by a comma-separated list in the argument of the log statement. Log items may be individual language elements specified in Table 12b or expressions composed of such log items.

It is strongly recommended that the execution of the log statement has no effect on the test behaviour. In particular, functions used in a log statement should neither explicitly nor implicitly change component variable values, port or timer status, and should not change the value of any of its **inout** or **out** parameters.

EXAMPLE:

```
var integer myVar:= 1;
log("Line 248 in PTC_A: ", myVar, " (actual value of myVar)");
// The string "Line 248 in PTC_A: 1 (actual value of myVar)" is written to some log device
// of the test system
```

NOTE 1 – Functions used in log statements should not use directly or indirectly statements other than **if…else**, **for**, **while**, **do…while**, **label**, **goto**, **return**, **mtc**, **system**, **self**, **running** (PTC or timer), **read** and **getverdict**.

NOTE 2 – It is outside the scope of this Recommendation to define complex logging and trace capabilities which may be tool dependent.

Table 12b/Z.140 – TTCN-3 language elements that can be logged

Used in a log statement	What is logged	Comment
module parameter identifier	actual value	
literal value	value	This also includes free text.
data constant identifier	actual value	
external constant identifier	actual value	
template instance	actual template or field values and matching symbols	
data type variable identifier	actual value or "UNITIALIZED"	See Notes 3 and 4.
self, **mtc**, **system** or component type variable identifier	actual value and if assigned the component instance name or "UNITIALIZED"	On logging actual values see Notes 2 to 4. Actual component states shall be logged according to Note 5.
running operation (component or timer)	return value	**true** or **false**. In case of component or timer arrays, array element specification shall be included.
alive operation (component)	return value	**true** or **false**. In case of arrays, array element specifications shall be included.
port instance	actual state	Port states shall be logged according to Note 6.
default type variable identifier	actual state or 'UNITIALIZED'	Default states shall be logged according to Note 7. See also Notes 2 to 4.
timer name	actual state	Timer states shall be logged according to Note 8.
read operation	return value	See 24.3.
predefined functions	return value	See Annex C.
function instance	return value	Only functions with return clause are allowed.
external function instance	return value	Only external functions with return clause are allowed.
formal parameter identifier	See comment column	Logging of actual parameters shall follow rules specified for the language elements they are substituting. In the case of value parameters the actual parameter value; in the case of template-type parameters, the actual template or field values and matching symbols; in the case of component type parameters the actual component reference, etc.; shall be logged. For timer parameters, the use of the read operation, and for component type and timer parameters, the use of the running operation are also allowed.

Table 12b/Z.140 – TTCN-3 language elements that can be logged

NOTE 1 – Actual value/template is the value/template at the moment of the execution of the log statement.
NOTE 2 – The type of the logged value is tool dependent.
NOTE 3 – In case of array identifiers without array element specification, actual values and for component references names of all array elements shall be logged.
NOTE 4 – The string "UNITIALIZED" shall be logged only if the log item is unbound (uninitialized).
NOTE 5 – Component states that can be logged are: Inactive, Running, Stopped and Killed (for further details, see Annex F).
NOTE 6 – Port states that can be logged are: Started and Stopped (for further details, see Annex F).
NOTE 7 – Default states that can be logged are: Activated and Deactivated.
NOTE 8 – Timer states that can be logged are: Inactive, Running and Expired (for further details, see Annex F).

19.4 The Label statement

The **label** statement allows the specification of labels in test cases, functions, altsteps 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 not as the first statement of an alternative or top alternative in an **alt** statement, **interleave** statement or **altstep**. Labels used following the **label** keyword shall be unique among all labels defined in the same test case, function, altstep or control part.

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);
   }
}
  :
```

19.5 The Goto statement

The **goto** statement can be used in functions, test cases, altsteps and the control part of a TTCN-3 module. The **goto** statement performs a jump to a **label**.

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, altsteps and the control part of a TTCN-3 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) It is not allowed to use the **goto** statement within an **interleave** statement.

EXAMPLE:

```
// The following TTCN-3 code fragment includes
  :
label L1;                          // … the definition of label L1,
MyVar := 2 * MyVar;
if (MyVar < 2000) { goto L1; }     // … a jump backward to L1,
```

```
MyVar2 := Myfunction(MyVar);
if (MyVar2 > MyVar) { goto L2; }    // … a jump forward to L2,
PCO1.send(MyVar);
PCO1.receive -> value MyVar2;
label L2;                           // … the definition of label L2,
PCO2.send(integer: 21);
alt {
  [] PCO1.receive { }
  [] PCO2.receive(integer: 67) {
      label L3;                     // … the definition of label L3,
      PCO2.send(MyVar);
      alt {
        [] PCO1.receive { }
        [] PCO2.receive(integer: 90) {
           PCO2.send(integer: 33);
           PCO2.receive(integer: 13);
           goto L4;                 // … a jump forward out of two nested alt statements,
           }
        [] 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;
 :
```

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 == "1.1.2005") { 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.

EXAMPLE 1:

```
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 block of statements in the **for** statement; if it evaluates to **false**, 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 it can be declared and initialized in the **for** statement header. If the index variable is declared and initialized 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.

EXAMPLE 2:

```
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.

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 that when using a **do-while** loop the behaviour is executed at least once before the loop condition is evaluated for the first time.

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 or in a function used by the control part of a module, it terminates the execution of the module control part. When used in a test case, altstep or function that are executed on a test component, it terminates the relevant test component.

EXAMPLE:

```
module MyModule {
    : // Module definitions
   testcase MyTestCase() runs on MyMTCType system MySystemType{
      var MyPTCType ptc:= MyPTCType.create; // PTC creation
      ptc.start(MyFunction());              // start PTC execution
       :       // test case behaviour continued
      stop     // stops the MTC, all PTCs and the whole test case
   }
   function MyFunction() runs on MyPTCType {
       :
      stop  // stops the PTC only, the test case continues
   }
   control {
       :   // test execution
      stop // stops the test campaign
   } // end control
} // end module
```

NOTE – The semantics of a **stop** statement that terminates a test component is identical to the stop component operation **self.stop** (see 22.6).

19.11 The Select Case statement

The **select case** statement is an alternative to using **if** .. **else** statements when comparing a value to one or several other values. The statement contains a header part and zero or more branches. Never more than one of the branches is executed. Schematically, the **select case** statement looks as follows:

```
select (expression)
   {
   case (templateInstance1a, templateInstance1b,…)
      statementblock1
   case (templateInstance2a, templateInstance2b,…)
      statementblock2
   case else
      statementblock3
   }
```

where templateInstance refers to a defined or in-line template and statementblock_x refers to a block of statements.

NOTE – The above schematic look is equivalent to the following schematic look using the if-else statements:

```
if (match(expression, templateInstance1a or match(expression, templateInstance1b
or …)
   statementblock1
else if (match(expression, templateInstance2a or match(expression, templateInstance2b or …)
   statementblock2
else
   statementblock3
```

In the header part of the **select case** statement, an expression shall be given. Each branch starts with the **case** keyword followed by a list of templateInstance (a list branch, which may also contain a single element) or the **else** keyword (an else branch) and a block of statements.

All templateInstance in all list branches shall be of a type compatible with the type of the expression in the header. A list branch is selected and the block of statements of the selected branch is executed only, if any of the templateInstance matches the value of the expression in the header of the statement. On executing the block of statements of the selected branch (i.e., not jumping out by a go to statement), execution continues with the statement following the select case statement.

The block of statements of an else branch is always executed if no other branch textually preceding the else branch has been selected.

Branches are evaluated in their textual order. If none of the templateInstance-s matches the value of the expression in the header and the statement contains no else branch, execution continues without executing any of the **select case** branches.

EXAMPLE:

```
select (MyModulePar)  // where MyModulePar is of charstring type
{
   case ("firstValue")
       {
        log ("The first branch is selected");
       }
   case (MyCharVar, MyCharConst)
       {
        log ("The second branch is selected");
       }
   case else
       {
        log ("The value of the module parameter MyModulePar is selected");
       }
}
```

20 Behavioural program statements

20.0 General

Behavioural program statements may be used in test cases, functions and altsteps, except for:

a) the **return** statement which shall only be used in functions; and

b) the **alt** statement, the **interleave** statement and the **repeat** statement which may also be used in module control.

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 13/Z.140 – Overview of TTCN-3 behavioural program statements

Behavioural program statements	
Statement	**Associated keyword or symbol**
Alternative behaviour	**alt { … }**
Re-evaluation of alt statements	**repeat**
Interleaved behaviour	**interleave { … }**
Returning control	**return**

20.1 Alternative behaviour

20.1.0 General

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 in Figure 9.

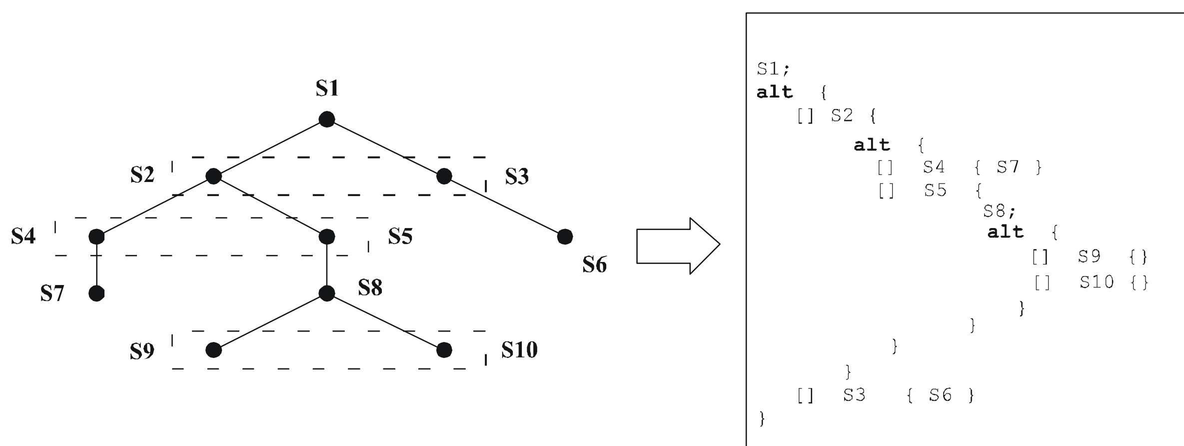


Figure 9/Z.140 – Illustration of alternative behaviour

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, done** and **killed**. The **alt** statement denotes a set of possible events that are to be matched against a particular snapshot (see 20.1.1).

EXAMPLE:

```
// Use of nested alternative statements
 :
alt {
   [] L1.receive(DL_REL_CO:*) {
       setverdict(pass);
       TAC.stop;
       TNOAC.start;
       alt {
          [] L1.receive(t_DL_EST_IN) {
                 TNOAC.stop;
                 setverdict(pass);
             }
          [] TNOAC.timeout {
             L1.send(t_DEL_EST_RQ);
             TAC.start;
             alt {
                [] L1.receive(DL_EST_CO:*) {
                       TAC.stop;
                       setverdict(pass)
                 }
                 [] TAC.timeout {
                       setverdict(inconc);
                    }
                [] L1.receive {
                       setverdict(inconc)
                    }
             }
        }
          [] L1.receive {
                 setverdict(inconc)
             }
       }
      }
   [] TAC.timeout {
       setverdict(inconc)
      }
   [] L1.receive {
       setverdict(inconc)
      }
}
 :
```

20.1.1 Execution of alternative behaviour

When entering an **alt** statement, a snapshot is taken. A snapshot is considered to be a partial state of a test component that includes all information necessary to evaluate the Boolean conditions that guard alternative branches, all relevant stopped test components, all relevant timeout events and the top messages, calls, replies and exceptions in the relevant incoming port queues. Any test component, timer and port which is referenced in at least one alternative in the **alt** statement, or in a top alternative of an altstep that is invoked as an alternative in the **alt** statement or activated as default is considered to be relevant. A detailed description of the snapshot semantics is given in the operational semantics of TTCN-3 (ITU-T Rec. Z.143 [3]).

NOTE 1 – Snapshots are only a conceptual means for describing the behaviour of the **alt** statement. The concrete algorithms for the snapshot handling can be found in ITU-T Rec. Z.143 [3].

NOTE 2 – The TTCN-3 semantics assumes that taking a snapshot is instantaneous, i.e., has no duration. In a real implementation, taking a snapshot may take some time and race conditions may occur. The handling of such race conditions is outside the scope of this Recommendation.

The alternative branches in the **alt** statement and the top alternatives of invoked altsteps and altsteps that are activated as defaults are processed in the order of their appearance. If several defaults are active, the reverse order of their activation determines the evaluation order of the top alternatives in the defaults. The alternative branches in active defaults are reached by the default mechanism described in clause 21.

The individual alternative branches are either branches that may be guarded by a Boolean expression or else-branches, i.e., alternative branches starting with `[else]`.

Else-branches are always chosen and executed when they are reached (see 20.1.3).

Branches that may be guarded by a Boolean expressions either invoke an altstep (*altstep-branch*), or start with a **done** operation (*done-branch*), a **killed** operation (*killed-branch*), **timeout** operation (*timeout-branch*) or a receiving operation (*receiving-branch*), i.e., **receive**, **trigger**, **getcall**, **getrepy, catch** or a **check** operation. The evaluation of the Boolean guards shall be based on the snapshot. The Boolean guard is considered to be *fulfilled* if no Boolean guard is defined, or if the Boolean guard evaluates to **true**. The branches are processed and executed in the following manner.

An *altstep-branch* is selected if the Boolean guard is fulfilled. The selection of an *altstep-branch* causes the invocation of the referenced altstep, i.e., the altstep is invoked and the evaluation of the snapshot continues within the altstep.

A *done-branch* is selected if the Boolean guard is fulfilled and if the specified test component is in the list of stopped components of the snapshot. The selection causes the execution of the statement block following the **done** operation. The **done** operation itself has no further effect.

A *killed-branch* is selected if the Boolean guard is fulfilled and if the specified test component is in the list of killed components of the snapshot. The selection causes the execution of the statement block following the **killed** operation. The **killed** operation itself has no further effect.

A *timeout-branch* is selected if the Boolean guard is fulfilled and if the specified timeout event is in the timeout-list of the snapshot. The selection causes execution of the specified **timeout** operation, i.e., removal of the timeout event from the timeout-list, and the execution of the statement block following the **timeout** operation.

A *receiving-branch* is selected if the Boolean guard is fulfilled and if the matching criteria of receiving operation is fulfilled by one of the messages, calls, replies or exceptions in the snapshot. The selection causes execution of the receiving operation, i.e., removal of the matching message, call, reply or exception from the port queue, maybe an assignment of the received information to a variable and the execution of the statement block following the receiving operation. In the case of the **trigger** operation, the top message of the queue is also removed if the Boolean guard is fulfilled but the matching criteria is not. In this case the statement block of the given alternative is not executed.

NOTE 3 – The TTCN-3 semantics describe the evaluation of a snapshot as a series of indivisible actions of a test component. The semantics do not assume that the evaluation of a snapshot has no duration. During the evaluation of a snapshot, test components may stop, timers may time out and new messages, calls, replies or exceptions may enter the port queues of the component However, these events do not change the actual snapshot and are thus not considered for the snapshot evaluation.

If none of the alternative branches in the **alt** statement and top alternatives in the invoked altsteps and active defaults can be selected and executed, the **alt** statement shall be executed again; i.e., a new snapshot is taken and the evaluation of the alternative branches is repeated with the new snapshot. This repetitive procedure shall continue until either an alternative branch is selected and executed, or the test case is stopped by another component or by the test system (e.g., because the MTC is stopped) or with a dynamic error.

The test case shall stop and indicate a dynamic error if a test component is completely blocked. This means none of the alternatives can be chosen, no relevant test component is running, no relevant timer is running and all relevant ports contain at least one message, call, reply or exception that do not match.

NOTE 4 – The repetitive procedure of taking a complete snapshot and re-evaluation of all alternatives is only a conceptual means for describing the semantics of the **alt** statement. The concrete algorithm that implements this semantics is outside the scope of this Recommendation.

20.1.2 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.

The evaluation of a Boolean expression guarding an alternative may have side effects. To avoid side effects that cause an inconsistency between the actual snapshot and the state of the component, the same restrictions as the restrictions for the initialization of local definitions within altsteps shall apply (see 16.2.2.1).

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.

EXAMPLE:

```
// Use of alternative with Boolean expressions (or guard)
 :
alt {
  [x>1] L2.receive {           // Boolean guard/expression
        setverdict(pass);
     }
  [x<=1] L2.receive {          // Boolean guard/expression
         setverdict(inconc);
     }
}
 :
```

20.1.3 Else branch in alternatives

Any branch in an **alt** statement can be defined as an else branch by including the **else** keyword between the opening and closing brackets at the beginning of the alternative. The else branch shall not contain any of the actions allowed in branches guarded by a Boolean expression (i.e., an **altstep** call or a **done**, a **killed**, a **timeout** or a receiving operation). The statement block of the else branch is always executed if no other alternative textually preceding the else branch has proceeded.

EXAMPLE:

```
// Use of alternative with Boolean expressions (or guard) and else branch
 :
alt {
  [x>1] L2.receive {
      setverdict(pass);
     }
  [x<=1] L2.receive {
      setverdict(inconc);
     }
  [else] {             // else branch
      MyErrorHandling();
      setverdict(fail);
      stop;
     }
}
 :
```

It should be noted that the default mechanism (see clause 21) is always invoked at the end of all alternatives. If an **else** branch is defined, the default mechanism will never be called, i.e., active defaults will never be entered.

NOTE 1 – It is also possible to use **else** in altsteps.

NOTE 2 – It is allowed to use a **repeat** statement within an **else** branch.

NOTE 3 – It is allowed to define more than one else branch in an alt statement or in an altstep, however, always only the first else branch is executed.

20.1.4 Void

20.1.5 Re-evaluation of alt statements

The re-evaluation of an **alt** statement can be specified by using a **repeat** statement (see 20.2).

EXAMPLE:

```
alt {
  [] PCO3.receive {
      count := count + 1;
      repeat                 // usage of repeat
     }
  [] T1.timeout { }
  [] any port.receive {
      setverdict(fail);
      stop;
     }
}
```

20.1.6 Invocation of altsteps as alternatives

TTCN-3 allows the invocation of altsteps as alternatives in **alt** statements (see 16.2.3).

EXAMPLE:

```
  :
alt {
   [] PCO3.receive { }
   [] AnotherAltStep(); // explicit call of altstep AnotherAltStep as alternative
                        // of an alt statement
   [] MyTimer.timeout { }
}
  :
```

20.2 The Repeat statement

The **repeat** statement, when used in the block of statements and declarations of alternatives of **alt** statements, causes the re-evaluation of the **alt** statement, i.e., a new snapshot is taken and the alternatives of the **alt** statement are evaluated in the order of their specification. When used in blocks of statements and declarations of the response, and exception handling parts of blocking procedure calls, the repeat statement causes the re-evaluation of the response and exception handling part of the call (see 23.3.1.5).

EXAMPLE 1:

```
// Usage of repeat in an alt statement
alt {
   [] PCO3.receive {
       count := count + 1;
       repeat           // usage of repeat
      }
   [] T1.timeout { }
   [] any port.receive {
       setverdict(fail);
       stop;
      }
}
```

If a **repeat** statement is used in a top alternative in an altstep definition, it causes a new snapshot and the re-evaluation of the **alt** statement from which the altstep has been called. The call of the altstep may either be done implicitly by the default mechanism (see clause 21) or explicitly in the **alt** statement (see 20.1.6).

EXAMPLE 2:

```
// Usage of repeat in an altstep
altstep AnotherAltStep() runs on MyComponentType {
   [] PCO1.receive{
       setverdict(inconc);
       repeat                 // usage of repeat
      }
   [] PCO2.receive {}
}
```

20.3 Interleaved behaviour

The **interleave** statement allows to specify the interleaved occurrence and handling of the statements **done**, **killed**, **timeout**, **receive**, **trigger**, **getcall**, **catch** and **check**.

Control transfer statements **for**, **while**, **do-while**, **goto**, **activate**, **deactivate**, **stop**, **repeat**, **return**, direct call of altsteps as alternatives 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 specify **else** branches in interleaved behaviour.

Interleaved behaviour can always be replaced by an equivalent set of nested alternatives. The procedures for this replacement and the operational semantics of interleaving are described in ITU-T Rec. Z.143 [3].

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**, **done, killed** and **timeout**. Non-reception statements denote all other non-control-transfer statements which can be used within the **interleave** statement.

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

The operational semantics of interleaving are fully defined in ITU-T Rec. Z.143 [3].

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);
   }
}
:

// is 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 The Return statement

The **return** statement terminates execution of a function or altstep and returns control to the point from which the function or altstep was called. When used in functions, a **return** statement may be optionally associated with a return value.

NOTE – The **return** statement, when used in altsteps has the same effect as if the end of the block of statements and declarations of the chosen alternative has been reached; e.g., when the altstep is called from an **alt** statement, the execution continues with the first statement following the **alt** statement.

EXAMPLE:

```
function MyFunction() return boolean {
   :
  if (date == "1.1.2005") {
     return false; // execution stops on the 1.1.2000 and returns the boolean false
  }
   :
  return true; // true is returned
}

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

21 Default handling

21.0 General

TTCN-3 allows the activation of altsteps (see 16.2) as defaults. For each test component the defaults, i.e., activated altsteps, are stored as an ordered list. The defaults are listed in the reversed order of their activation; i.e., the last activated default is the first element in the list of active defaults. The TTCN-3 operations **activate** (see 21.3) and **deactivate** (see 21.4) operate on the list of defaults. An **activate** puts a new default as the first element into the list and a **deactivate** removes a default from the list. A default in the default list can be identified by means of a default reference that is generated as a result of the corresponding **activate** operation.

Table 14/Z.140 – Overview of TTCN-3 statement for default handling

Statements for default handling	
Statement	**Associated keyword or symbol**
Activate a default	**Activate**
Deactivate a default	**deactivate**

21.1 The default mechanism

The default mechanism is evoked at the end of each **alt** statement if, due to the actual snapshot, none of the specified alternatives could be executed. An evoked default mechanism invokes the first altstep in the list of defaults, i.e., the last activated default, and waits for the result of its termination. The termination can be successful or unsuccessful. Unsuccessful means that none of the top alternatives of the **altstep** (see 16.2) defining the default behaviour could be selected, successful means that one of the top alternatives of the default has been selected and executed.

In the case of an unsuccessful termination, the default mechanism invokes the next default in the list. If the last default in the list has terminated unsuccessfully, the default mechanism will return to the place in the **alt** statement in which it has been invoked, i.e., at the end of the **alt** statement, and indicate an unsuccessful default execution. An unsuccessful default execution will also be indicated if the list of defaults is empty.

An unsuccessful default execution may cause a new snapshot or a dynamic error if the test component is blocked (see 20.1).

In the case of a successful termination, the default may either stop the test component by means of a **stop** statement, or the main control flow of the test component will continue immediately after the **alt** statement from which the default mechanism was called, or the test component will take new snapshot and re-evaluate the **alt** statement. The latter has to be specified by means of a **repeat** statement (see 20.2). If the selected top alternative of the default ends without a **repeat** statement, the control flow of the test component will continue immediately after the **alt** statement.

NOTE – TTCN-3 does not restrict the implementation of the default mechanism. It may, for example, be implemented in form of a process that is implicitly called at the end of each **alt** statement or in form of a separate thread that is only responsible for the default handling. The only requirement is that defaults are called in the reverse order of their activation when the default mechanism has been invoked.

21.2 Default references

Default references are unique references to activated defaults. Such a unique default reference is generated by a test component when an altstep is activated as a default; i.e., a default reference is the result of an **activate** operation (see 21.3).

Default references have the special and predefined type **default**. Variables of type **default** can be used to handle activated defaults in test components. The special value **null** is available to indicate an undefined default reference, e.g., for the initialization of variables to handle default references.

Default references are used in **deactivate** operations (see 21.4) in order to identify the default to be deactivated.

The actual data representation of the **default** type 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 defaults.

EXAMPLE:

```
// Declaration of a variable for the handling of defaults
// and initialization with the null value
var default MyDefaultVar := null;
  :
// Usage of MyDefaultVar for storing an activated default
MyDefaultVar := activate(MyDefAltStep()); // MyDefAltStep is activated as default
  :
// Usage of MyDefaultVar for the deactivation of default MyDefAltStep
deactivate(MyDefaultVar);
  :
```

21.3 The Activate operation

21.3.0 General

The **activate** operation is used to activate altsteps as defaults. An **activate** operation will put the referenced altstep as the first element into the list of defaults and return a default reference. The default reference is a unique identifier for the default and may be used in a **deactivate** operation for the deactivation of the default.

The effect of an **activate** operation is local to the test component in which it is called. This means that a test component cannot activate a default in another test component.

EXAMPLE 1:

```
  :
// Declaration of a variable for the handling of defaults
var default MyDefaultVar := null;
  :
// Declaration of a default reference variable and activation of an altstep as default
var default MyDefVarTwo := activate(MySecondAltStep());
  :
// Activation of altstep MyAltStep as a default
MyDefaultVar := activate(MyAltStep()); // MyAltStep is activated as default
  :
```

The **activate** operation can be called without saving the returned default reference. This form is useful in test cases which do not require explicit deactivation of the activated default; i.e., deactivation of a default is done implicitly at MTC termination.

EXAMPLE 2:

```
  :
// Activation of an altstep as a default, without assignment of default reference
activate(MyCommonDefault());
```

21.3.1 Activation of parameterized altsteps

The actual parameters of a parameterized altstep (see 16.2.1), that should be activated as a default, shall be provided in the corresponding **activate** statement. This means that the actual parameters are bound to the default at the time of

its activation (and not e.g., at the time of its invocation by the default mechanism). All timer instances in the actual parameter list shall be declared as component type local timers (see 8.5.1).

EXAMPLE:

```
altstep MyAltStep2 ( integer    par_value1, MyType par_value2,
                     MyPortType par_port,   timer  par_timer )
{
 :
}

function MyFunc () runs on MyCompType
{ :
var default MyDefaultVar := null;

MyDefaultVar := activate(MyAltStep2(5, myVar, myCompPort, myCompTimer);
  // MyAltStep2 is activated as default with the actual parameters 5 and
  // the value of myVar. A change of myVar before a call of MyAltStep2 by
  // the default mechanism will not change the actual parameters of the call.
 :
}
```

21.4 The Deactivate operation

The **deactivate** operation is used to deactivate defaults, i.e., previously activated altsteps. A **deactivate** operation will remove the referenced default from the list of defaults.

The effect of a **deactivate** operation is local to the test component in which it is called. This means that a test component cannot deactivate a default in another test component.

A **deactivate** operation without parameter deactivates all defaults of a test component.

Calling a **deactivate** operation with the special value **null** has no effect. Calling a **deactivate** operation with an undefined default reference, e.g., an old reference to a default that has already been deactivated or an uninitialized default reference variable, shall cause a runtime error.

EXAMPLE:

```
 :
var default MyDefaultVar := null;
var default MyDefVarTwo := activate(MySecondAltStep());
var default MyDefVarThree := activate(MyThirdAltStep());
 :
MyDefaultVar := activate(MyAltStep());
 :
deactivate(MyDefaultVar); // deactivates MyAltStep
 :
deactivate; // deactivates all other defaults, i.e., in this case MySecondAltStep
      // and MyThirdAltStep
 :
```

22 Configuration operations

22.0 General

Configuration operations (see Table 15) are used to set up and control test components. These operations shall only be used in TTCN-3 test cases, functions and altsteps (i.e., not in the module control part).

Table 15/Z.140 – Overview of TTCN-3 configuration operations

Operation	Explanation	Syntax examples
Connection operations		
connect	Connects the port of one test component to the port of another test component	**connect**(ptc1:p1, ptc2:p2);
disconnect	Disconnects two or more connected ports	**disconnect**(ptc1:p1, ptc2:p2);
map	Maps the port of one test component to the port of the test system interface	**map**(ptc1:q, **system**:sutPort1);
unmap	Unmaps two or more mapped ports	**unmap**(ptc1:q, **system**:sutPort1);

Table 15/Z.140 – Overview of TTCN-3 configuration operations

Operation	Explanation	Syntax examples
Test component operations		

create	Creation of a normal or alive test component; the distinction between normal and alive test components is made during creation (MTC behaves as a normal test component)	Non-alive test components:
		`var PTCType c := PTCType.create;`
		Alive test components:
		`var PTCType c := PTCType.create alive;`

start	Starting test behaviour on a test component. Starting a behaviour does not affect the status of component variables, timers or ports	`c.start(PTCBehaviour());`
stop	Stopping test behaviour on a test component	`c.stop;`
kill	Causes a test component to cease to exist	`c.kill;`
alive	Returns true if the test component has been created and is ready to execute or is executing already a behaviour; otherwise returns false	`if (c.alive) …`
running	Returns true as long as the test component is executing a behaviour; otherwise returns false	`if (c.running) …`
done	Checks whether the function running on a test component has terminated	`c.done;`
killed	Checks whether a test component has ceased to exist	`c.killed { … }`
Reference operations		
mtc	Gets the reference to the MTC	`connect(mtc:p, ptc:p);`
system	Gets the reference to the test system interface	`map(c:p, system:sutPort);`
self	Gets the reference to the test component that executes this operation	`self.stop;`

22.1 The Create operation

The MTC is the only test component, which is automatically created when a test case starts. All other test components (the PTCs) shall be 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 and with its full set of constants, variables and timers. 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.

All component variables and timers are reset to their initial value (if any) and all component constants are reset to their assigned values when the component is explicitly or implicitly created.

Two types of PTCs are distinguished: a PTC that can execute a behaviour function only once and a PTC that is kept alive after termination of a behaviour function and can be, therefore, reused to execute another function. The latter is created using the additional **alive** keyword. An alive-type PTC must be destroyed explicitly using the **kill** operation (see 22.9), whereas a non-alive PTC is destroyed implicitly after its behaviour function terminates. Termination of a test case, i.e., the MTC, terminates all PTCs that still exist, if any.

Since all test 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.

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 8.7) and can be used for connecting instances and for communication purposes such as sending and receiving.

Optionally, a name can be associated with the newly created component instance. The name shall be a **charstring** value and when assigned it shall appear as the argument of the **create** function. The test system shall associate the names 'MTC' to the MTC and 'SYSTEM' to the test system interface automatically at creation. Associated component names are not required to be unique.

NOTE – The component instance name is used for logging purposes (see 19.3) only and shall not be used to refer to the component instance (the component reference shall be used for this purpose) and has no effect on matching.

EXAMPLE:

```
// This example declares variables of type MyComponentType, which is used to store the
// references of newly created component instances of type MyComponentType which is the
// result of the create operations. An associated name is allocated to some of the created
// component instances.
 :
var MyComponentType MyNewComponent;
var MyComponentType MyNewestComponent;
var MyComponentType MyAliveComponent;
var MyComponentType MyAnotherAliveComponent;
 :
MyNewComponent := MyComponentType.create;
MyNewestComponent := MyComponentType.create("Newest");
MyAliveComponent := MyComponentType.create alive;
MyAnotherAliveComponent := MyComponentType.create("Another Alive") alive;
 :
```

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 PTC). 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.

22.2 The Connect and Map operations

22.2.0 General

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.

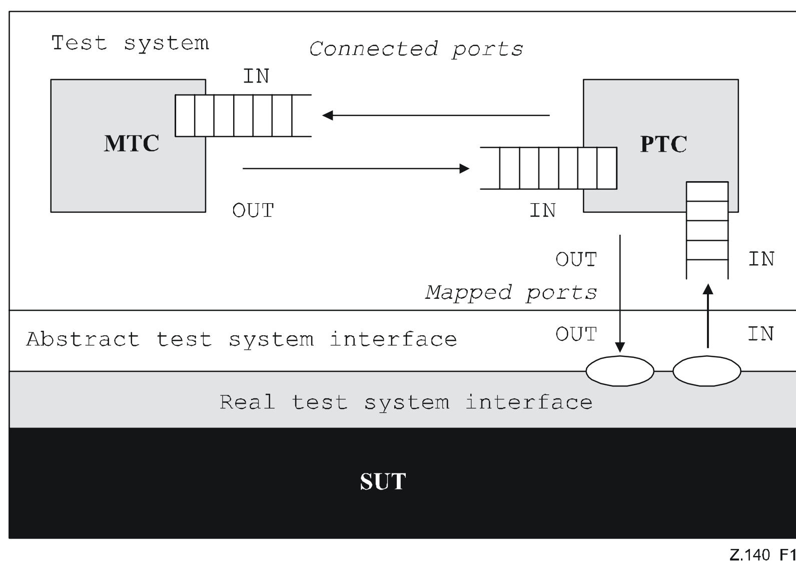


Figure 10/Z.140 – Illustration of the connect and map operations

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.

The operation **mtc** identifies the MTC, the operation **system** identifies the test system interface and the operation **self** identifies the test component in which **self** has been called (see 22.4). All these operations can be used for identifying and connecting ports.

Both the **connect** and **map** operations can be called from any behaviour definition except for the control part of a module. 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  MyNewPTC;
  :
MyNewPTC := MyComponentType.create;
  :
  :
connect(MyNewPTC:Port1, mtc:Port3);
map(MyNewPTC:Port2, system:PCO1);
  :
  :
// In this example a new component of type MyComponentType is created and its reference stored
// in variable MyNewPTC. 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
```

22.2.1 Consistent connections and mappings

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-PORT1 defines 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.

22.3 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 only has an effect if the connection or mapping to be removed has been created beforehand.

EXAMPLE 1:

```
  :
  :
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
```

To ease **disconnect** and **unmap** operations related to all connections and mappings of a component or a port, it is allowed to use **disconnect** and **unmap** operations with one argument only. This one argument specifies one side of

the connections to be disconnected or unmapped. The **all port** keyword can be used to denote all ports of a component.

EXAMPLE 2:

```
  :
disconnect(MyNewComponent:Port1);          // disconnects all connections of Port1, which
                                           // is owned by component MyNewComponent.
unmap(MyNewComponent:all port);            // unmaps all ports of component MyNewComponent
  :
```

The usage of a **disconnect** or **unmap** operation without any parameters is a shorthand form for using the operation with the parameter **self:all port**. It disconnects or unmaps all ports of the component that calls the operation.

EXAMPLE 3:

```
  :
disconnect;                              // is a shorthand form for …
disconnect(self:all port);               // which disconnects all ports of the component
                                         // that called the operation
  :
unmap;                                   // is a shorthand form for …
unmap(self:all port);                    // which unmaps all ports of the component
                                         // that called the operation
  :
```

The **all component** keyword shall only be used in combination with the **all port** keyword, i.e., **all component:all port**, and shall only be used by the MTC. Furthermore, the **all component:all port** argument shall be used as the one and only argument of a **disconnect** or **unmap** operation, and it allows release of all connections and mappings of the test configuration.

EXAMPLE 4:

```
  :
  :
disconnect(all component:all port);        // the MTC disconnects all ports of all
                                           // components in the test configuration.
  :
  :
unmap(all component:all port);             // the MTC unmaps all ports of all
                                           // components in the test configuration.
  :
```

22.4 The MTC, System and Self operations

The component reference (see 8.7) has three operations: **mtc** and **system** which return the reference of the main test component and the test system interface respectively. The operation **self** can be used to return the reference of the component in which it is called.

EXAMPLE:

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

The only operations allowed on component references are assignment, equality and non-equality.

22.5 The Start test component operation

Once a PTC has been created and connected, behaviour has to be bound to this PTC and the execution of its behaviour has to be started. This is done by using the **start** operation (as PTC 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.

An alive-type PTC may perform several behaviour functions in sequential order. Starting a second behaviour function on a non-alive PTC or starting a function on a PTC that is still running results in a test case error. If a function is started on an alive-type PTC after termination of a previous function, it uses variable values, timers, ports, and the local verdict as they were left after termination of the previous function. In particular, if a timer was started in the previous function, the subsequent function should be enabled to handle a possible timeout event.

EXAMPLE:

```
function MyFirstBehaviour() runs on MyComponentType { … }
function MySecondBehaviour() runs on MyComponentType { … }
 :
var MyComponentType MyNewPTC;
var MyComponentType MyAlivePTC;
 :
MyNewPTC := MyComponentType.create;        // Creation of a new non-alive test component.
MyAlivePTC := MyComponentType.create alive;// Creation of a new alive-type test component
 :
MyNewPTC.start(MyFirstBehaviour());        // Start of the non-alive component.
MyNewPTC.done;                             // Wait for termination
MyNewPTC.start(MySecondBehaviour());       // Test case error
 :
MyAlivePTC.start(MyFirstBehaviour());      // Start of the alive-type component
MyAlivePTC.done;                           // Wait for termination
MyAlivePTC.start(MySecondBehaviour());     // Start of the next function on the same component
   :
```

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., parameters by value.
- This function shall have a **runs on** definition referencing a component type that is compatible with the newly created component (see 6.7.3).
- Ports and timers shall not be passed into this function.

NOTE – As **in** and **inout** ports start listening when the component is created, at the moment when they start execution there may be messages in the incoming queues of such ports already waiting to be processed.

22.6 The Stop test behaviour operation

By using the **stop** test component statement a test component can stop the execution of its own currently running test behaviour or the execution of the test behaviour running on another test component. If a component does not stop its own behaviour, but the behaviour running on another test component in the test system, the component to be stopped has to be identified by using its component reference. A component can stop its own behaviour by using a simple **stop** execution statement (see 19.10) or by addressing itself in the **stop** operation, e.g., by using the **self** operation.

EXAMPLE 1:

NOTE 1 – While the **create**, **start**, **running, done** and **killed** operations can be used for PTC(s) only, the **stop** operation can also be applied to the MTC.

```
var MyComponentType MyComp := MyComponentType.create;// A new test component is created
MyComp.start(CompBehaviour());                       // The new component is started
 :
if (date == "1.1.2005") {
  MyComp.stop;         // The component "MyComp" is stopped
}

 :
if (a < b ) {
   :
  self.stop;      // The test component that is currently executing stops its own behaviour
}
 :
  stop            // The test component stops its own behaviour
```

Stopping a test component is the explicit form of terminating the execution of the currently running behaviour. A test component behaviour terminates also by completing its execution upon reaching the end of the testcase or function that is started on this component or by an explicit **return** statement. This termination is also called implicit stop. The implicit stop has the same effects as an explicit stop; i.e., the global verdict is updated with the local verdict of the stopped test component (see clause 25).

If the stopped test component is the MTC, resources of all existing PTCs shall be released, the PTCs shall be removed from the test system and the test case shall terminate (see 27.2).

Stopping a non-alive-type test component (implicitly or explicitly) shall destroy it, and all resources associated with a the test component shall be released.

Stopping an alive-type component shall stop the currently running behaviour only, but the component continues to exist and can execute new behaviour (started on it using the **start** operation). The component shall be left in a consistent state after stopping its behaviour.

NOTE 2 – For example, if the behaviour of an alive-type component is stopped during assigning a new value to an already bound variable, the variable shall remain bound after the component is stopped (with the old or the new value). Similarly, if the component is stopped during re-starting an already running timer, the timer shall be left in the running state after termination of the behaviour.

The rules for the termination of test cases and the calculation of the final test verdict are described in clause 25.

The **all** keyword can be used by the MTC only in order to stop all running PTCs but the MTC itself.

NOTE 3 – A PTC can stop the test case execution by stopping the MTC.

EXAMPLE 2:

```
  :
all component.stop      // The MTC stops all PTCs of the test case but not itself.
  :
```

NOTE 4 – The concrete mechanism for stopping PTCs is outside the scope of this Recommendation.

22.7 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 can be used for PTCs only. The running operation returns **true** for PTCs that have been started but not yet terminated or stopped. It returns **false** otherwise. 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.

When the **all** keyword is used with the **running** operation, it will return **true** if all PTCs started but not stopped explicitly by another component are executing their behaviour. Otherwise, it returns **false**.

When the **any** keyword is used with the **running** operation, it will return **true** if at least one PTC is executing its behaviour. Otherwise, it returns **false**.

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()
}
```

22.8 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 can be used for PTCs only.

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 a 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.

When the **done** operation is applied to a PTC, it matches only if the behaviour of that PTC has been stopped (implicitly or explicitly) or the PTC has been killed. Otherwise, the match is unsuccessful.

When the **all** keyword is used with the **done** operation, it matches if no one PTC is executing its behaviour. It also matches if no PTC has been created.

When the **any** keyword is used with the **done** operation, it matches if at least the behaviour of one PTC has been stopped or killed. Otherwise, the match is unsuccessful.

NOTE – Stopping the behaviour of a non-alive component also results in removing that component from the test system, while stopping an alive-type component leaves the component alive in the test system. In both cases the **done** operation matches.

EXAMPLE:

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

   []  any port.receive {
        repeat
       }
}
:

var MyComp c := MyComp.create alive;
c.start(MyPTCBehaviour());
:
c.done;
   // matches as soon as the function MyPTCBehaviour (or function/altstep called by it) stops
c.done;
   // matches the end of MyPTCBehaviour (or function/altstep called by it) too
if(c.running) {c.done}
   // done here matches the end of the next behaviour only

// 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
```

22.9 The Kill test component operation

The **kill** operation applied on a test component stops the execution of the currently running behaviour – if any – of that component and frees all resources associated to it (including all port connections of the killed component) and removes the component from the test system. The **kill** operation can be applied on the current test component itself by a simple **kill** statement or by addressing itself using the **self** operation in conjunction with the kill operation. The **kill** operation can also be applied to another test component. In this case the component to be killed shall be addressed using its component reference. If the **kill** operation is applied on the MTC, e.g., **mtc.kill**, it terminates the test case.

EXAMPLE 1:

```
var PTCType MyAliveComp := PTCType.create alive;    // Create an alive-type test component
MyAliveComp.start(MyFirstBehavior());               // The new component is started
MyAliveComp.done;                                   // Wait for termination
MyAliveComp.start(MySecondBehavior());              // Start the component a 2nd time
MyAliveComp.done;                                   // Wait for termination
MyAliveComp.kill;                                   // Free its resources
```

The **all** keyword can be used by the MTC only in order to stop and kill all running PTCs but the MTC itself.

EXAMPLE 2:

```
all component.kill;  // The MTC stops all (alive-type and normal) PTCs of the test case first
// and frees their resources.
```

22.10 The Alive operation

The **alive** operation is a Boolean operation that checks whether a test component has been created and is ready to execute or is already executing a behaviour function. Applied on a normal test component, the **alive** operation returns true if the component is inactive or running a function, and false otherwise. Applied on an alive-type test component, the operation returns true if the component is inactive, running or stopped. It returns false if the component has been killed.

The **alive** operation can be used similar to the **running** operation on PTCs only (see 22.7). In particular, in combination with the **all** keyword it returns true if all (alive-type or normal) PTCs are alive.

The **alive** operation used in combination with the **any** keyword returns true if at least one PTC is alive.

EXAMPLE:

```
  :
PTC1.done;                          // Waits for termination of the component
if (PTC1.alive) {                   // If the component is still alive …
  PTC1.start(AnotherFunction());    // … execute another function on it.
}
```

22.11 The Killed operation

The **killed** operation allows to ascertain whether a different test component is alive or has been removed from the test system.

The **killed** operation shall be used in the same manner as receiving operations. This means that it shall not be used in **boolean** expressions, but it can be used to determine an alternative in an **alt** statement or as a 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 – When checking normal test components a killed operation matches if it stopped (implicitly or explicitly) the execution of its behaviour or has been **killed** explicitly, i.e., the operation is equivalent to the **done** operation (see 22.8). When checking alive-type test components, however, the **killed** operation matches only if the component has been killed using the **kill** operation. Otherwise the **killed** operation is unsuccessful.

The **killed** operation can be used for PTCs only.

When the **all** keyword is used with the **killed** operation, it matches if all PTCs of the test case have ceased to exist. It also matches if no PTC has been created.

When the **any** keyword is used with the **killed** operation, it matches if at least one PTC ceased to exist. Otherwise, the match is unsuccessful.

EXAMPLE:

```
var MyPTCType ptc := MyPTCType.create alive;   // create an alive-type test component
timer T(10.0);                                 // create a timer
T.start;                                       // start the timer
ptc.start(MyTestBehavior());                   // start executing a function on the PTC
alt {
[] ptc.killed {                                // if the PTC was killed during execution …
  T.stop;                                      // … stop the timer and …
  setverdict(inconc);                          // … set the verdict to 'inconclusive'
   }
[] ptc.done {                                  // if the PTC terminated regularly …
  T.stop;                                      // … stop the timer and …
  ptc.start(AnotherFunction());                // … start another function on the PTC
   }
[] T.timeout {                                 // if the timeout occurs before the PTC stopped
  ptc.kill;                                    // … kill the PTC and …
  setverdict(fail);                            // … set the verdict to 'fail'
   }
}
```

22.12 Using component arrays

The **create**, **connect**, **start**, **stop** and **kill** operations do not work directly on arrays of components. Instead, a specific element of the array shall be provided as the parameter to these operations. 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.

EXAMPLE:

```
// 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  MyPtc[11];
    :
   for (i:= 0; i<=10; i:=i+1)
   {
      MyPtc[i] := MyPTCType1.create;
      connect(self:PtcCoordination, MyPtc[i]:MtcCoordination);
      MyPtc[i].start(MyPtcBehaviour());
   }
    :
}
```

22.13 Summary of the use of any and all with components

The keywords **any** and **all** may be used with configuration operations as indicated in Table 16.

Table 16/Z.140 – Any and All with components

Operation	Allowed		Example	Comment
	any (see Note)	**all** (see Note)		
create				
start				
running	Yes, but from MTC only	Yes, but from MTC only	**any component.running;** **all component.running;**	Is there any PTC performing test behaviour? Are all PTCs performing test behaviour?
alive	Yes, but from MTC only	Yes, but from MTC only	**any component.alive;** **all component.alive;**	Is there any alive PTC? Are all PTCs alive?
done	Yes, but from MTC only	Yes, but from MTC only	**any component.done;** **all component.done;**	Is there any PTC that completed execution? Did all PTCs complete their execution?
killed	Yes, but from MTC only	Yes, but from MTC only	**any component.killed;** **all component.killed;**	Is there any PTC that ceased to exist? Did all PTCs cease to exist?
stop		Yes, but from MTC only	**all component.stop;**	Stop the behaviour on all PTCs.
kill		Yes, but from MTC only	**all component.kill;**	Kill all PTCs, i.e., they cease to exist.
NOTE – **any** and **all** refer to PTCs only; i.e., the MTC is not considered.				

23 Communication operations

23.0 General

TTCN-3 supports *message-based* and *procedure-based* communication. Furthermore, TTCN-3 allows to examine the top element of incoming port queues and to control the access to ports by means of *controlling operations*.

Table 17/Z.140 – 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**
Message-based communication			
Send message	**send**	Yes	
Receive message	**receive**	Yes	
Trigger on message	**trigger**	Yes	
Procedure-based communication			
Invoke procedure call	**call**		Yes
Accept procedure call from remote entity	**getcall**		Yes
Reply to procedure call from remote entity	**reply**		Yes
Raise exception (to an accepted call)	**raise**		Yes
Handle response from a previous call	**getreply**		Yes
Catch exception (from called entity)	**catch**		Yes
Examine top element of incoming port queues			
Check msg/call/exception/reply received	**check**	Yes	Yes
Controlling operations			
Clear port queue	**clear**	Yes	Yes
Clear queue and enable sending and receiving at a port	**start**	Yes	Yes
Disable sending and disallow receiving operations to match at a port	**stop**	Yes	Yes
Disable sending and disallow receiving operations to match new messages/calls	**halt**	Yes	Yes

23.1 General format of communication operations

23.1.0 General

Operations such as **send** and **call** are used for the exchange of information among test components and between an SUT and test components. For explaining the general format of these operations, they can be structured into two groups:

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

b) a component receives a message (**receive** operation), awaits a message (**trigger** operation), accepts a procedure call (**getcall** operation), receives a reply for a previously called procedure (**getreply** operation), or catches an exception (**catch** operation). These actions are collectively referred to as *receiving operations*.

23.1.1 General format of the sending operations

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

The send part:

- specifies the port at which the specified operation shall take place;
- defines the message or procedure call to be transmitted;
- gives an (optional) address part that uniquely identifies one or more communication partners to which a message, call, reply or exception shall be sent.

The port name, operation name and value shall be present in all sending operations. The address part (denoted by the **to** keyword) is optional and need only be specified in cases of one-to-many connections where:

- unicast communication is used and one receiving entity shall be explicitly identified;
- multicast communication is used and a set of receiving entities has to be explicitly identified;
- broadcast communication is used and all entities connected to the specified port have to be addressed.

NOTE – The terms "unicast", "multicast" and "broadcast" communication are used related to port communication. This means that it is only possible to address one, several or all test components that are connected to the specified port. Unicast, multicast and broadcast can also be used for mapped ports. In this case, one, several or all entities within the SUT can be reached via the specified mapped port.

EXAMPLE 1:

Send part			**(Optional) response and exception**
Port and operation	**Value part**	**(Optional) address part**	**handling part**
`MyP1.send`	`(MyVariable + YourVariable - 2)`	`to MyPartner;`	

Response and exception handling is only needed in cases of procedure-based 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.

EXAMPLE 2:

Send part			**(Optional) response and exception handling part**
Port and operation	**Value part**	**(Optional) address part**	

`MyP1.call`	`(MyProc:{MyVar1})`		`{`
			`[] MyP1.getreply(MyProc:{MyVar2}) {}`
			`[] MyP1.catch(MyProc, ExceptionOne) {}`
			`}`

23.1.2 General format of the receiving operations

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

The receive part:

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

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

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

The port 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.

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, for storing the return value, or for storing exceptions. For the assignment part, strong typing is required; e.g., the variable used for storing a message shall have the same type as the incoming message.

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.

EXAMPLE:

Receive part				(Optional) assignment part		
Port and operation	Matching part	(Optional) address expression		(Optional) value assignment	(Optional) parameter value assignment	(Optional) sender value assignment
MyP1.getreply	(AProc:{?} value 5)		->		param (V1)	sender APeer

Receive part				(Optional) assignment part		
Port and operation	Matching part	(Optional) address expression		(Optional) value assignment	(Optional) parameter value assignment	(Optional) sender value assignment
MyP2.receive	(MyTemplate(5,7))	from APeer	->	value MyVar		

23.2 Message-based communication

23.2.0 General

Message-based communication is communication based on an asynchronous message exchange. Message-based communication is non-blocking on the **send** operation, as illustrated in Figure 11, where processing in the SENDER continues immediately after the **send** operation occurs. The RECEIVER is blocked on the **receive** operation until it processes the received message.

In addition to the **receive** operation, TTCN-3 provides a **trigger** operation that filters messages with certain matching criteria from a stream of received messages on a given incoming port. Messages at the top of the queue that do not fulfil the matching criteria are removed from the port without any further action.

Figure 11/Z.140 – Illustration of the asynchronous send and receive

23.2.1 The Send operation

23.2.1.0 General

The **send** operation is used to place a message on an outgoing message port. The message may be specified by referencing a defined template or can be defined as an in-line template. When defining the message in-line, the optional type part shall be used if there is ambiguity of the type of the message being sent.

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

EXAMPLE:

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

MyPort.send(5);                     // Sends the integer value 5 (which is an in-line template)
```

23.2.1.1 Sending unicast, multicast or broadcast

TTCN-3 supports unicast, multicast and broadcast communication. The used communication mechanism can be determined by the optional **to** clause in the **send** operation. A **to** clause can be omitted in case of a one-to-one connection where unicast communication is used and the message receiver is uniquely determined by the test system structure. A **to** clause shall be present in case of one-to-many connections.

Unicast communication is specified if the **to** clause addresses one communication partner only. Multicast communication is used if the **to** clause includes a list of communication partners. Broadcast is defined by using the **to** clause with **all component** keyword.

EXAMPLE:

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

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

MyPCO2.send(MyTemplate) to (MyPeerOne, MyPeerTwo);
                   // Specifies a multicast communication, where the value of
                   // MyTemplate is sent to the two component references stored
                   // in the variables MyPeerOne and MyPeerTwo.

MyPCO3.send(MyTemplate) to all component;
                   // Broadcast communication: the value of Mytemplate is sent to
                   // all components which can be addressed via this port. If
                   // MyPCO3 is a mapped port, the components may reside inside
                   // the SUT.
```

23.2.2 The Receive operation

23.2.2.0 General

The **receive** operation is used to receive a message from an incoming message port queue. The message may be specified by referencing a defined template or can be defined as an in-line template. When defining the message in-line, the optional type part shall be present whenever the type of the message being received is ambiguous. 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 used as an alternative of an **alt** statement and it is not successful, the execution of the test case shall continue with the next alternative of the **alt** statement.

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 are determined by the argument of the **receive** operation, i.e., may either be derived from the defined template or be specified in-line. 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.

NOTE 1 – Encoding attributes also participate in matching in an implicit way, by preventing the decoder to produce an abstract value from the received message encoded in a different way than specified by the attributes.

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.

EXAMPLE 1:

```
MyPort.receive(MyTemplate(5, MyVar));   // Matches a message that fulfils the conditions
                                        // defined by template MyTemplate at port MyPort.

MyPort.receive(A<B);                    // Matches a Boolean value that depends on the
                                        //outcome of A<B

MyPort.receive(integer:MyVar);          // Matches an integer value with the value of MyVar
                                        // at port MyPort

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

MyPort.receive(charstring:"Hello")from MyPeer; // Matches charstring "Hello" from MyPeer
```

If the match is successful, the value removed from the port queue can be stored in a variable can be retrieved and stored in a variable. This is denoted by the symbol '->' and the keyword **value**.

It is also possible to retrieve and store the component reference or address of the sender of a message. This is denoted by the keyword **sender**.

NOTE 2 – When the message is received on a connected port, only the component reference is stored in the following **sender** keyword, but the test system shall internally store the component name too, if any (to be used in logging).

EXAMPLE 2:

```
MyPort.receive(MyType:?) -> value MyVar; // The value of the received message is
                                         // assigned to MyVar.

MyPort.receive(A<B) -> sender MyPeer;    // The address of the sender is assigned to MyPeer

MyPort.receive(MyTemplate:{5, MyVarOne}) -> value MyVarTwo sender MyPeer;
// The received message value is stored in MyVarTwo and the sender address is stored in MyPeer.
```

23.2.2.1 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.

A message received by *ReceiveAnyMessage* shall not be assigned to a variable.

EXAMPLE:

```
MyPort.receive;                          // Removes the top value from MyPort.

MyPort.receive from MyPeer;              // Removes the top message from MyPort if its sender
                                         // is MyPeer

MyPort.receive -> sender MySenderVar;    // Removes the top message from MyPort and assigns
                                         // the sender address to MySenderVar
```

23.2.2.2 Receive on any port

To **receive** a message on any port, use the **any port** keywords.

EXAMPLE:

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

23.2.3 The Trigger operation

23.2.3.0 General

The **trigger** operation removes the top message from the associated incoming port queue. If that top message meets the matching criteria, the **trigger** operation behaves in the same manner as a **receive** operation. If that top message does not fulfil the matching criteria, it shall be removed from the queue without any further action. 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.

NOTE –Note 1 in 22.2.2.0 is also valid for the **trigger** operation.

The **trigger** operation can be used as a stand-alone statement in a behaviour description. In this latter case the **trigger** operation is considered to be shorthand for an **alt** statement with only one alternative; i.e., it has blocking semantics, and therefore provides the ability of waiting for the next message matching the specified template or value on that queue.

EXAMPLE 1:

```
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 2:

```
MyPort.trigger(MyType:?) from MyPartner;
// Triggers on the reception of the first message of type MyType at port MyPort
// received from MyPartner.

MyPort.trigger(MyType:?) from MyPartner -> value MyRecMessage;
// This example is almost identical to the previous example. In addition, the message which
// triggers i.e., all matching criteria are met, is stored in the variable MyRecMessage.

MyPort.trigger(MyType:?) -> sender MyPartner;
// This example is almost identical to the first example. In addition, the reference of the
// sender component will be retrieved and stored in variable MyPartner.
```

```
MyPort.trigger(integer:?) -> value MyVar sender MyPartner;
// Trigger on the reception of an arbitrary integer value which afterwards is stored in
// variable MyVar. The reference of the sender component will be stored in variable MyPartner.
```

23.2.3.1 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;
```

23.2.3.2 Trigger on any port

To **trigger** on a message at any port, use the **any port** keywords.

EXAMPLE:

```
any port.trigger
```

23.3 Procedure-based communication

23.3.0 General

The principle of procedure-based communication is to call procedures in remote entities. TTCN-3 supports *blocking* and *non-blocking* procedure-based communication. Blocking procedure-based communication is blocking on the calling and the called side, whereas non-blocking procedure-based communication is only blocking on the called side. Signatures of procedures that are used for non-blocking procedure-based communication shall be specified according to the rules in clause 13.

The communication scheme of blocking procedure-based communication is shown in Figure 12. The CALLER calls a remote procedure in the CALLEE by using the **call** operation. The CALLEE accepts the call by means of a **getcall** operation and reacts by either using a **reply** operation to answer the call or by raising (**raise** operation) an exception. The CALLER handles the reply or exception by using **getreply** or **catch** operations. In Figure 12, the blocking of CALLER and CALLEE is indicated by means of dashed lines.

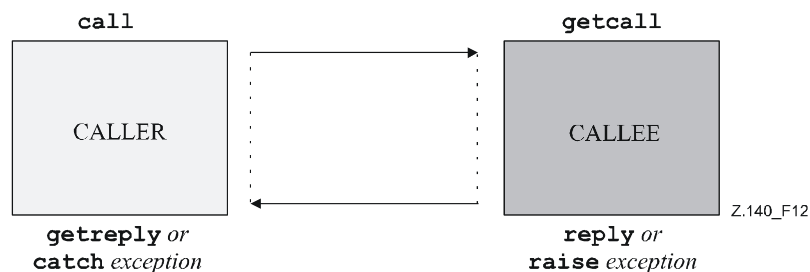


Figure 12/Z.140 – Illustration of blocking procedure-based communication

The communication scheme of non-blocking procedure-based communication is shown in Figure 13. The CALLER calls a remote procedure in the CALLEE by using the **call** operation and continues its execution, i.e., does not wait for a reply or exception. The CALLEE accepts the call by means of a **getcall** operation and executes the requested procedure. If the execution is not successful, the CALLEE may raise an exception to inform the CALLER. The CALLER may handle the exception by using a **catch** operation in an **alt** statement. In Figure 13, the blocking of the CALLEE until the end of the call handling and possible raise of an exception is indicated by means of a dashed line.

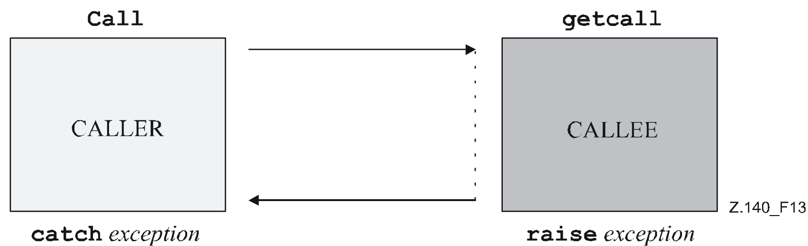


Figure 13/Z.140 – Illustration of non-blocking procedure-based communication

23.3.1 The Call operation

23.3.1.0 General

The **call** operation is used to specify that a test component calls a procedure in the SUT or in another test component. 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 information to be transmitted in the send part of the **call** operation is a signature that may either be defined in the form of a signature template or be defined in-line. 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 and exception handling part of the **call** operation by means of **getreply** and **catch** operations. This allows the use of signature templates in **call** operations in the same manner as templates can be used for types.

EXAMPLE 1:

```
// Given …
signature MyProc (out integer MyPar1, inout boolean MyPar2);
 :
// a call of MyProc
MyPort.call(MyProc:{ -, MyVar2}) {      // in-line signature template for the call of MyProc
   [] MyPort.getreply(MyProc:{?, ?}) { }
}

// … and another call of MyProc
MyPort.call(MyProcTemplate) {           // using signature template for the call of MyProc
   [] MyPort.getreply(MyProc:{?, ?}) { }
}
```

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

EXAMPLE 2:

```
MyPort.call(MyProcTemplate) to MyPeer {     // calling MyProc at MyPeer
   [] MyPort.getreply(MyProc:{?, ?}) { }
}
```

23.3.1.1 Handling responses and exceptions to a Call

In case of non-blocking procedure-based communication (see 23.3.1.4) the handling of exceptions to **call** operations is done by using **catch** (see 23.3.6) operations as alternatives in **alt** statements.

If the **nowait** option is used (see 23.3.1.2), the handling of responses or exceptions to **call** operations is done by using **getreply** (see 23.3.4) and **catch** (see 23.3.6) operations as alternatives in **alt** statements.

In case of blocking procedure-based communication, the handling of responses or exceptions to a call is done in the response and exception handling part of the **call** operation by means of **getreply** (see 23.3.4) and **catch** (see 23.3.6) operations.

The response and exception handling part of a **call** operation looks similar to the body of an **alt** statement. It defines a set of alternatives, describing the possible responses and exceptions to the call. The selection of the alternatives shall only be based on **getreply** and **catch** operations for the called procedure. Unqualified **getreply** and **catch**

operations shall only treat replies from and exceptions raised by the called procedure. The use of **else** branches and the invocation of altsteps is not allowed.

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

The response and exception handling part of a call operation is executed like an **alt** statement without any active default. This means a corresponding snapshot includes all information necessary to evaluate the (optional) Boolean guards, may include the top element (if any) of the port over which the procedure has been called and may include a timeout exception generated by the (optional) timer that supervises the call (see 23.3.1.2).

The evaluation of the Boolean expressions guarding the alternatives in the response and exception handling part may have side effects. In order to avoid unexpected side effects, the same rules as for the Boolean guards in **alt** statements shall be applied (see 20.1.1).

EXAMPLE:

```
// Given
signature MyProc3 (out integer MyPar1, inout boolean MyPar2) return MyResultType
   exception (ExceptionTypeOne, ExceptionTypeTwo);
  :

// Call of MyProc3
MyPort.call(MyProc3:{ -, true }) to MyPartner {

   [] MyPort.getreply(MyProc3:{?, ?}) -> value MyResult param (MyPar1Var,MyPar2Var) { }

   [] MyPort.catch(MyProc3, MyExceptionOne) {
        setverdict(fail);
        stop;
       }
   [] MyPort.catch(MyProc3, ExceptionTypeTwo : ?) {
        setverdict(inconc);
       }
   [MyCondition] MyPort.catch(MyProc3, MyExceptionThree) { }
}
```

23.3.1.2 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.

EXAMPLE 1:

```
MyPort.call(MyProc:{5,MyVar}, 20E-3) {

   [] MyPort.getreply(MyProc:{?, ?}) { }

   [] MyPort.catch(timeout) {            // timeout exception after 20ms
        setverdict(fail);
        stop;
       }
}
```

Using the keyword **nowait** instead of a timeout exception value in a **call** operation allows calling a procedure to continue without waiting either for a response or an exception raised by the called procedure or a timeout exception.

EXAMPLE 2:

```
MyPort.call(MyProc:{5, MyVar}, nowait);   // The calling test component will continue
                                          // its execution without waiting for the
                                          // termination of MyProc
```

If the **nowait** keyword is used, a possible response or exception of the called procedure has to be removed from the port queue by using a **getreply** or a **catch** operation in a subsequent **alt** statement.

23.3.1.3 Calling blocking procedures without return value, out parameters, inout parameters and exceptions

A blocking procedure may have no return values, no **out** and **inout** parameters and may raise no exception. The call operation for such a procedure cases shall also have a response and exception handling part to handle the blocking in a uniform manner.

EXAMPLE:

```
// Given …
signature MyBlockingProc (in integer MyPar1, in boolean MyPar2);
 :
// a call of MyBlockingProc
MyPort.call(MyBlockingProc:{ 7, false }) {
  [] MyPort.getreply( MyBlockingProc:{ -, - } ) { }
}
```

23.3.1.4 Calling non-blocking procedures

A non-blocking procedure has no **out** and **inout** parameters, no return value and the non-blocking property is indicated in the corresponding signature definition by means of a **noblock** keyword.

The **call** operation for a non-blocking procedure shall have no response and exception handling part, shall raise no timeout exception and shall not use the **nowait** keyword.

Possible exceptions raised by non-blocking procedures have to be removed from the port queue by using **catch** operations in subsequent **alt** or **interleave** statements.

23.3.1.5 Unicast, multicast and broadcast calls of procedures

Like for the **send** operation, TTCN-3 also supports unicast, multicast and broadcast calls of procedures. This can be done in the same manner as described in 23.2.1.1; i.e., the argument of the **to** clause of a **call** operation is, for unicast calls, the address of one receiving entity (or can be omitted in case of one-to-one connections); for multicast calls, a list of addresses of a set of receivers; and, for broadcast calls, the **all component** keyword. In case of one-to-one connections, the **to** clause may be omitted because the receiving entity is uniquely identified by the system structure.

The handling of responses and exceptions for a blocking or non-blocking unicast **call** operation has been explained in 23.3.1.1 to 23.3.1.4. A multicast or broadcast **call** operation may cause several responses and exceptions from different communication partners.

In case of a multicast or broadcast **call** operation of a non-blocking procedure, all exceptions which may be raised from the different communication partners can be handled in subsequent **catch**, **alt** or **interleave** statements.

In case of a multicast or broadcast **call** operation of a blocking procedure, two options exist: In the first, only one response or exception is handled in the response and exception handling part of the **call** operation. Then, further responses and exceptions can be handled in subsequent **alt** or **interleave** statements. In the second, several responses or exceptions are handled by the use of repeat statements in one or more of the block of statements and declarations of the response and exception handling part of the call operation: the execution of a repeat statement causes the re-evaluation of the call body.

NOTE – In the second case, the user needs to handle the number of repetitions.

EXAMPLE 1:

```
var boolean first:= true;
MyPort.call(MyProc:{5,MyVar}, 20E-3) to (MyPeerOne, MyPeerTwo) {     // Multicast call of MyProc
   // Handles the response from MyPeerOne
   [first] MyPort.getreply(MyProc:{?, ?}) from MyPeerOne {
       if (first) { first := false; repeat; }
       :
   }
   // Handles the response from MyPeerTwo
   [first] MyPort.getreply(MyProc:{?, ?}) from MyPeerTwo {
       if (first) { first := false; repeat; }
       :
   }
   [] MyPort.catch(timeout) {            // timeout exception after 20ms
       setverdict(fail);
       stop;
   }
}

alt {
  [] MyPort.getreply(MyProc:{?, ?}) {     // Handles all other responses to the broadcast call
       repeat
  }
}
```

In case of a multicast or broadcast **call** operation of a blocking procedure, where the **nowait** keyword is used, all responses and exceptions have to be handled in subsequent **alt** or **interleave** statements.

EXAMPLE 2:

```
MyPort.call(MyProc:{5,MyVar}) to (MyPeer1, MyPeer2) nowait; // Multicast call of MyProc

interleave {
  [] MyPort.getreply(MyProc:{?, ?}) from MyPeer1 { } // Handles the response of MyPeer1
  [] MyPort.getreply(MyProc:{?, ?}) from MyPeer2 { } // Handles the response of MyPeer2
}
```

23.3.2 The Getcall operation

23.3.2.0 General

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.

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 by using the **from** keyword.

EXAMPLE 1:

```
MyPort.getcall(MyProc: MyProcTemplate(5, MyVar)); // accepts a call of MyProc at MyPort

MyPort.getcall(MyProc:{5, MyVar}) from MyPeer;    // accepts a call of MyProc at MyPort from
MyPeer
```

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 (optional) assignment part of the **getcall** operation comprises the assignment of **in** and **inout** parameter values to variables and the retrieval of the address of the calling component. The value assignment part shall not be used with the getcall operation. The keyword **param** is used to retrieve the parameter values of a call.

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).

EXAMPLE 2:

```
MyPort.getcall(MyProc:{?, ?}) from MyPartner -> param (MyPar1Var, MyPar2Var);
// The in or inout parameter values of MyProc are assigned to MyPar1Var and MyPar2Var.

MyPort.getcall(MyProc:{5, MyVar}) -> sender MySenderVar;
// Accepts a call of MyProc at MyPort with the in or inout parameters 5 and MyVar.
// The address of the calling party is retrieved and stored in MySenderVar.

// The following getcall examples show the possibilities to use matching attributes
// and omit optional parts, which may be of no importance for the test specification.

MyPort.getcall(MyProc:{5, MyVar}) -> param(MyVar1, MyVar2) sender MySenderVar;

MyPort.getcall(MyProc:{5, ?}) -> param(MyVar1, MyVar2);

MyPort.getcall(MyProc:{?, MyVar}) -> param( - , MyVar2);
// The 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, inout integer E);

MyPort.getcall(MyProc2:{?, ?, 3, - , ?}) -> param (MyVarA, MyVarB, - , -, MyVarE);
// The parameters A, B, and E are assigned to the variables MyVarA, MyVarB, and
// MyVarE. The out parameter D needs not be considered.
```

```
MyPort.getcall(MyProc2:{?, ?, 3, -, ?}) -> param (MyVarA:= A, MyVarB:= B, MyVarE:= E);
// 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 (MyVarE:= E);
// Only the inout parameter value is needed for the further test case execution
```

23.3.2.1 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 a call from MyPartner from port MyPort

MyPort.getcall -> sender MySenderVar;  // Removes a call from MyPort and retrieves
                                       // the address of the calling entity
```

23.3.2.2 Getcall on any port

To **getcall** on any port is denoted by the **any** keyword.

EXAMPLE:

```
any port.getcall(MyProc)
```

23.3.3 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.

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

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.

Responses to one or more **call** operations may be sent to one, several or all peer entities connected to the addressed port. This can be specified in the same manner as described in 23.2.1.1. This means the argument of the **to** clause of a **reply** operation is, for unicast responses, the address of one receiving entity; for multicast responses, a list of addresses of a set of receivers; and, for broadcast responses, the **all component** keywords.

In the case of one-to-one connections, the **to** clause may be omitted, because the receiving entity is uniquely identified by the system structure.

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

EXAMPLE:

```
MyPort.reply(MyProc2:{ - ,5});          // Replies to an accepted call of MyProc2.

MyPort.reply(MyProc2:{ - ,5}) to MyPeer; // Replies to an accepted call of MyProc2 from MyPeer

MyPort.reply(MyProc2:{ - ,5}) to (MyPeer1, MyPeer2); // Multicast reply to MyPeer1 and MyPeer2

MyPort.reply(MyProc2:{ - ,5}) to all component;// Broadcast reply to all entities connected
                                               // to MyPort

MyPort.reply(MyProc3:{5,MyVar} value 20); // Replies to an accepted call of MyProc3.
```

23.3.4 The Getreply operation

23.3.4.0 General

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. The type definition of the port shall include the name of the procedure to which the **getreply** operation belongs.

The **getreply** operation shall remove the top reply from the incoming port queue, if, and only if, the matching criteria associated to the **getreply** operation are fulfilled. These matching criteria are related to the signature of the procedure 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.

Matching against a received return value can be specified by using the **value** keyword.

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

EXAMPLE 1:

```
MyPort.getreply(MyProc:{5, ?} value 20); // Accepts a reply of MyProc with two out or
                                          // inout parameters and a return value of 20

MyPort.getreply(MyProc2:{ - , 5}) from MyPeer; // Accepts a reply of MyProc2 from MyPeer
```

The signature argument of the **getreply** operation shall not be used to pass in variable names for **out** and **inout** parameters. The assignment of **out** and **inout** parameter values to variables shall be made in the assignment part of the **getreply** operation. This allows the use of signature templates in **getreply** operations in the same manner as templates are used for types.

The (optional) assignment part of the **getreply** operation comprises the assignment of **out** and **inout** parameter values to variables and the retrieval of the address of the sender of the reply. The keyword **value** is used to retrieve return values and the keyword **param** is used to retrieve the parameter values of a reply. The keyword **sender** is used when it is required to retrieve the address of the sender.

EXAMPLE 2:

```
MyPort.getreply(MyProc1:{?, ?} value ?) -> value MyRetValue param(MyPar1,MyPar2);
// The returned value is assigned to variable MyRetValue and the value
// of the two out or inout parameters are assigned to the variables MyPar1 and MyPar2.

MyPort.getreply(MyProc1:{?, ?} value ?) -> value MyRetValue param( - , MyPar2) sender MySender;
// The value of the first parameter is not considered for the further test execution and
// the address of the sender component is retrieved and 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, inout integer E);

MyPort.getreply(ATemplate) -> param( - , - , - , MyVarOut1, MyVarInout1);

MyPort.getreply(ATemplate) -> param(MyVarOut1:=D, MyVarOut2:=E);

MyPort.getreply(MyProc2:{ - , - , - , 3, ?}) -> param(MyVarInout1:=E);
```

23.3.4.1 Get any reply

A **getreply** operation with no argument list for the signature matching criteria shall remove the 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. If *GetAnyReply* is used in the response and exception handling part of a **call** operation, it shall only treat replies from the procedure invoked by the **call** operation.

EXAMPLE:

```
MyPort.getreply;              // Removes the top reply from MyPort.

MyPort.getreply from MyPeer;  // Removes the top reply received from MyPeer from MyPort.

MyPort.getreply -> sender MySenderVar;  // Removes the top reply from MyPort and retrieves the
                                        // address of the sender entity
```

23.3.4.2 Get a reply on any port

To get a reply on any port, use the **any port** keywords.

EXAMPLE:

```
any port.getreply(Myproc)
```

23.3.5 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.

Exceptions are specified as types. 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.

Exceptions to one or more **call** operations may be sent to one, several or all peer entities connected to the addressed port. This can be specified in the same manner as described in 23.2.1.1. This means that the argument of the **to** clause of a **raise** operation is, for unicast exceptions, the address of one receiving entity; for multicast exceptions, a list of addresses of a set of receivers; and, for broadcast exceptions, the **all component** keywords.

In case of one-to-one connections, the **to** clause may be omitted, because the receiving entity is uniquely identified by the system structure.

EXAMPLE:

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

MyPort.raise(MyProc, integer:5});  // Raises an exception with the integer value 5 for MyProc

MyPort.raise(MySignature, "My string") to MyPartner;
// Raises an exception with the value "My string" at MyPort for MySignature and
// send it to MyPartner

MyPort.raise(MySignature, "My string") to (MyPartnerOne, MyPartnerTwo);
// Raises an exception with the value "My string" at MyPort and sends it to MyPartnerOne and
// MyPartnerTwo (i.e., multicast communication)

MyPort.raise(MySignature, "My string") to all component;
// Raises an exception with the value "My string" at MyPort for MySignature and sends it
// to all entites connected to MyPort (i.e., broadcast communication)
```

23.3.6 The Catch operation

23.3.6.0 General

The **catch** operation is used to catch exceptions raised by a test component or the SUT 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 of the procedure that raised the exception. 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.

The **catch** operation removes the top exception from the associated incoming port queue if, and only if, that top exception satisfies all the matching criteria associated with the **catch** operation. No binding of the incoming values to the terms of the expression or to the template shall occur. The assignment of the exception values to variables shall be made in the assignment part of the **catch** operation.

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

EXAMPLE 1:

```
MyPort.catch(MyProc, integer: MyVar);   // Catches an integer exception of value
                                        // MyVar raised by MyProc at port MyPort.

MyPort.catch(MyProc, MyVar);            // Is an alternative to the previous example.

MyPort.catch(MyProc, A<B);              // Catches a boolean exception

MyPort.catch(MyProc, MyType:{5, MyVar}); // In-line template definition of an exception value.

MyPort.catch(MyProc, charstring:"Hello")from MyPeer; // Catches "Hello" exception from MyPeer
```

The (optional) assignment part of the **catch** operation comprises the assignment of the exception value and the retrieval of the address of the calling component. The keyword **value** is used to retrieve the value of an exception and the keyword **sender** is used when it is required to retrieve the address of the sender.

EXAMPLE 2:

```
MyPort.catch(MyProc, MyType:?) from MyPartner -> value MyVar;
// Catches an exception from MyPartner and assigns its value to MyVar.

MyPort.catch(MyProc, MyTemplate(5)) -> value MyVarTwo sender MyPeer;
// Catches an exception, assigns its value to MyVarTwo and retrieves the
// address of the sender.
```

The **catch** operation may be part of the response and exception handling part of a **call** operation 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 that raised 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.

23.3.6.1 The Timeout exception

There is one special **timeout** exception that can be 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 (see 23.3.1.2).

EXAMPLE:

```
MyPort.call(MyProc:{5,MyVar}, 20E-3) {
  [] MyPort.getreply(MyProc:{?, ?}) { }
  [] MyPort.catch(timeout) {            // timeout exception after 20 ms
     setverdict(fail);
     stop;
    }
}
```

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.

23.3.6.2 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. Exception values accepted by *CatchAnyException* shall not be assigned to a variable. If *CatchAnyException* is used in the response and exception handling part of a **call** operation, it shall only treat exceptions raised by the procedure invoked by the **call** operation. *CatchAnyException* will also catch the **timeout** exception.

EXAMPLE:

```
MyPort.catch;

MyPort.catch from MyPartner;

MyPort.catch -> sender MySenderVar;
```

23.3.6.3 Catch on any port

To **catch** an exception on any port use the **any** keyword.

EXAMPLE:

```
any port.catch;
```

23.4 The Check operation

23.4.0 General

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 that has to be checked and to extract the value or values of its parameters, if required.

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 taken and the receiving operation specified in the **check** operation is performed on the copy. The **check** operation fails if the receiving operation 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 statement or alternative to the check operation is evaluated. The **check** operation is successful if the receiving operation 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/during compilation.

EXAMPLE:

```
MyPort1.check(receive(5)); // Checks for an integer message of value 5.

MyPort2.check(getcall(MyProc:{5, MyVar}) from MyPartner);
// Checks for a call of MyProc at port MyPort2 from MyPartner

MyPort2.check(getreply(MyProc:{5, MyVar} value 20));
// Checks for a reply from procedure MyProc at MyPort2 where the returned value is 20 and
// the values of the two out or inout parameters are 5 and the value of MyVar.

MyPort2.check(catch(MyProc, MyTemplate(5, MyVar)));

MyPort2.check(getreply(MyProc1:{?, MyVar} value *) -> value MyReturnValue param(MyPar1,-));

MyPort.check(getcall(MyProc:{5, MyVar}) from MyPartner -> param (MyPar1Var, MyPar2Var));

MyPort.check(getcall(MyProc:{5, MyVar}) -> sender MySenderVar);
```

23.4.1 The Check any operation

A **check** operation with no argument list allows checking whether something waits for processing in an incoming port queue. The **check** any 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. In case of mixed ports, the **check** any operation checks both the message-based and the procedure-based input queues of the mixed port. If the **check** any operation matches on both input queues of the mixed port, information related to the procedure-based queue shall be given priority, i.e., returned as a result of the **check** any operation. For example, if the message-based and the procedure-based input queues of a mixed port are not empty and sender information should be retrieved by a **check** any operation, the sender of the call, reply or exception in the procedure-based input queue shall be returned.

NOTE – Information related to the message-based input queue of a mixed port can be retrieved easily by using the **check** operation in combination with a **receive** any operation, e.g.

```
MyPort.check(receive) -> sender Mysender.
```

EXAMPLE:

```
MyPort.check;

MyPort.check(from MyPartner);

MyPort.check(-> sender MySenderVar);
```

23.4.2 Check on any port

To **check** on any port, use the **any port** keywords. In case of a **check** on any port operation without argument, input queues of mixed ports shall be checked as specified in 23.4.1.

EXAMPLE:

```
any port.check;
```

23.5 Controlling communication ports

23.5.0 General

TTCN-3 operations for controlling message-based, procedure-based and mixed ports are:

- **clear**: remove the contents of the incoming port queue;
- **start**: remove the contents of the incoming port queue and enable sending and receiving operations at the port;
- **stop**: disable sending and disallow receiving operations to match at the port;
- **halt**: disable sending operations at the port immediately and disallow receiving operations to match new messages/calls/replies/exceptions that enter the port queue after the halt operation was performed. Entries already in the queue can still be processed.

23.5.1 The Clear port operation

The **clear** operation removes the contents of the *incoming* queue of the specified port. If the port queue is already empty, then this operation shall have no effect.

EXAMPLE:

```
MyPort.clear;  // clears port MyPort
```

23.5.2 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.

EXAMPLE:

```
MyPort.start;  // starts MyPort
```

By default, all ports of a component shall be started implicitly when a component is created. The start port operation will cause unstopped ports to be restarted by removing all messages waiting in the incoming queue.

23.5.3 The Stop port operation

If a port is defined as allowing receiving operations such as **receive** and **getcall,** the **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.

EXAMPLE 1:

```
MyPort.stop;   // stops MyPort
```

NOTE – To cease listening at the port means that all receiving operations defined before the stop operation shall be completely performed before the working of the port is suspended.

EXAMPLE 2:

```
MyPort.receive (MyTemplate1) -> value RecPDU;
                                // the received value is decoded, matched against
                                // MyTemplate1 and the matching value is stored
                                // in the variable RecPDU
MyPort.stop;                    // No receiving operation defined following the stop
                                // operation is executed (unless the port is restarted
                                // by a subsequent start operation)
MyPort.receive (MyTemplate2);   // This operation does not match and will block (assuming
                                // that no default is activated)
```

23.5.4 The halt port operation

If a port allows receiving operations such as **`receive`**, **`trigger`** and **`getcall`,** the **`halt`** operation disallows receiving operations to succeed for messages and procedure call elements that enter the port queue after performing the **`halt`** operation at that port. Messages and procedure call elements that were already in the queue before the **`halt`** operation can still be processed with receiving operations. If the port allows sending operations, then **`halt`** port immediately disallows sending operations such as **`send`**, **`call`**, **`raise`**, etc., to be performed. Subsequent halt operations have no effect on the state of the port or its queue.

NOTE 1 – The port **`halt`** operation virtually puts a marker after the last entry in the queue received when the operation is performed. Entries ahead of the marker can be processed normally. After all entries in the queue ahead of the marker have been processed, the state of the port is equivalent to the stopped state.

NOTE 2 – If a port **`stop`** operation is performed on a halted port before all entries in the queue ahead of the marker have been processed, further receive operations are disallowed immediately (i.e., the marker is virtually moved to the top of the queue).

NOTE 3 – A port **`start`** operation on a halted port clears all entries in the queue irrespective of whether if they arrived before or after performing the port **`halt`** operation. It also removes the marker.

NOTE 4 – A port **`clear`** operation on a halted port clears all entries in the queue irrespective of whether if they arrived before or after performing the port **`halt`** operation. It also virtually puts the marker at the top of the queue.

EXAMPLE:

```
MyPort.halt;                    // No sending allowed on Myport from this moment on;
                                // processing of messages in the queue still possible.
MyPort.receive (MyTemplate1);   // If a message was already in the queue before the halt
                                // operation and it matches MyTemplate1, it is processed;
                                // otherwise the receive operation blocks.
```

23.6 Use of any and all with ports

The keywords **`any`** and **`all`** may be used with configuration and communication operations as indicated in Table 18.

Table 18/Z.140 – Any and All with ports

Operation	Allowed		Example
	`any`	**`all`**	
`receive, trigger, getcall, getreply, catch, check`)	yes		any port.receive
`connect/map`			
`start, stop, clear, halt`		yes	all port.start

24 Timer operations

24.0 General

TTCN-3 supports a number of timer operations. These operations may be used in test cases, functions, altsteps and module control.

It is assumed that each TTCN-3 scope unit in which timers are declared, maintains its own *running-timers list* and *timeout-list*, i.e., a list of all timers that is actually running and a list of all timers that have timed out. The timeout-lists are part of the snapshots that are taken when a test case is executed. A timeout-list is updated if a timer in the scope unit is started, is stopped, times out or if a **`timeout`** operation is executed.

NOTE 1 – The running-timers list and the timeout-list are only a conceptual lists and do not restrict the implementation of timers. Other data structures such as a set, where the access to timeout events is not restricted by, e.g., the order in which the timeout events have happened, may also be used.

NOTE 2 – It is assumed that for each test component a special running-timers list and timeout-list exist that handle timer start/stop and timeout events of timers declared in the corresponding component type definition.

When a timer expires (conceptually immediately before a snapshot processing of a set of alternative events), a timeout event is placed in the timeout list of the scope unit in which the timer has been declared. The timer becomes immediately inactive. Only one entry for any particular timer may appear in the timeout list of the scope unit in which the timer has been declared at any one time.

All running timers shall automatically be cancelled when the component is explicitly or implicitly stopped.

Table 19/Z.140 – 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`**

24.1 The Start timer operation

The **`start`** timer operation is used to indicate that a timer should start running. Timer values shall be non-negative **`float`** numbers (i.e., greater than or equal to 0.0). When a timer is started, its name is added to the list of running timers (for the given scope unit).

EXAMPLE:

```
MyTimer1.start;        // MyTimer1 is started with the default duration
MyTimer2.start(20E-3); // MyTimer2 is started with a duration of 20 ms

// Elements of timer arrays may also be started in a loop, for example
timer t_Mytimer [5];
var float v_timerValues [5];

for (var integer i := 0; i<=4; i:=i+1)
  { v_timerValues [i] := 1.0 }

for (var integer i := 0; i<=4; i:=i+1)
  {t_Mytimer [i].start ( v_timerValues [i])}
```

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.

Starting a timer with the timer value 0.0 means that the timer times out immediately. Starting a timer with a negative timer value, e.g., the timer value is the result of an expression, or without a specified timer value shall cause a runtime error.

The timer clock runs from the float value zero (0.0) up to the maximum stated by the duration parameter.

The **`start`** operation may be applied to a running timer, in which case the timer is stopped and re-started. Any entry in a timeout-list for this timer shall be removed from the timeout-list.

24.2 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).

Stopping an inactive timer is a valid operation, although it does not have any effect. Stopping an expired timer causes the entry for this timer in the timeout-list to be removed. The **`all`** keyword may be used to stop all timers that are visible in the scope unit in which the **`stop`** operation has been called.

EXAMPLE:

```
MyTimer1.stop;      // stops MyTimer1
all timer.stop;     // stops all running timers
```

24.3 The Read timer operation

The **read** operation is used to retrieve the time that has elapsed since the specified timer was started. The returned value shall be of type **float**.

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, i.e., on a timer not listed on the running-timer list, will return the float value zero (0.0).

24.4 The Running timer operation

The **running** timer operation is used to check whether or not a timer is listed on the running-timer list of the given scope unit (i.e., that it has been started and has neither timed out nor been stopped). The operation returns the value **true** if the timer is listed on the list, **false** otherwise.

EXAMPLE:

```
if (MyTimer1.running) { … }
```

24.5 The Timeout operation

The **timeout** operation allows to check expiration of a timer, or of all timers, in a scope unit of a test component or module control in which the timeout operation has been called.

When a **timeout** operation is processed, if a timer name is indicated, the timeout-lists searched according to the TTCN-3 scope rules. If there is a timeout event matching the timer name, that event is removed from the timeout-list, and the **timeout** operation succeeds. The **timeout** 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 **timeout** operation is considered to be 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 timeout of timer(s).

EXAMPLE 1:

```
MyTimer1.timeout; // checks for the timeout of the previously started timer MyTimer1
```

The **any** keyword used with the **timeout** operation (rather than an explicitly named timer) succeeds if the timeout-list is not empty.

EXAMPLE 2:

```
any timer.timeout; // checks for the timeout of any previously started timer
```

24.6 Summary of use of any and all with timers

The keywords **any** and **all** may be used with timer operations as indicated in Table 20.

Table 20/Z.140 – 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**

25 Test verdict operations

25.0 General

Verdict operations allow to set and retrieve verdicts using the **setverdict** and **getverdict** operations respectively. These operations shall only be used in test cases, altsteps and functions.

Table 21/Z.140 – Overview of TTCN-3 test verdict operations

Test verdict operations	
Statement	**Associated keyword or symbol**
Set local verdict	**setverdict**
Get local verdict	**getverdict**

Each test component of the active configuration shall maintain its own local verdict. The local verdict is an object which is created for each test component at the time of its creation. It is used to track the individual verdict in each test component (i.e., in the MTC and in each and every PTC).

25.1 Test case verdict

Additionally, there is a global test case verdict instantiated and handled by the test system 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 **getverdict** and **setverdict** 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.

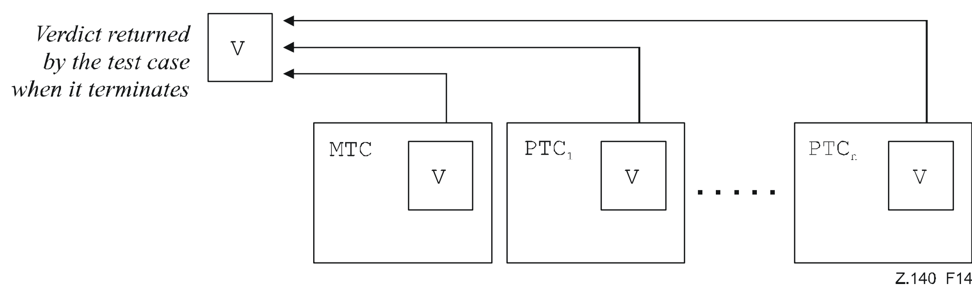


Figure 14/Z.140 – Illustration of the relationship between verdicts

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.

25.2 Verdict values and overwriting rules

25.2.0 General

The verdict can have five different values: **pass**, **fail**, **inconc**, **none** and **error**, i.e., the distinguished values of the **verdicttype** (see 6.1).

NOTE – **inconc** means an inconclusive verdict.

The **setverdict** operation shall only be used with the values **pass**, **fail**, **inconc** and **none**.

EXAMPLE 1:

```
setverdict(pass);
setverdict(inconc);
```

The value of the local verdict may be retrieved using the **getverdict** operation.

EXAMPLE 2:

```
MyResult := getverdict; // 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 local verdict (i.e., using the **`setverdict`** operation), the effect of this change shall follow the overwriting rules listed in Table 22. The test case verdict is implicitly updated on the termination of a test component. The effect of this implicit operation shall also follow the overwriting rules listed in Table 22.

Table 22/Z.140 – 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 3:

```
  :
setverdict(pass); // the local verdict is set to pass
  :
setverdict(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
```

25.2.1 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 **`setverdict`** operation and will not be returned by the **`getverdict`** 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.

26 External actions

In some testing situations, some interface(s) to the SUT may be missing or unknown *a priori* (e.g., management interface) but it may be necessary that the SUT is stimulated to carry out certain actions (e.g., send a message to the test system). Also, certain actions may be required from the test executing personnel (e.g., to change the environmental conditions of testing such as the temperature, voltage of the power feeding, etc.).

The required action may be described as a string expression; i.e., the use of literal strings, string typed variables and parameters, etc., and any concatenation thereof are allowed.

EXAMPLE:

```
var charstring myString:= " now."
action("Send MyTemplate on lower PCO" & myString);  // Informal description of the
                                                     // external action
```

There is no specification of what is done to or by the SUT to trigger this action, only an informal description of the required action itself.

External actions can be used in test cases, functions, altsteps and module control.

27 Module control part

27.0 General

Test cases are defined in the module definitions part while the module control part manages their execution. All variables (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 components and ports conducted by **`create`**, **`connect`,** etc., operations in a previous test case were destroyed when that test case was stopped (hence are not 'visible' to the new test case).

27.1 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 case verdict of either **none**, **pass**, **inconc**, **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 (see 27.5).

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'
```

27.2 Termination of test cases

A test case terminates with the termination of the MTC. On termination of the MTC (explicitly or implicitly), all running parallel test components shall be removed by the test system.

NOTE 1 – The concrete mechanism for stopping all PTCs is tool specific and therefore outside the scope of this Recommendation.

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 25. The actual local verdict of a test component becomes its final local verdict when the test component terminates itself or is stopped by itself, another test component or by the 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** or **killed** statement) before it stops itself.

27.3 Controlling execution of test cases

Program statements, limited to those defined in Tables 11 and 12 may be used in the control part of a module to specify such things as the order in which the test cases are to be executed or the number of times a test case should run.

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.

The selection and deselection of test cases can also be used to control the execution of test cases (see 27.4).

27.4 Selection of test cases

There are different ways in TTCN-3 to select and deselect test cases. For example, 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.

EXAMPLE 1:

```
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.

EXAMPLE 2:

```
function MyTestCaseGroup1()
{  execute(MySimpleTestCase1());
   execute(MySimpleTestCase2());
   execute(MySimpleTestCase3());
}
function MyTestCaseGroup2()
{  execute(MySimpleTestCase4());
   execute(MySimpleTestCase5());
   execute(MySimpleTestCase6());
}
 :
control
{  if (MySelectionExpression1()) { MyTestCaseGroup1(); }
   if (MySelectionExpression2()) { MyTestCaseGroup2(); }
    :
}
```

As a test case returns a single value of type **verdicttype**, it is also possible to control the order of test case execution depending on the outcome of a test case. The use of the TTCN-3 **verdicttype** is another way to select test cases.

EXAMPLE 3:

```
if ( execute (MySimpleTestCase()) == pass )
 { execute (MyGoOnTestCase()) }
else
 { execute (MyErrorRecoveryTestCase()) };
```

27.5 Use of timers in control

Timers may be used to supervise the execution of a test case. This may be done using an explicit timeout in the **execute** statement. 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. The timer used for test case supervision is a system timer and need not be declared or started.

EXAMPLE 1:

```
MyReturnVal := execute (MyTestCase(), 7E-3);
// Where the return verdict will be error if MyTestCase does not complete execution
// within 7 ms
```

Timer operations may also be used explicitly to control test case execution.

EXAMPLE 2:

```
// 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 := 1.0;
 :
execute(MyTestCase1());
T1.start;
T1.timeout; // Pause before executing the next test case
execute(MyTestCase2());
```

28 Specifying attributes

28.0 General

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 defined as a free text string.

There are four 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) **variant**: allows references to specific encoding variants;

d) **extension**: allows the specification of user-defined attributes.

28.1 Display attributes

All TTCN-3 language elements can have **display** attributes to specify how particular language elements should be displayed, for example, in a tabular format.

Special attribute strings related to the display attributes for the tabular (conformance) presentation format can be found in ITU-T Rec. Z.141 [1].

Special attribute strings related to the display attributes for the graphical presentation format can be found in ITU-T Rec. Z.142 [2].

Other **display** attributes may be defined by the user.

NOTE – Because user-defined attributes are not standardized, the interpretation of these attributes may differ between tools or may even not be supported.

28.2 Encoding of values

28.2.0 General

Encoding rules define how a particular value, template, etc., shall be encoded and transmitted over a communication **port** and how received signals shall be decoded. 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.

In TTCN-3, general or particular encoding rules can be specified by using **encode** and **variant** attributes.

28.2.1 Encode attributes

The **encode** attribute allows the association of some referenced encoding rule or encoding directive to be made to a TTCN-3 definition.

The manner in which the actual encoding rules are defined (e.g., prose, functions, etc.) is outside the scope of this Recommendation. 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 and the lowest is an individual type or definition:

a) **module**: encoding applies to all types defined in the module, including TTCN-3 types (built-in types);

b) **group**: encoding applies to a group of user-defined type definitions;

c) **type or definition**: encoding applies to a single user-defined type or definition;

d) **field**: encoding applies to a field in a **record** or **set** type or **template**.

EXAMPLE:

```
module MyTTCNmodule
{ :
  import from MySecondModule {
     type MyRecord
  }
  with { encode "MyRule 1" } // Instances of MyRecord will be encoded according to MyRule 1

  :
  type charstring MyType;  // Normally encoded according to the 'Global encoding 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" }
```

28.2.2 Variant attributes

To specify a refinement of the currently specified encoding scheme instead of its replacement, the **variant** attribute shall be used. The variant attributes are different from other attributes, because they are closely related to encode attributes. Therefore, for variant attributes, additional overwriting rules apply (see 28.5.1).

EXAMPLE:

```
module MyTTCNmodule1
{ :
  type charstring MyType;  // Normally encoded according to the 'Global encoding rule'
  :
  group MyRecords
  {   :
     type record MyPDU1
     {
      integer    field1,     // field1 will be encoded according to 'Rule 2'
                             // using encoding variant 'length form 3'
      Mytype     field3      // field3 will be encoded according to 'Rule 2'
                             // using any possible length encoding format
     }
     with { variant (field1) "length form 3" }
     :
  }
  with { encode "Rule 2" }

}
with { encode "Global encoding rule" }
```

28.2.3 Special strings

The following strings are the predefined (standardized) **variant** attributes for simple basic types (see E.2.1):

a) `"8 bit"` and `"unsigned 8 bit"` mean, when applied to integer and enumerated types, that the integer value or the integer numbers associated with enumerations shall be handled as it was represented on 8 bits (a single byte) within the system.

b) `"16 bit"` and `"unsigned 16 bit"` mean, when applied to integer and enumerated types, that the integer value or the integer numbers associated with enumerations shall be handled as it was represented on 16 bits (two bytes) within the system.

c) `"32 bit"` and `"unsigned 32 bit"` mean, when applied to integer and enumerated types, that the integer value or the integer numbers associated with enumerations shall be handled as it was represented on 32 bits (four bytes) within the system.

d) `"64 bit"` and `"unsigned 64 bit"` mean, when applied to integer and enumerated types, that the integer value or the integer numbers associated with enumerations shall be handled as it was represented on 64 bits (eight bytes) within the system.

e) "IEEE754 float","IEEE754 double", "IEEE754 extended float" and "IEEE754 extended double" mean, when applied to a float type, that the value shall be encoded and decoded according to the standard IEEE 754 (see bibliography).

The following strings are the predefined (standardized) **variant** attributes for **charstring** and **universal charstring** (see E.2.2):

a) "UTF-8" means, when applied to the universal charstring type, that each character of the value shall be individually encoded and decoded according to the UCS Transformation Format 8 (UTF-8) as defined in Annex R of ISO/IEC 10646 [10].

b) "UCS-2" means, when applied to the universal charstring type, that each character of the value shall be individually encoded and decoded according to the UCS-2 coded representation form (see 14.1 of ISO/IEC 10646 [10]).

c) "UTF-16" means, when applied to the universal charstring type, that each character of the value shall be individually encoded and decoded according to the UCS Transformation Format 16 (UTF-16) as defined in Annex Q of ISO/IEC 10646 [10].

d) "8 bit" means, when applied to charstring and universal charstring types, that each character of the value shall be individually encoded and decoded according to the coded representation as specified in ISO/IEC 8859 (an 8-bit coding).

The following string is the predefined (standardized) **variant** attribute for structured types (see E.2.3):

a) "IDL:fixed FORMAL/01-12-01 v.2.6" means, when applied to a record type, that the value shall be handled as an IDL fixed point decimal value (see bibliography).

These variant attributes can be used in combination with the more general encode attributes specified at a higher level. For example a **universal charstring** specified with the **variant** attribute "UTF-8" within a module which itself has a global encoding attribute "BER:1997" (see 12.2/Z.146 [6]) will cause each character of the values within the string to first be encoded following the UTF-8 rules and then this UTF-8 value will be encoded following the more global BER rules.

28.2.4 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.

28.3 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.

28.4 Scope of attributes

A **with** statement may associate attributes to a single language element. It is also possible to associate attributes to a number of language elements by, e.g., listing fields of a structured type in an attribute statement associated with a single type definition or associating a **with** statement to the surrounding scope unit or **group** of language elements.

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"}
}
```

28.5 Overwriting rules for attributes

An attribute definition in a lower scope unit will override a general attribute definition in a higher scope. Additional overwriting rules for variant attributes are defined in 28.5.1.

EXAMPLE 1:

```
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" }
```

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 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 MyPDUs 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" }
   }
}
```

An attribute definition in a lower scope can be overwritten in a higher scope by using the **override** directive.

EXAMPLE 2:

```
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.

28.5.1 Additional overwriting rules for variant attributes

A **variant** attribute is always related to an **encode** attribute. Whereas a variant of an encoding may change, an encoding shall not change without overwriting all current variant attributes. Therefore, for variant attributes the following overwriting rules apply:

- a **variant** attribute overwrites a current **variant** attribute according to the rules defined in 28.5;
- an **encoding** attribute, which overwrites a current **encoding** attribute according to the rules defined in 28.5, also overwrites a corresponding current **variant** attribute; i.e., no new **variant** attribute is provided, but the current **variant** attribute becomes inactive;
- an **encoding** attribute, which changes a current **encoding** attribute of an imported language element according to the rules defined in 28.6, also changes a corresponding current **variant** attribute; i.e., no new **variant** attribute is provided, but the current **variant** attribute becomes inactive.

EXAMPLE:

```
module MyVariantEncodingModule {
  :
  type charstring MyCharString;    // Normally encoded according to "Encoding 1"
  :
  group MyVariantsOne {
      :
      type record MyPDUone
      {
       integer    field1,   // field1 will be encoded according to "Encoding 2" only.
                            // "Encoding 2" overwrites "Encoding 1" and variant 'Variant 1'
       Mytype     field3    // field3 will be encoded according to "Encoding 1" with
                            // variant "Variant 1".
      }
      with { encoding (field1) "Encoding 2" }
      :
  }
  with { variant "Variant 1" }

  group MyVariantsTwo
  {   :
      type record MyPDUtwo
      {
       integer    field1,   // field1 will be encoded according to "Encoding 3"
                            // using encoding variant 'Variant 3'
       Mytype     field3    // field3 will be encoded according to "Encoding 3"
                            // using encoding variant "Variant 2"
      }
      with { variant (field1) "Variant 3" }
      :
  }
  with { encode "Encoding 3"; variant 'Variant 2'}

}
with { encode "Encoding 1" }
```

28.6 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 from MyModule {
  type MyType
}
with { display "ASP" }  // MyType will be displayed as ASP

import from MyModule {
  group MyGroup
}
with {
  display "PDU";       // By default all types will be displayed as PDU
  extension "MyRule"
}
```

Annex A

BNF and static semantics

A.1 TTCN-3 BNF

A.1.0 General

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/Z.140 – The syntactic metanotation

<table>
<tr><td>::=</td><td>is defined to be</td></tr>
<tr><td>abc xyz</td><td>abc followed by xyz</td></tr>
<tr><td>|</td><td>alternative</td></tr>
<tr><td>[abc]</td><td>0 or 1 instances of abc</td></tr>
<tr><td>{abc}</td><td>0 or more instances of abc</td></tr>
<tr><td>{abc}+</td><td>1 or more instances of abc</td></tr>
<tr><td>(...)</td><td>textual grouping</td></tr>
<tr><td>Abc</td><td>the non-terminal symbol abc</td></tr>
<tr><td>"abc"</td><td>a terminal symbol abc</td></tr>
</table>

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 block of statements, operations and declarations.

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 */.

EXAMPLE 1:

```
/* 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*>.

EXAMPLE 2:

```
// 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.

EXAMPLE 3:

```
// 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 Tables A.2 and A.3.

Table A.2/Z.140 – 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 predefined function identifiers defined in Table 10 and described in Annex C shall also be treated as reserved words.

Table A.3/Z.140 – List of TTCN-3 terminals which are reserved words

action	fail	noblock	select
activate	false	none	self
address	float	not	send
alive	for	not4b	sender
all	from	nowait	set
alt	function	null	setverdict
altstep			signature
and	getverdict	octetstring	start
and4b	getcall	of	stop
any	getreply	omit	subset
anytype	goto	on	superset
	group	optional	system
bitstring		or	
boolean	hexstring	or4b	template
		out	testcase
case	if	override	timeout
call	ifpresent		timer
catch	import	param	to
char	in	pass	trigger
charstring	inconc	pattern	true
check	infinity	port	type
clear	inout	procedure	
complement	integer		union
component	interleave	raise	universal
connect		read	unmap
const	kill	receive	
control	killed	record	value
create			valueof
	label	rem	var
deactivate	language	repeat	variant
default	length	reply	verdicttype
disconnect	log	return	
display		running	while
do	map	runs	with
done	match		
	message		xor
else	mixed		xor4b
encode	mod		
enumerated	modifies		
error	module		
except	modulepar		
exception	mtc		
execute			
extends			
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.1.6.0 TTCN-3 module

```
1. TTCN3Module ::= TTCN3ModuleKeyword TTCN3ModuleId
                   "{"
                   [ModuleDefinitionsPart]
                   [ModuleControlPart]
                   "}"
                   [WithStatement] [SemiColon]
2. TTCN3ModuleKeyword ::= "module"
3. TTCN3ModuleId ::= ModuleId
4. ModuleId ::= GlobalModuleId [LanguageSpec]
/* STATIC SEMANTICS - LanguageSpec may only be omitted if the referenced module contains TTCN-3
notation */
5. GlobalModuleId ::= ModuleIdentifier
6. ModuleIdentifier ::= Identifier
7. LanguageSpec ::= LanguageKeyword FreeText
8. LanguageKeyword ::= "language"
```

A.1.6.1 Module definitions part

A.1.6.1.0 General

```
9. ModuleDefinitionsPart ::= ModuleDefinitionsList
10. ModuleDefinitionsList ::= {ModuleDefinition [SemiColon]}+
11. ModuleDefinition ::= (TypeDef |
                          ConstDef |
                          TemplateDef |
                          ModuleParDef |
                          FunctionDef |
                          SignatureDef |
                          TestcaseDef |
                          AltstepDef |
                          ImportDef |
                          GroupDef |
                          ExtFunctionDef |
                          ExtConstDef) [WithStatement]
```

A.1.6.1.1 Typedef definitions

```
12. TypeDef ::= TypeDefKeyword TypeDefBody
13. TypeDefBody ::= StructuredTypeDef | SubTypeDef
14. TypeDefKeyword ::= "type"
15. StructuredTypeDef ::= RecordDef |
                          UnionDef |
                          SetDef |
                          RecordOfDef |
                          SetOfDef |
                          EnumDef |
                          PortDef |
                          ComponentDef
16. RecordDef ::= RecordKeyword StructDefBody
17. RecordKeyword ::= "record"
18. StructDefBody ::= (StructTypeIdentifier [StructDefFormalParList] | AddressKeyword)
                      "{" [StructFieldDef {"," StructFieldDef}] "}"
19. StructTypeIdentifier ::= Identifier
20. StructDefFormalParList ::= "(" StructDefFormalPar {"," StructDefFormalPar} ")"
21. StructDefFormalPar ::=  FormalValuePar
/* STATIC SEMANTICS - FormalValuePar shall resolve to an in parameter */
22. StructFieldDef ::= (Type | NestedTypeDef) StructFieldIdentifier [ArrayDef] [SubTypeSpec]
                       [OptionalKeyword]
23. NestedTypeDef ::= NestedRecordDef |
                      NestedUnionDef |
                      NestedSetDef |
                      NestedRecordOfDef |
                      NestedSetOfDef |
                      NestedEnumDef
24. NestedRecordDef ::= RecordKeyword "{" [StructFieldDef {"," StructFieldDef}] "}"
25. NestedUnionDef ::= UnionKeyword "{" UnionFieldDef {"," UnionFieldDef} "}"
26. NestedSetDef ::= SetKeyword "{" [StructFieldDef {"," StructFieldDef}] "}"
27. NestedRecordOfDef ::= RecordKeyword [StringLength] OfKeyword (Type | NestedTypeDef)
28. NestedSetOfDef ::= SetKeyword [StringLength] OfKeyword (Type | NestedTypeDef)
29. NestedEnumDef ::= EnumKeyword "{" EnumerationList "}"
30. StructFieldIdentifier ::= Identifier
31. OptionalKeyword ::= "optional"
32. UnionDef ::= UnionKeyword UnionDefBody
33. UnionKeyword ::= "union"
34. UnionDefBody ::= (StructTypeIdentifier [StructDefFormalParList] | AddressKeyword)
```

```
                     "{" UnionFieldDef {"," UnionFieldDef} "}"
35. UnionFieldDef ::= (Type | NestedTypeDef) StructFieldIdentifier [ArrayDef] [SubTypeSpec]
36. SetDef ::= SetKeyword StructDefBody
37. SetKeyword ::= "set"
38. RecordOfDef ::= RecordKeyword [StringLength] OfKeyword StructOfDefBody
39. OfKeyword ::= "of"
40. StructOfDefBody ::= (Type | NestedTypeDef) (StructTypeIdentifier | AddressKeyword) [SubTypeSpec]
41. SetOfDef ::= SetKeyword [StringLength] OfKeyword StructOfDefBody
42. EnumDef ::= EnumKeyword (EnumTypeIdentifier | AddressKeyword)
                "{" EnumerationList "}"
43. EnumKeyword ::= "enumerated"
44. EnumTypeIdentifier ::= Identifier
45. EnumerationList ::= Enumeration {"," Enumeration}
46. Enumeration ::= EnumerationIdentifier ["("[Minus] Number ")"]
47. EnumerationIdentifier ::= Identifier
48. SubTypeDef ::= Type (SubTypeIdentifier | AddressKeyword)  [ArrayDef] [SubTypeSpec]
49. SubTypeIdentifier ::= Identifier
50. SubTypeSpec ::= AllowedValues [StringLength] | StringLength
/* STATIC SEMANTICS - AllowedValues shall be of the same type as the field being subtyped */
51. AllowedValues ::= "(" (ValueOrRange {"," ValueOrRange}) | CharStringMatch ")"
52. ValueOrRange ::= RangeDef | ConstantExpression
/* STATIC SEMANTICS - RangeDef production shall only be used with integer, charstring, universal
charstring or float based types */
/* STATIC SEMANTICS - When subtyping charstring or universal charstring range and values shall not
be mixed in the same SubTypeSpec */
53. RangeDef ::= LowerBound ".." UpperBound
54. StringLength ::= LengthKeyword "(" SingleConstExpression [".." UpperBound] ")"
/* STATIC SEMANTICS - StringLength shall only be used with String types or to limit set of and
record of. SingleConstExpression and UpperBound shall evaluate to non-negative integer values (in
case of UpperBound including infinity) */
55.  LengthKeyword ::= "length"
56. PortType ::= [GlobalModuleId Dot] PortTypeIdentifier
57. PortDef ::= PortKeyword PortDefBody
58. PortDefBody ::= PortTypeIdentifier PortDefAttribs
59. PortKeyword ::= "port"
60. PortTypeIdentifier ::= Identifier
61. PortDefAttribs ::= MessageAttribs | ProcedureAttribs | MixedAttribs
62. MessageAttribs ::= MessageKeyword
                       "{" {MessageList [SemiColon]}+ "}"
63. MessageList ::= Direction AllOrTypeList
64. Direction ::= InParKeyword | OutParKeyword | InOutParKeyword
65. MessageKeyword ::= "message"
66. AllOrTypeList ::= AllKeyword | TypeList
/* NOTE: The use of AllKeyword in port definitions is deprecated */
67. AllKeyword ::= "all"
68. TypeList ::= Type {"," Type}
69. ProcedureAttribs ::= ProcedureKeyword
                         "{" {ProcedureList  [SemiColon]}+ "}"
70. ProcedureKeyword ::= "procedure"
71. ProcedureList ::= Direction AllOrSignatureList
72. AllOrSignatureList ::= AllKeyword | SignatureList
73. SignatureList ::= Signature {"," Signature}
74. MixedAttribs ::= MixedKeyword
                     "{" {MixedList [SemiColon]}+ "}"
75. MixedKeyword ::= "mixed"
76. MixedList ::= Direction ProcOrTypeList
77. ProcOrTypeList ::= AllKeyword | (ProcOrType {"," ProcOrType})
78. ProcOrType ::= Signature | Type
79. ComponentDef ::= ComponentKeyword ComponentTypeIdentifier
                     [ExtendsKeyword ComponentType {"," ComponentType}]
                     "{" [ComponentDefList] "}"
80. ComponentKeyword ::= "component"
81. ExtendsKeyword ::= "extends"
82. ComponentType ::= [GlobalModuleId Dot] ComponentTypeIdentifier
83. ComponentTypeIdentifier ::= Identifier
84. ComponentDefList ::= {ComponentElementDef [SemiColon]}
85. ComponentElementDef ::= PortInstance | VarInstance | TimerInstance | ConstDef
86. PortInstance ::= PortKeyword PortType PortElement {"," PortElement}
87. PortElement ::= PortIdentifier [ArrayDef]
88. PortIdentifier ::= Identifier
```

A.1.6.1.2 Constant definitions

```
89. ConstDef ::= ConstKeyword Type ConstList
/* STATIC SEMANTICS - Type shall follow the rules given in clause 9.*/
90. ConstList ::= SingleConstDef  {"," SingleConstDef}
91. SingleConstDef ::= ConstIdentifier [ArrayDef] AssignmentChar ConstantExpression
/* STATIC SEMANTICS - The Value of the ConstantExpression shall be of the same type as the stated
type for the constants */
92. ConstKeyword ::= "const"
```

```
93. ConstIdentifier ::= Identifier
```

A.1.6.1.3 Template definitions

```
94. TemplateDef ::= TemplateKeyword BaseTemplate  [DerivedDef] AssignmentChar TemplateBody
95. BaseTemplate ::= (Type | Signature) TemplateIdentifier  ["(" TemplateFormalParList ")"]
96. TemplateKeyword ::= "template"
97. TemplateIdentifier ::= Identifier
98. DerivedDef ::= ModifiesKeyword TemplateRef
99. ModifiesKeyword ::= "modifies"
100. TemplateFormalParList ::= TemplateFormalPar {"," TemplateFormalPar}
101. TemplateFormalPar ::= FormalValuePar | FormalTemplatePar
/* STATIC SEMANTICS - FormalValuePar shall resolve to an in parameter */
102. TemplateBody ::= (SimpleSpec | FieldSpecList | ArrayValueOrAttrib) | [ExtraMatchingAttributes]
/* STATIC SEMANTICS - Within TeplateBody the ArrayValueOrAttrib can be used for array, record,
record of and set of types. */
103. SimpleSpec ::= SingleValueOrAttrib
104. FieldSpecList ::= "{"[FieldSpec {"," FieldSpec}] "}"
105. FieldSpec ::= FieldReference AssignmentChar TemplateBody
106. FieldReference ::= StructFieldRef | ArrayOrBitRef | ParRef
/* STATIC SEMANTICS - Within FieldReference ArrayOrBitRef can be used for record of and set of
templates/template fields in modified templates only*/
107. StructFieldRef ::= StructFieldIdentifier| PredefinedType | TypeReference
/* STATIC SEMANTICS - PredefinedType and TypeReference shall be used for anytype value notation
only. PredefinedType shall not be AnyTypeKeyword.*/

108. ParRef ::= SignatureParIdentifier
/* STATIC SEMANTICS - SignatureParIdentifier shall be a formal parameter Identifier from the
associated signature definition */
109. SignatureParIdentifier ::= ValueParIdentifier
110. ArrayOrBitRef ::= "[" FieldOrBitNumber "]"
/* STATIC SEMANTICS - ArrayRef shall be optionally used for array types and ASN.1 SET OF and
SEQUENCE OF and TTCN-3 record of and set of. The same notation can be used for a Bit reference
inside an ASN.1 or TTCN-3 bitstring type */
111. FieldOrBitNumber ::= SingleExpression
/* STATIC SEMANTICS - SingleExpression will resolve to a value of integer type */
112. SingleValueOrAttrib ::= MatchingSymbol |
                             SingleExpression |
                             TemplateRefWithParList
/* STATIC SEMANTIC - VariableIdentifier (accessed via singleExpression) may only be used in in-line
template definitions to reference variables in the current scope */
113. ArrayValueOrAttrib ::= "{" ArrayElementSpecList "}"
114. ArrayElementSpecList ::= ArrayElementSpec {"," ArrayElementSpec}
115. ArrayElementSpec ::= NotUsedSymbol | PermutationMatch | TemplateBody
116. NotUsedSymbol ::= Dash
117. MatchingSymbol ::= Complement |
                        AnyValue |
                        AnyOrOmit |
                        ValueOrAttribList |
                        Range |
                        BitStringMatch |
                        HexStringMatch |
                        OctetStringMatch |
                        CharStringMatch |
                        SubsetMatch |
                        SupersetMatch
118. ExtraMatchingAttributes ::= LengthMatch | IfPresentMatch | LengthMatch  IfPresentMatch
119. BitStringMatch ::= "'" {BinOrMatch} "'" "B"
120. BinOrMatch ::= Bin | AnyValue | AnyOrOmit
121. HexStringMatch ::= "'" {HexOrMatch} "'" "H"
122. HexOrMatch ::= Hex | AnyValue | AnyOrOmit
123. OctetStringMatch ::= "'" {OctOrMatch} "'" "O"
124. OctOrMatch ::= Oct | AnyValue | AnyOrOmit
125. CharStringMatch ::= PatternKeyword Cstring
126. PatternKeyword ::= "pattern"
127. Complement ::= ComplementKeyword ValueList
128. ComplementKeyword ::= "complement"
129. ValueList ::= "(" ConstantExpression {"," ConstantExpression} ")"
130. SubsetMatch ::= SubsetKeyword ValueList
/* STATIC SEMANTIC - Subset matching shall only be used with the set of type */
131. SubsetKeyword ::= "subset"
132. SupersetMatch ::= SupersetKeyword ValueList
/* STATIC SEMANTIC - Superset matching shall only be used with the set of type */
133. SupersetKeyword ::= "superset"
134. PermutationMatch ::= PermutationKeyword PermutationList
135. PermutationKeyword ::= "permutation"
136. PermutationList ::= "(" TemplateBody { "," TemplateBody } ")"
/* STATIC SEMANTICS - Restrictions on the content of TemplateBody are given in clause B.1.3.3 */
137. AnyValue ::= "?"
138. AnyOrOmit ::= "*"
```

```
139. ValueOrAttribList ::= "(" TemplateBody {"," TemplateBody}+ ")"
140. LengthMatch ::= StringLength
141. IfPresentMatch ::= IfPresentKeyword
142. IfPresentKeyword ::= "ifpresent"
143. Range ::= "(" LowerBound ".." UpperBound ")"
144. LowerBound ::= SingleConstExpression | Minus InfinityKeyword
145. UpperBound ::= SingleConstExpression | InfinityKeyword
/* STATIC SEMANTICS - LowerBound and UpperBound shall evaluate to types integer, charstring,
universal charstring or float. In case LowerBound or UpperBound evaluates to types charstring or
universal charstring, only SingleConstExpression may be present and the string length shall be 1*/
146. InfinityKeyword ::= "infinity"
147. TemplateInstance ::= InLineTemplate
148. TemplateRefWithParList ::= [GlobalModuleId Dot] TemplateIdentifier [TemplateActualParList] |
                                TemplateParIdentifier
149. TemplateRef ::= [GlobalModuleId Dot] TemplateIdentifier | TemplateParIdentifier
150. InLineTemplate ::= [(Type | Signature) Colon] [DerivedRefWithParList AssignmentChar]
                       TemplateBody
/* STATIC SEMANTICS - The type field may only be omitted when the type is implicitly unambiguous */
151. DerivedRefWithParList ::= ModifiesKeyword TemplateRefWithParList
152. TemplateActualParList ::= "(" TemplateActualPar {"," TemplateActualPar} ")"
153. TemplateActualPar ::= TemplateInstance
/* STATIC SEMANTICS - When the corresponding formal parameter is not of template type the
TemplateInstance production shall resolve to one or more SingleExpressions */
154. TemplateOps ::= MatchOp | ValueofOp
155. MatchOp ::= MatchKeyword "(" Expression "," TemplateInstance")"
/* STATIC SEMANTICS - The type of the value returned by the expression must be the same as the
template type and each field of the template shall resolve to a single value */
156. MatchKeyword ::= "match"
157. ValueofOp ::= ValueofKeyword "(" TemplateInstance")"
158. ValueofKeyword ::= "valueof"
```

A.1.6.1.4 Function definitions

```
159. FunctionDef ::= FunctionKeyword FunctionIdentifier
                    "("[FunctionFormalParList] ")" [RunsOnSpec] [ReturnType]
                    StatementBlock
160. FunctionKeyword ::= "function"
161. FunctionIdentifier ::= Identifier
162. FunctionFormalParList ::= FunctionFormalPar {"," FunctionFormalPar}
163. FunctionFormalPar ::= FormalValuePar |
                          FormalTimerPar |
                          FormalTemplatePar |
                          FormalPortPar
164. ReturnType ::= ReturnKeyword [TemplateKeyword] Type
/* STATIC SEMANTICS - The use of the template keyword shall conform to restrictions in
clause 16.1.0 */
165. ReturnKeyword ::= "return"
166. RunsOnSpec ::= RunsKeyword OnKeyword ComponentType
167. RunsKeyword ::= "runs"
168. OnKeyword ::= "on"
169. MTCKeyword ::= "mtc"
170. StatementBlock ::= "{" [FunctionStatementOrDefList] "}"
171. FunctionStatementOrDefList ::= {FunctionStatementOrDef [SemiColon]}+
172. FunctionStatementOrDef ::= FunctionLocalDef |
                               FunctionLocalInst |
                               FunctionStatement
173. FunctionLocalInst ::= VarInstance | TimerInstance
174. FunctionLocalDef ::= ConstDef | TemplateDef
175. FunctionStatement ::= ConfigurationStatements |
                          TimerStatements |
                          CommunicationStatements |
                          BasicStatements |
                          BehaviourStatements |
                          VerdictStatements |
                          SUTStatements
176. FunctionInstance ::= FunctionRef "(" [FunctionActualParList] ")"
177. FunctionRef ::= [GlobalModuleId Dot] (FunctionIdentifier | ExtFunctionIdentifier ) |
                    PreDefFunctionIdentifier
178. PreDefFunctionIdentifier ::= Identifier
/* STATIC SEMANTICS - The Identifier will be one of the pre-defined TTCN-3 Function Identifiers from
Annex C */
179. FunctionActualParList ::= FunctionActualPar {"," FunctionActualPar}
180. 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. equivalent to the
Expression production */
```

A.1.6.1.5 Signature definitions

```
181. SignatureDef ::= SignatureKeyword SignatureIdentifier
                      "("[SignatureFormalParList] ")" [ReturnType | NoBlockKeyword]
                      [ExceptionSpec]
182. SignatureKeyword ::= "signature"
183. SignatureIdentifier ::= Identifier
184. SignatureFormalParList ::= SignatureFormalPar {"," SignatureFormalPar}
185. SignatureFormalPar ::=  FormalValuePar
186. ExceptionSpec ::= ExceptionKeyword "(" ExceptionTypeList ")"
187. ExceptionKeyword ::= "exception"
188. ExceptionTypeList ::= Type {"," Type}
189. NoBlockKeyword ::= "noblock"
190. Signature ::= [GlobalModuleId Dot] SignatureIdentifier
```

A.1.6.1.6 Testcase definitions

```
191. TestcaseDef ::= TestcaseKeyword TestcaseIdentifier
                     "("[TestcaseFormalParList] ")" ConfigSpec
                     StatementBlock
192. TestcaseKeyword ::= "testcase"
193. TestcaseIdentifier ::= Identifier
194. TestcaseFormalParList ::= TestcaseFormalPar {"," TestcaseFormalPar}
195. TestcaseFormalPar ::= FormalValuePar |
                           FormalTemplatePar
196. ConfigSpec ::= RunsOnSpec [SystemSpec]
197. SystemSpec ::= SystemKeyword ComponentType
198. SystemKeyword ::= "system"
199. TestcaseInstance ::= ExecuteKeyword "(" TestcaseRef "(" [TestcaseActualParList] ")"
                          ["," TimerValue] ")"
200. ExecuteKeyword ::= "execute"
201. TestcaseRef ::= [GlobalModuleId Dot] TestcaseIdentifier
202. TestcaseActualParList ::= TestcaseActualPar {"," TestcaseActualPar}
203. 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.1.6.1.7 Altstep definitions

```
204. AltstepDef ::= AltstepKeyword AltstepIdentifier
                    "("[AltstepFormalParList] ")" [RunsOnSpec]
                    "{" AltstepLocalDefList AltGuardList "}"
205. AltstepKeyword ::= "altstep"
206. AltstepIdentifier ::= Identifier
207. AltstepFormalParList ::= FunctionFormalParList
/* STATIC SEMANTICS - altsteps that are activated as defaults shall only have in parameters, port
parameters, or timer parameters */
/* STATIC SEMANTICS - altsteps that are only invoked as an alternative in an alt statement or as
stand-alone statement in a TTCN-3 behaviour description may have in, out and inout parameters. */
208. AltstepLocalDefList ::= {AltstepLocalDef [SemiColon]}
209. AltstepLocalDef ::= VarInstance | TimerInstance | ConstDef | TemplateDef
/*STATIC SEMANTICS - AltstepLocalDef shall conform to restrictions in clause 16.2.2.1 */
210. AltstepInstance ::= AltstepRef "(" [FunctionActualParList] ")"
/* STATIC SEMANTICS - all timer instances in FunctionActualParList shall be declared as component
local timers (see also production ComponentElementDef) */
211. AltstepRef ::= [GlobalModuleId Dot] AltstepIdentifier
```

A.1.6.1.8 Import definitions

```
212. ImportDef ::= ImportKeyword ImportFromSpec (AllWithExcepts | ("{" ImportSpec "}"))
213. ImportKeyword ::= "import"
214. AllWithExcepts ::= AllKeyword [ExceptsDef]
215. ExceptsDef ::= ExceptKeyword "{" ExceptSpec "}"
216. ExceptKeyword ::= "except"
217. ExceptSpec ::= {ExceptElement [SemiColon]}
/* STATIC SEMANTICS - Any of the production components (ExceptGroupSpec, ExceptTypeDefSpec etc.) may
be present only once in the ExceptSpec production */
218. ExceptElement ::= ExceptGroupSpec |
                       ExceptTypeDefSpec |
                       ExceptTemplateSpec |
                       ExceptConstSpec |
                       ExceptTestcaseSpec |
                       ExceptAltstepSpec |
                       ExceptFunctionSpec |
                       ExceptSignatureSpec |
                       ExceptModuleParSpec
219. ExceptGroupSpec ::= GroupKeyword (ExceptGroupRefList | AllKeyword)
220. ExceptTypeDefSpec ::= TypeDefKeyword (TypeRefList | AllKeyword)
```

```
221. ExceptTemplateSpec ::= TemplateKeyword (TemplateRefList | AllKeyword)
222. ExceptConstSpec ::= ConstKeyword (ConstRefList | AllKeyword)
223. ExceptTestcaseSpec ::= TestcaseKeyword (TestcaseRefList | AllKeyword)
224. ExceptAltstepSpec ::= AltstepKeyword (AltstepRefList | AllKeyword)
225. ExceptFunctionSpec ::= FunctionKeyword (FunctionRefList | AllKeyword)
226. ExceptSignatureSpec ::= SignatureKeyword  (SignatureRefList | AllKeyword)
227. ExceptModuleParSpec ::= ModuleParKeyword  (ModuleParRefList | AllKeyword)
228. ImportSpec ::= {ImportElement [SemiColon]}
229. ImportElement ::= ImportGroupSpec |
                       ImportTypeDefSpec |
                       ImportTemplateSpec |
                       ImportConstSpec |
                       ImportTestcaseSpec |
                       ImportAltstepSpec |
                       ImportFunctionSpec |
                       ImportSignatureSpec |
                       ImportModuleParSpec
230. ImportFromSpec ::= FromKeyword ModuleId [RecursiveKeyword]
/* NOTE - The use of RecursiveKeyword is deprecated*/
231. RecursiveKeyword ::= "recursive"
232. ImportGroupSpec ::= GroupKeyword (GroupRefListWithExcept | AllGroupsWithExcept)
233. GroupRefList ::= FullGroupIdentifier {"," FullGroupIdentifier}
234. GroupRefListWithExcept ::= FullGroupIdentifierWithExcept {"," FullGroupIdentifierWithExcept}
235. AllGroupsWithExcept ::= AllKeyword [ExceptKeyword GroupRefList]
236. FullGroupIdentifier ::= GroupIdentifier {Dot GroupIdentifier}
237. FullGroupIdentifierWithExcept ::= FullGroupIdentifier [ExceptsDef]
238. ExceptGroupRefList ::= ExceptFullGroupIdentifier {"," ExceptFullGroupIdentifier}
239. ExceptFullGroupIdentifier ::= FullGroupIdentifier
240. ImportTypeDefSpec ::= TypeDefKeyword (TypeRefList | AllTypesWithExcept)
241. TypeRefList ::= TypeDefIdentifier {"," TypeDefIdentifier}
242. AllTypesWithExcept ::= AllKeyword [ExceptKeyword TypeRefList]
243. TypeDefIdentifier ::= StructTypeIdentifier |
                           EnumTypeIdentifier |
                           PortTypeIdentifier |
                           ComponentTypeIdentifier |
                           SubTypeIdentifier
244. ImportTemplateSpec ::= TemplateKeyword (TemplateRefList | AllTemplsWithExcept)
245. TemplateRefList ::= TemplateIdentifier {"," TemplateIdentifier}
246. AllTemplsWithExcept ::= AllKeyword [ExceptKeyword TemplateRefList]
247. ImportConstSpec ::= ConstKeyword (ConstRefList | AllConstsWithExcept)
248. ConstRefList ::= ConstIdentifier {"," ConstIdentifier}
249. AllConstsWithExcept ::= AllKeyword [ExceptKeyword ConstRefList]
250. ImportAltstepSpec ::= AltstepKeyword (AltstepRefList | AllAltstepsWithExcept)
251. AltstepRefList ::= AltstepIdentifier {"," AltstepIdentifier}
252. AllAltstepsWithExcept ::= AllKeyword [ExceptKeyword AltstepRefList]
253. ImportTestcaseSpec ::= TestcaseKeyword (TestcaseRefList | AllTestcasesWithExcept)
254. TestcaseRefList ::= TestcaseIdentifier {"," TestcaseIdentifier}
255. AllTestcasesWithExcept ::= AllKeyword [ExceptKeyword TestcaseRefList]
256. ImportFunctionSpec ::= FunctionKeyword (FunctionRefList | AllFunctionsWithExcept)
257. FunctionRefList ::= FunctionIdentifier {"," FunctionIdentifier}
258. AllFunctionsWithExcept ::= AllKeyword [ExceptKeyword FunctionRefList]
259. ImportSignatureSpec ::= SignatureKeyword  (SignatureRefList | AllSignaturesWithExcept)
260. SignatureRefList ::= SignatureIdentifier {"," SignatureIdentifier}
261. AllSignaturesWithExcept ::= AllKeyword [ExceptKeyword SignatureRefList]
262. ImportModuleParSpec ::= ModuleParKeyword  (ModuleParRefList | AllModuleParWithExcept)
263. ModuleParRefList ::= ModuleParIdentifier {"," ModuleParIdentifier}
264. AllModuleParWithExcept ::= AllKeyword [ExceptKeyword ModuleParRefList]
```

A.1.6.1.9 Group definitions

```
265. GroupDef ::= GroupKeyword GroupIdentifier
                  "{" [ModuleDefinitionsPart] "}"
266. GroupKeyword ::= "group"
267. GroupIdentifier ::= Identifier
```

A.1.6.1.10 External function definitions

```
268. ExtFunctionDef ::= ExtKeyword FunctionKeyword ExtFunctionIdentifier
                        "("[FunctionFormalParList] ")" [ReturnType]
269. ExtKeyword ::= "external"
270. ExtFunctionIdentifier ::= Identifier
```

A.1.6.1.11 External constant definitions

```
271. ExtConstDef ::= ExtKeyword ConstKeyword Type ExtConstIdentifier
/* STATIC SEMANTICS - Type shall follow the rules given in clause 9.*/
272. ExtConstIdentifier ::= Identifier
```

A.1.6.1.12 Module parameter definitions

```
273. ModuleParDef ::= ModuleParKeyword ( ModulePar | ("{" MultitypedModuleParList "}"))
274. ModuleParKeyword ::= "modulepar"
275. MultitypedModuleParList ::= { ModulePar SemiColon }+
276. ModulePar ::= ModuleParType ModuleParList
/* STATIC SEMANTICS - The Value of the ConstantExpression shall be of the same type as the stated
type for the Parameter */
277. ModuleParType ::= Type
/* STATIC SEMANTICS - Type shall not be of component, default or anytype. Type shall only resolve to
address type if a definition for the address type is defined within the module */
278. ModuleParList ::= ModuleParIdentifier [AssignmentChar ConstantExpression]
                      {","ModuleParIdentifier [AssignmentChar ConstantExpression]}
279. ModuleParIdentifier ::= Identifier
```

A.1.6.2 Control part

A.1.6.2.0 General

```
280. ModuleControlPart ::= ControlKeyword
                           "{" ModuleControlBody "}"
                           [WithStatement] [SemiColon]
281. ControlKeyword ::= "control"
282. ModuleControlBody ::= [ControlStatementOrDefList]
283. ControlStatementOrDefList ::= {ControlStatementOrDef [SemiColon]}+
284. ControlStatementOrDef ::= FunctionLocalDef |
                               FunctionLocalInst |
                               ControlStatement
285. ControlStatement ::= TimerStatements |
                          BasicStatements |
                          BehaviourStatements |
                          SUTStatements |
                          StopKeyword
/* STATIC SEMANTICS - Restrictions on use of statements in the control part are given in Table 11 */
```

A.1.6.2.1 Variable instantiation

```
286. VarInstance ::= VarKeyword ((Type VarList) | (TemplateKeyword Type TempVarList))
287. VarList ::= SingleVarInstance {"," SingleVarInstance}
288. SingleVarInstance ::= VarIdentifier [ArrayDef] [AssignmentChar VarInitialValue]
289. VarInitialValue ::= Expression
290. VarKeyword ::= "var"
291. VarIdentifier ::= Identifier
292. TempVarList ::= SingleTempVarInstance {"," SingleTempVarInstance}
293. SingleTempVarInstance ::= VarIdentifier [ArrayDef] [AssignmentChar TempVarInitialValue]
294. TempVarInitialValue ::= TemplateBody
295. VariableRef ::= (VarIdentifier | ValueParIdentifier) [ExtendedFieldReference]
```

A.1.6.2.2 Timer instantiation

```
296. TimerInstance ::= TimerKeyword TimerList
297. TimerList ::= SingleTimerInstance{"," SingleTimerInstance}
298. SingleTimerInstance ::= TimerIdentifier [ArrayDef] [AssignmentChar TimerValue]
299. TimerKeyword ::= "timer"
300. TimerIdentifier ::= Identifier
301. TimerValue ::= Expression
/* STATIC SEMANTICS - When Expression resolves to SingleExpression it must resolve to a value of
type float. Expression shall only resolve to CompoundExpression in the initialization in default
timer value assignment for timer arrays */
302. TimerRef ::= (TimerIdentifier | TimerParIdentifier) {ArrayOrBitRef}
```

A.1.6.2.3 Component operations

```
303. ConfigurationStatements ::= ConnectStatement |
                                 MapStatement |
                                 DisconnectStatement |
                                 UnmapStatement |
                                 DoneStatement |
                                 KilledStatement |
                                 StartTCStatement |
                                 StopTCStatement |
                                 KillTCStatement
304. ConfigurationOps ::= CreateOp | SelfOp | SystemOp | MTCOp | RunningOp | AliveOp
305. CreateOp ::= ComponentType Dot CreateKeyword ["(" SingleExpression ")"] [AliveKeyword]
/* STATIC SEMANTICS - Restrictions on SingleExpression see in clause 22.1 */
306. SystemOp ::= SystemKeyword
307. SelfOp ::= "self"
308. MTCOp ::= MTCKeyword
```

```
309. DoneStatement ::= ComponentId Dot DoneKeyword
310. KilledStatement ::= ComponentId Dot KilledKeyword
311. ComponentId ::= ComponentOrDefaultReference | (AnyKeyword | AllKeyword) ComponentKeyword
312. DoneKeyword ::= "done"
313. KilledKeyword ::= "killed"
314. RunningOp ::= ComponentId Dot RunningKeyword
315. RunningKeyword ::= "running"
316. AliveOp ::= ComponentId Dot AliveKeyword
317. CreateKeyword ::= "create"
318. AliveKeyword ::= "alive"
319. ConnectStatement ::= ConnectKeyword SingleConnectionSpec
320. ConnectKeyword ::= "connect"
321. SingleConnectionSpec ::= "(" PortRef "," PortRef ")"
322. PortRef ::= ComponentRef Colon Port
323. ComponentRef ::= ComponentOrDefaultReference | SystemOp | SelfOp | MTCOp
324. DisconnectStatement ::= DisconnectKeyword [SingleOrMultiConnectionSpec]
325. SingleOrMultiConnectionSpec ::= SingleConnectionSpec |
                                    AllConnectionsSpec |
                                    AllPortsSpec |
                                    AllCompsAllPortsSpec]
326. AllConnectionsSpec ::= "(" PortRef ")"
327. AllPortsSpec ::= "(" ComponentRef ":" AllKeyword PortKeyword ")"
328. AllCompsAllPortsSpec ::= "(" AllKeyword ComponentKeyword ":" AllKeyword PortKeyword ")"
329. DisconnectKeyword ::= "disconnect"
330. MapStatement ::= MapKeyword SingleConnectionSpec
331. MapKeyword ::= "map"
332. UnmapStatement ::= UnmapKeyword [SingleOrMultiConnectionSpec]
333. UnmapKeyword ::= "unmap"
334. StartTCStatement ::= ComponentOrDefaultReference Dot StartKeyword "(" FunctionInstance ")"
/* STATIC SEMANTICS - the Function instance may only have in parameters */
/* STATIC SEMANTICS - the Function instance shall not have timer parameters */
335. StartKeyword ::= "start"
336. StopTCStatement ::= StopKeyword | (ComponentReferenceOrLiteral Dot StopKeyword) |
                       (AllKeyword ComponentKeyword Dot StopKeyword)
337. ComponentReferenceOrLiteral ::= ComponentOrDefaultReference | MTCOp | SelfOp
338. KillTCStatement ::= KillKeyword | (ComponentIdentifierOrLiteral Dot KillKeyword) |
                       (AllKeyword ComponentKeyword Dot KillKeyword)
339. ComponentOrDefaultReference ::= VariableRef | FunctionInstance
/* STATIC SEMANTICS - The variable associated with VariableRef or the return type associated with
FunctionInstance must be of component type when used in configuration statements and shall be of
default type when used in the deactivate statement. */
340. KillKeyword ::= "kill"
```

A.1.6.2.4 Port operations

```
341. Port ::= (PortIdentifier | PortParIdentifier) {ArrayOrBitRef}
342. CommunicationStatements ::= SendStatement |
                                CallStatement |
                                ReplyStatement |
                                RaiseStatement |
                                ReceiveStatement |
                                TriggerStatement |
                                GetCallStatement |
                                GetReplyStatement |
                                CatchStatement |
                                CheckStatement |
                                ClearStatement |
                                StartStatement |
                                StopStatement
343. SendStatement ::= Port Dot PortSendOp
344. PortSendOp ::= SendOpKeyword "(" SendParameter ")" [ToClause]
345. SendOpKeyword ::= "send"
346. SendParameter ::= TemplateInstance
347. ToClause ::= ToKeyword AddressRef |
                 AddressRefList |
                 AllKeyword ComponentKeyword
/* STATIC SEMANTICS - AddressRef should not contain matching mechanisms */
348. AddressRefList ::= "(" AddressRef {"," AddressRef} ")"
349. ToKeyword ::= "to"
350. AddressRef ::= TemplateInstance
/* STATIC SEMANTICS - TemplateInstance must be of address or component type */
351. CallStatement ::= Port Dot PortCallOp  [PortCallBody]
352. PortCallOp ::= CallOpKeyword "(" CallParameters ")" [ToClause]
353. CallOpKeyword ::= "call"
354. CallParameters ::= TemplateInstance ["," CallTimerValue]
/* STATIC SEMANTICS - only out parameters may be omitted or specified with a matching attribute */
355. CallTimerValue ::= TimerValue | NowaitKeyword
/* STATIC SEMANTICS - Value must be of type float */
356. NowaitKeyword ::= "nowait"
357. PortCallBody ::= "{" CallBodyStatementList "}"
```

```
358. CallBodyStatementList ::= {CallBodyStatement [SemiColon]}+
359. CallBodyStatement ::= CallBodyGuard  StatementBlock
360. CallBodyGuard ::= AltGuardChar CallBodyOps
361. CallBodyOps ::= GetReplyStatement | CatchStatement
362. ReplyStatement ::= Port Dot PortReplyOp
363. PortReplyOp ::= ReplyKeyword "(" TemplateInstance [ReplyValue]")" [ToClause]
364. ReplyKeyword ::= "reply"
365. ReplyValue ::= ValueKeyword Expression
366. RaiseStatement ::= Port Dot PortRaiseOp
367. PortRaiseOp ::= RaiseKeyword "(" Signature "," TemplateInstance ")" [ToClause]
368. RaiseKeyword ::= "raise"
369. ReceiveStatement ::= PortOrAny Dot PortReceiveOp
370. PortOrAny ::= Port | AnyKeyword PortKeyword
371. PortReceiveOp ::= ReceiveOpKeyword ["(" ReceiveParameter ")"] [FromClause] [PortRedirect]
/* STATIC SEMANTICS - the PortRedirect option may only be present if the ReceiveParameter option is
also present */
372. ReceiveOpKeyword ::= "receive"
373. ReceiveParameter ::= TemplateInstance
374. FromClause ::= FromKeyword AddressRef
375. FromKeyword ::= "from"
376. PortRedirect ::= PortRedirectSymbol (ValueSpec [SenderSpec] | SenderSpec)
377. PortRedirectSymbol ::= "->"
378. ValueSpec ::= ValueKeyword VariableRef
379. ValueKeyword ::= "value"
380. SenderSpec ::= SenderKeyword VariableRef
/* STATIC SEMANTICS - Variable ref must be of address or component type */
381. SenderKeyword ::= "sender"
382. TriggerStatement ::= PortOrAny Dot PortTriggerOp
383. PortTriggerOp ::= TriggerOpKeyword ["(" ReceiveParameter ")"] [FromClause] [PortRedirect]
/* STATIC SEMANTICS - the PortRedirect option may only be present if the ReceiveParameter option is
also present */
384. TriggerOpKeyword ::= "trigger"
385. GetCallStatement ::= PortOrAny  Dot PortGetCallOp
386. PortGetCallOp ::= GetCallOpKeyword ["(" ReceiveParameter ")"] [FromClause]
                       [PortRedirectWithParam]
/* STATIC SEMANTICS - the PortRedirectWithParam option may only be present if the ReceiveParameter
option is also present */
387. GetCallOpKeyword ::= "getcall"
388. PortRedirectWithParam ::= PortRedirectSymbol RedirectWithParamSpec
389. RedirectWithParamSpec ::= ParamSpec [SenderSpec] |
                               SenderSpec
390. ParamSpec ::= ParamKeyword ParamAssignmentList
391. ParamKeyword ::= "param"
392. ParamAssignmentList ::= "(" (AssignmentList | VariableList) ")"
393. AssignmentList ::= VariableAssignment {"," VariableAssignment}
394. VariableAssignment ::= VariableRef AssignmentChar ParameterIdentifier
/* STATIC SEMANTICS - the parameterIdentifiers must be from the corresponding signature
definition */
395. ParameterIdentifier ::= ValueParIdentifier
396. VariableList ::= VariableEntry  {"," VariableEntry}
397. VariableEntry ::= VariableRef | NotUsedSymbol
398. GetReplyStatement ::= PortOrAny Dot PortGetReplyOp
399. PortGetReplyOp ::= GetReplyOpKeyword ["(" ReceiveParameter [ValueMatchSpec] ")"]
                        [FromClause] [PortRedirectWithValueAndParam]
/* STATIC SEMANTICS - the PortRedirectWithParam option may only be present if the ReceiveParameter
option is also present */
400. PortRedirectWithValueAndParam ::= PortRedirectSymbol RedirectWithValueAndParamSpec
401. RedirectWithValueAndParamSpec ::= ValueSpec [ParamSpec] [SenderSpec] |
                                       RedirectWithParamSpec
402. GetReplyOpKeyword ::= "getreply"
403. ValueMatchSpec ::= ValueKeyword TemplateInstance
404. CheckStatement ::= PortOrAny Dot PortCheckOp
405. PortCheckOp ::= CheckOpKeyword ["(" CheckParameter ")"]
406. CheckOpKeyword ::= "check"
407. CheckParameter ::= CheckPortOpsPresent | FromClausePresent | RedirectPresent
408. FromClausePresent ::= FromClause [PortRedirectSymbol SenderSpec]
409. RedirectPresent ::= PortRedirectSymbol SenderSpec
410. CheckPortOpsPresent ::= PortReceiveOp | PortGetCallOp | PortGetReplyOp | PortCatchOp
411. CatchStatement ::= PortOrAny Dot PortCatchOp
412. PortCatchOp ::= CatchOpKeyword ["("CatchOpParameter ")"] [FromClause] [PortRedirect]
/* STATIC SEMANTICS - the PortRedirect option may only be present if the CatchOpParameter option is
also present */
413. CatchOpKeyword ::= "catch"
414. CatchOpParameter ::= Signature "," TemplateInstance | TimeoutKeyword
415. ClearStatement ::= PortOrAll Dot PortClearOp
416. PortOrAll ::= Port | AllKeyword PortKeyword
417. PortClearOp ::= ClearOpKeyword
418. ClearOpKeyword ::= "clear"
419. StartStatement ::= PortOrAll Dot PortStartOp
420. PortStartOp ::= StartKeyword
```

```
421. StopStatement ::= PortOrAll Dot PortStopOp
422. PortStopOp ::= StopKeyword
423. StopKeyword ::= "stop"
424. AnyKeyword ::= "any"
```

A.1.6.2.5 Timer operations

```
425. TimerStatements ::= StartTimerStatement | StopTimerStatement | TimeoutStatement
426. TimerOps ::= ReadTimerOp | RunningTimerOp
427. StartTimerStatement ::= TimerRef Dot StartKeyword ["(" TimerValue ")"]
428. StopTimerStatement ::= TimerRefOrAll Dot StopKeyword
429. TimerRefOrAll ::= TimerRef | AllKeyword TimerKeyword
430. ReadTimerOp ::= TimerRef Dot ReadKeyword
431. ReadKeyword ::= "read"
432. RunningTimerOp ::= TimerRefOrAny  Dot RunningKeyword
433. TimeoutStatement ::= TimerRefOrAny  Dot TimeoutKeyword
434. TimerRefOrAny ::= TimerRef | AnyKeyword TimerKeyword
435. TimeoutKeyword ::= "timeout"
```

A.1.6.3 Type

```
436. Type ::= PredefinedType | ReferencedType
437. PredefinedType ::= BitStringKeyword |
                        BooleanKeyword |
                        CharStringKeyword |
                        UniversalCharString |
                        IntegerKeyword |
                        OctetStringKeyword |
                        HexStringKeyword |
                        VerdictTypeKeyword |
                        FloatKeyword |
                        AddressKeyword |
                        DefaultKeyword |
                        AnyTypeKeyword
438. BitStringKeyword ::= "bitstring"
439. BooleanKeyword ::= "boolean"
440. IntegerKeyword ::= "integer"
441. OctetStringKeyword ::= "octetstring"
442. HexStringKeyword ::= "hexstring"
443. VerdictTypeKeyword ::= "verdicttype"
444. FloatKeyword ::= "float"
445. AddressKeyword ::= "address"
446. DefaultKeyword ::= "default"
447. AnyTypeKeyword ::= "anytype"
448. CharStringKeyword ::= "charstring"
449. UniversalCharString ::= UniversalKeyword CharStringKeyword
450. UniversalKeyword ::= "universal"
451. ReferencedType ::= [GlobalModuleId Dot] TypeReference [ExtendedFieldReference]
452. TypeReference ::= StructTypeIdentifier[TypeActualParList] |
                       EnumTypeIdentifier |
                       SubTypeIdentifier |
                       ComponentTypeIdentifier
453. TypeActualParList ::= "(" TypeActualPar {"," TypeActualPar} ")"
454. TypeActualPar ::= ConstantExpression
455. ArrayDef ::= {"[" ArrayBounds [".." ArrayBounds] "]"}+
456. ArrayBounds ::= SingleConstExpression
/* STATIC SEMANTICS - ArrayBounds will resolve to a non-negative value of integer type */
```

A.1.6.4 Value

```
457. Value ::= PredefinedValue | ReferencedValue
458. PredefinedValue ::= BitStringValue |
                         BooleanValue |
                         CharStringValue |
                         IntegerValue |
                         OctetStringValue |
                         HexStringValue |
                         VerdictTypeValue |
                         EnumeratedValue |
                         FloatValue |
                         AddressValue |
                         OmitValue
459. BitStringValue ::= Bstring
460. BooleanValue ::= "true" | "false"
461. IntegerValue ::= Number
462. OctetStringValue ::= Ostring
463. HexStringValue ::= Hstring
464. VerdictTypeValue ::= "pass" | "fail" | "inconc" | "none" | "error"
465. EnumeratedValue ::= EnumerationIdentifier
```

```
466. CharStringValue ::= Cstring | Quadruple
467. Quadruple ::= CharKeyword "(" Group "," Plane "," Row "," Cell ")"
468. CharKeyword ::= "char"
469. Group ::= Number
470. Plane ::= Number
471. Row ::= Number
472. Cell ::= Number
473. FloatValue ::= FloatDotNotation | FloatENotation
474. FloatDotNotation ::= Number Dot DecimalNumber
475. FloatENotation ::= Number [Dot DecimalNumber] Exponential [Minus] Number
476. Exponential ::= "E"
477. ReferencedValue ::= ValueReference [ExtendedFieldReference]
478. ValueReference ::= [GlobalModuleId Dot] (ConstIdentifier | ExtConstIdentifier |
                        ModuleParIdentifier ) |
                        ValueParIdentifier |
                        VarIdentifier
479. Number ::= (NonZeroNum {Num}) | "0"
480. NonZeroNum ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
481. DecimalNumber ::= {Num}+
482. Num ::= "0" | NonZeroNum
483. Bstring ::= "'" {Bin} "'" "B"
484. Bin ::= "0" | "1"
485. Hstring ::= "'" {Hex} "'" "H"
486. Hex ::= Num | "A" | "B" | "C" | "D" | "E" | "F"| "a" | "b" | "c" | "d" | "e" | "f"
487. Ostring ::= "'" {Oct} "'" "O"
488. Oct ::= Hex Hex
489. Cstring ::= """ {Char} """
490. Char ::= /* REFERENCE - A character defined by the relevant CharacterString type. For
charstring a character from the character set defined in ITU-T Rec. T.50. For universal charstring a
character from any character set defined in ISO/IEC 10646 */
491. Identifier ::= Alpha{AlphaNum | Underscore}
492. Alpha ::= UpperAlpha | LowerAlpha
493. AlphaNum ::= Alpha | Num
494. 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"
495. 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"
496. ExtendedAlphaNum ::= /* REFERENCE - A graphical character from the BASIC LATIN or from the
LATIN-1 SUPPLEMENT character sets defined in ISO/IEC 10646  (characters from char (0,0,0,32) to char
(0,0,0,126), from char (0,0,0,161) to char (0,0,0,172) and from char (0,0,0,174) to char
(0,0,0,255)) */
497. FreeText ::= """ {ExtendedAlphaNum} """
498. AddressValue ::= "null"
499. OmitValue ::= OmitKeyword
500. OmitKeyword ::= "omit"
```

A.1.6.5 Parameterization

```
501. InParKeyword ::= "in"
502. OutParKeyword ::= "out"
503. InOutParKeyword ::= "inout"
504. FormalValuePar ::= [(InParKeyword | InOutParKeyword | OutParKeyword)] Type ValueParIdentifier
505. ValueParIdentifier ::= Identifier
506. FormalPortPar ::= [InOutParKeyword] PortTypeIdentifier PortParIdentifier
507. PortParIdentifier ::= Identifier
508. FormalTimerPar ::= [InOutParKeyword] TimerKeyword TimerParIdentifier
509. TimerParIdentifier ::= Identifier
510. FormalTemplatePar ::= [(InParKeyword | OutParKeyword | InOutParKeyword )]
                           TemplateKeyword Type TemplateParIdentifier
511. TemplateParIdentifier ::= Identifier
```

A.1.6.6 With statement

```
512. WithStatement ::= WithKeyword WithAttribList
513. WithKeyword ::= "with"
514. WithAttribList ::= "{" MultiWithAttrib "}"
515. MultiWithAttrib ::= {SingleWithAttrib [SemiColon]}
516. SingleWithAttrib ::= AttribKeyword  [OverrideKeyword] [AttribQualifier] AttribSpec
517. AttribKeyword ::= EncodeKeyword |
                       VariantKeyword |
                       DisplayKeyword |
                       ExtensionKeyword
518. EncodeKeyword ::= "encode"
519. VariantKeyword ::= "variant"
520. DisplayKeyword ::= "display"
521. ExtensionKeyword ::= "extension"
522. OverrideKeyword ::= "override"
523. AttribQualifier ::= "(" DefOrFieldRefList ")"
524. DefOrFieldRefList ::= DefOrFieldRef {"," DefOrFieldRef}
```

```
525. DefOrFieldRef ::= DefinitionRef | FieldReference | AllRef
/* STATIC SEMANTICS - the DefOrFieldRef must refer to a definition or field which is within the
module, group or definition to which the with statement is associated */
526. DefinitionRef ::= StructTypeIdentifier |
                       EnumTypeIdentifier |
                       PortTypeIdentifier |
                       ComponentTypeIdentifier |
                       SubTypeIdentifier |
                       ConstIdentifier |
                       TemplateIdentifier |
                       AltstepIdentifier |
                       TestcaseIdentifier |
                       FunctionIdentifier |
                       SignatureIdentifier |
                       VarIdentifier |
                       TimerIdentifier |
                       PortIdentifier |
                       ModuleParIdentifier |
                       FullGroupIdentifier
527. AllRef ::= ( GroupKeyword AllKeyword [ExceptKeyword "{" GroupRefList "}"]) |
                ( TypeDefKeyword AllKeyword [ExceptKeyword "{" TypeRefList "}"]) |
                ( TemplateKeyword AllKeyword [ExceptKeyword "{" TemplateRefList "}"]) |
                ( ConstKeyword AllKeyword [ExceptKeyword "{" ConstRefList "}"]) |
                ( AltstepKeyword AllKeyword [ExceptKeyword "{" AltstepRefList "}"]) |
                ( TestcaseKeyword AllKeyword [ExceptKeyword "{" TestcaseRefList "}"]) |
                ( FunctionKeyword AllKeyword [ExceptKeyword "{" FunctionRefList "}"]) |
                ( SignatureKeyword AllKeyword [ExceptKeyword "{" SignatureRefList "}"]) |
                ( ModuleParKeyword AllKeyword [ExceptKeyword "{" ModuleParRefList "}"])
528. AttribSpec ::= FreeText
```

A.1.6.7 Behaviour statements

```
529. BehaviourStatements ::= TestcaseInstance |
                             FunctionInstance |
                             ReturnStatement |
                             AltConstruct |
                             InterleavedConstruct |
                             LabelStatement |
                             GotoStatement |
                             RepeatStatement |
                             DeactivateStatement |
                             AltstepInstance |
                             ActivateOp
/* 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 */
/* STATIC SEMANTICS - ActivateOp shall not be called from within the module control part */
530. VerdictStatements ::= SetLocalVerdict
531. VerdictOps ::= GetLocalVerdict
532. SetLocalVerdict ::= SetVerdictKeyword "(" SingleExpression ")"
/* STATIC SEMANTICS - SingleExpression must resolve to a value of type verdict */
/* STATIC SEMANTICS - the SetLocalVerdict shall not be used to assign the Value error */
533. SetVerdictKeyword ::= "setverdict"
534. GetLocalVerdict ::= "getverdict"
535. SUTStatements ::= ActionKeyword "(" [ActionText ] {StringOp ActionText} ")"
536. ActionKeyword ::= "action"
537. ActionText ::= FreeText | Expression
/*STATIC SEMANTICS - Expression shall have the base type charstring or universal charstring */
538. ReturnStatement ::= ReturnKeyword  [Expression]
539. AltConstruct ::= AltKeyword "{" AltGuardList "}"
540. AltKeyword ::= "alt"
541. AltGuardList ::= {GuardStatement | ElseStatement [SemiColon]}
542. GuardStatement ::= AltGuardChar (AltstepInstance [StatementBlock] | GuardOp StatementBlock)
543. ElseStatement ::= "["ElseKeyword "]" StatementBlock
544. AltGuardChar ::= "[" [BooleanExpression] "]"
/*STATIC SEMANTICS - BooleanExpression shall conform to restrictions in clause 20.1.2 */
545. GuardOp ::= TimeoutStatement |
                 ReceiveStatement |
                 TriggerStatement |
                 GetCallStatement |
                 CatchStatement |
                 CheckStatement |
                 GetReplyStatement |
                 DoneStatement |
                 KilledStatement
/* STATIC SEMANTICS - GuardOp used within the module control part shall only contain the
timeoutStatement */
546. InterleavedConstruct ::= InterleavedKeyword "{" InterleavedGuardList "}"
547. InterleavedKeyword ::= "interleave"
548. InterleavedGuardList ::= {InterleavedGuardElement [SemiColon]}+
```

```
549. InterleavedGuardElement ::= InterleavedGuard InterleavedAction
550. InterleavedGuard ::= "[" "]" GuardOp
551. InterleavedAction ::= StatementBlock
/* STATIC SEMANTICS - the StatementBlock may not contain loop statements, goto, activate,
deactivate, stop, return or calls to functions */
552. LabelStatement ::= LabelKeyword LabelIdentifier
553. LabelKeyword ::= "label"
554. LabelIdentifier ::= Identifier
555. GotoStatement ::= GotoKeyword LabelIdentifier
556. GotoKeyword ::= "goto"
557. RepeatStatement ::= "repeat"
558. ActivateOp ::= ActivateKeyword "(" AltstepInstance ")"
559. ActivateKeyword ::= "activate"
560. DeactivateStatement ::= DeactivateKeyword ["(" ComponentOrDefaultReference ")"]
561. DeactivateKeyword ::= "deactivate"
```

A.1.6.8 Basic statements

```
562. BasicStatements ::= Assignment | LogStatement | LoopConstruct | ConditionalConstruct |
                         SelectCaseConstruct
563. Expression ::= SingleExpression | CompoundExpression
/* STATIC SEMANTICS - Expression shall not contain Configuration, activate operation or verdict
operations within the module control part */
564. CompoundExpression ::= FieldExpressionList | ArrayExpression
/* STATIC SEMANTICS - Within CompoundExpression the ArrayExpression can be used for Arrays, record,
record of and set of types. */
565. FieldExpressionList ::= "{" FieldExpressionSpec {"," FieldExpressionSpec} "}"
566. FieldExpressionSpec ::= FieldReference AssignmentChar NotUsedOrExpression
567. ArrayExpression ::= "{" [ArrayElementExpressionList] "}"
568. ArrayElementExpressionList ::= NotUsedOrExpression  {"," NotUsedOrExpression}
569. NotUsedOrExpression ::= Expression | NotUsedSymbol
570. ConstantExpression ::= SingleConstExpression | CompoundConstExpression
571. SingleConstExpression ::= SingleExpression
/* STATIC SEMANTICS - SingleConstExpression shall not contain Variables or Module parameters and
shall resolve to a constant Value at compile time */
572. BooleanExpression ::= SingleExpression
/* STATIC SEMANTICS - BooleanExpression shall resolve to a Value of type Boolean */
573. CompoundConstExpression ::= FieldConstExpressionList | ArrayConstExpression
/* STATIC SEMANTICS - Within CompoundConstExpression the ArrayConstExpression can be used for
Arrays, record, record of and set of types. */
574. FieldConstExpressionList ::= "{" FieldConstExpressionSpec {"," FieldConstExpressionSpec} "}"
575. FieldConstExpressionSpec ::= FieldReference AssignmentChar ConstantExpression
576. ArrayConstExpression ::= "{" [ArrayElementConstExpressionList] "}"
577. ArrayElementConstExpressionList ::= ConstantExpression {"," ConstantExpression}
578. Assignment ::= VariableRef AssignmentChar (Expression | TemplateBody)
/* STATIC SEMANTICS - The Expression on the right hand side of Assignment shall evaluate to an
explicit Value of a type compatible with the type of the left hand side for value variables and
shall evaluate to an explicit Value, template (literal or a template instance) or a matching
mechanism compatible with the type of the left hand side for template variables. */
579. SingleExpression ::= XorExpression { "or" XorExpression }
/* STATIC SEMANTICS - If more than one XorExpression exists, then the XorExpressions shall evaluate
to specific values of compatible types */
580. XorExpression ::= AndExpression { "xor" AndExpression }
/* STATIC SEMANTICS - If more than one AndExpression exists, then the AndExpressions shall evaluate
to specific values of compatible types */
581. AndExpression ::= NotExpression { "and" NotExpression }
/* STATIC SEMANTICS - If more than one NotExpression exists, then the NotExpressions shall evaluate
to specific values of compatible types */
582. NotExpression ::= [ "not" ] EqualExpression
/* STATIC SEMANTICS - Operands of the not operator shall be of type boolean (TTCN or ASN.1) or
derivatives of type Boolean. */
583. EqualExpression ::= RelExpression { EqualOp RelExpression }
/* STATIC SEMANTICS - If more than one RelExpression exists, then the RelExpressions shall evaluate
to specific values of compatible types */
584. RelExpression ::= ShiftExpression [ RelOp ShiftExpression ]
/* STATIC SEMANTICS - If both ShiftExpressions exist, then each ShiftExpression shall evaluate to a
specific integer, Enumerated or float Value (these values can either be TTCN or ASN.1 values) or
derivatives of these types */
585. ShiftExpression ::= BitOrExpression { ShiftOp BitOrExpression }
/* STATIC SEMANTICS - Each Result shall resolve to a specific Value. If more than one Result exists
the right-hand operand shall be of type integer or derivatives and if the shift op is '<<' or '>>'
then the left-hand operand shall resolve to either bitstring, hexstring or octetstring type or
derivatives of these types. If the shift op is '<@' or '@>' then the left-hand operand shall be of
type bitstring, hexstring, charstring or universal charstring or derivatives of these types */
586. BitOrExpression ::= BitXorExpression { "or4b" BitXorExpression }
/* STATIC SEMANTICS - If more than one BitXorExpression exists, then the BitXorExpressions shall
evaluate to specific values of compatible types */
587. BitXorExpression ::= BitAndExpression { "xor4b" BitAndExpression }
```

```
/* STATIC SEMANTICS - If more than one BitAndExpression exists, then the BitAndExpressions shall
evaluate to specific values of compatible types */
588. BitAndExpression ::= BitNotExpression { "and4b" BitNotExpression }
/* STATIC SEMANTICS - If more than one BitNotExpression exists, then the BitNotExpressions shall
evaluate to specific values of compatible types */
589. BitNotExpression ::= [ "not4b" ] AddExpression
/* STATIC SEMANTICS - If the not4b operator exists, the operand shall be of type bitstring,
octetstring or hexstring or derivatives of these types. */
590. AddExpression ::= MulExpression { AddOp MulExpression }
/* STATIC SEMANTICS - Each MulExpression shall resolve to a specific Value. If more than one
MulExpression exists and the AddOp resolves to StringOp then the MulExpressions shall resolve to
same type which shall be of bitstring, hexstring, octetstring, charstring or universal charstring or
derivatives of these types. If more than one MulExpression exists and the AddOp does not resolve to
StringOp then the MulExpression shall both resolve to type integer or float or derivatives of these
types.*/
591. MulExpression ::= UnaryExpression { MultiplyOp UnaryExpression }
/* STATIC SEMANTICS - Each UnaryExpression shall resolve to a specific Value. If more than one
UnaryExpression exists then the UnaryExpressions shall resolve to type integer or float or
derivatives of these types. */
592. UnaryExpression ::= [ UnaryOp ] Primary
/* STATIC SEMANTICS - Primary shall resolve to a specific Value of type integer or float or
derivatives of these types.*/
593. Primary ::= OpCall | Value | "(" SingleExpression ")"
594. ExtendedFieldReference ::= {(Dot ( StructFieldIdentifier | TypeDefIdentifier))
                               | ArrayOrBitRef }+
/* STATIC SEMANTIC - The TypeDefIdentifier shall be used only if the type of the VarInstance or
ReferencedValue in which the ExtendedFieldReference is used is anytype.*/
595. OpCall ::= ConfigurationOps |
                VerdictOps |
                TimerOps |
                TestcaseInstance |
                FunctionInstance |
                TemplateOps |
                ActivateOp
596. AddOp ::= "+" | "-" | StringOp
/* STATIC SEMANTICS - Operands of the "+" or "-" operators shall be of type integer or float or
derivations of integer or float (i.e. subrange) */
597. MultiplyOp ::= "*" | "/" | "mod" | "rem"
/* STATIC SEMANTICS - Operands of the "*", "/", rem or mod operators shall be of type integer or
float or derivations of integer or float (i.e. subrange). */
598. UnaryOp ::= "+" | "-"
/* STATIC SEMANTICS - Operands of the "+" or "-" operators shall be of type integer or float or
derivations of integer or float (i.e. subrange). */
599. RelOp ::= "<" | ">" | ">=" | "<="
/* STATIC SEMANTICS - the precedence of the operators is defined in Table 7 */
600. EqualOp ::= "==" | "!="
601. StringOp ::= "&"
/* STATIC SEMANTICS - Operands of the string operator shall be bitstring, hexstring, octetstring or
character string */
602. ShiftOp ::= "<<" | ">>" | "<@" | "@>"
603. LogStatement ::= LogKeyword "(" LogItem { "," LogItem } ")"
604. LogKeyword ::= "log"
605. LogItem ::= FreeText | TemplateInstance
606. LoopConstruct ::= ForStatement |
                       WhileStatement |
                       DoWhileStatement
607. ForStatement ::= ForKeyword "(" Initial SemiColon Final SemiColon Step ")"
                      StatementBlock
608. ForKeyword ::= "for"
609. Initial ::= VarInstance | Assignment
610. Final ::= BooleanExpression
611. Step ::= Assignment
612. WhileStatement ::= WhileKeyword "(" BooleanExpression ")"
                        StatementBlock
613. WhileKeyword ::= "while"
614. DoWhileStatement ::= DoKeyword StatementBlock
                          WhileKeyword "(" BooleanExpression ")"
615. DoKeyword ::= "do"
616. ConditionalConstruct ::= IfKeyword "(" BooleanExpression ")"
                              StatementBlock
                              {ElseIfClause}[ElseClause]
617. IfKeyword ::= "if"
618. ElseIfClause ::= ElseKeyword  IfKeyword "(" BooleanExpression ")"  StatementBlock
619. ElseKeyword ::= "else"
620. ElseClause ::= ElseKeyword  StatementBlock
621. SelectCaseConstruct ::= SelectKeyword "(" SingleExpression ")" SelectCaseBody
622. SelectKeyword ::= "select"
623. SelectCaseBody ::= "{" { SelectCase }+ "}"
```

```
624. SelectCase ::= CaseKeyword ( '(' TemplateInstance {"," TemplateInstance } ')' | ElseKeyword )
                    StatementBlock
625. CaseKeyword ::= "case"
```

A.1.6.9 Miscellaneous productions

```
626. Dot ::= "."
627. Dash ::= "-"
628. Minus ::= Dash
629. SemiColon ::= ";"
630. Colon ::= ":"
631. Underscore ::= "_"
632. AssignmentChar ::= ":="
```

Annex B

Matching incoming values

B.1 Template matching mechanisms

B.1.0 General

This annex specifies the matching mechanisms that may be used in TTCN-3 templates (and only in templates).

B.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.

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
}
```

B.1.1.1 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.

EXAMPLE:

```
template Mymessage MyTemplate:=
{  :
   :
   field3 := omit, // omit this field
   :
}
```

B.1.2 Matching mechanisms instead of values

B.1.2.0 General

The following matching mechanisms may be used in place of explicit values.

B.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.

EXAMPLE:

```
template Mymessage MyTemplate:=
{
  field1 := (2,4,6),                // list of integer values
  field2 := ("String1", "String2"),// list of charstring values
  :
  :
}
```

B.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
  :
}
```

B.1.2.3 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.

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
}
```

B.1.2.4 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.

EXAMPLE:

```
template Mymessage MyTemplate:=
{ :
  field3 := *,   // will match true or false or omitted field
  :
}
```

B.1.2.5 Value range

Ranges indicate a bounded range of acceptable values. When used for values of **integer** or **float** types (and integer or float sub-types). A boundary value shall be either:

a) infinity or –infinity;

b) an expression that evaluates to a specific integer or float 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.

When used in templates or template fields of **charstring** or **universal charstring** types, the boundaries shall evaluate to valid character positions according to the coded character set table(s) of the type (e.g., the given

position shall not be empty). Empty positions between the lower and the upper boundaries are not considered to be valid values of the specified range.

EXAMPLE:

```
template Mymessage MyTemplate:=
{
  field1 := (1 .. 6),  // range of integer type
  :
  :
  :
}
// other entries for field1 might be (-infinity to 8) or (12 to infinity)
```

B.1.2.6 SuperSet

SuperSet is an operation for matching that shall be used only on values of **`set of`** types. SuperSet is denoted by the keyword **`superset`**. A field that uses SuperSet matches the corresponding incoming field if, and only if, the incoming field contains at least all of the elements defined within the SuperSet, and may contain more. The argument of SuperSet shall be of the type declared for the field in which the SuperSet mechanism is used.

EXAMPLE:

```
type set of integer MySetOfType;

template MySetOfType MyTemplate1 := superset ( 1, 2, 3 );
// any sequence of integers matches which contains at least one occurrence of the numbers
// 1, 2 and 3 in any order and positions
```

B.1.2.7 SubSet

SubSet is an operation for matching that can be used only on values of **`set of`** types. SubSet is denoted by the keyword **`subset`**.

A field that uses SubSet matches the corresponding incoming field if, and only if, the incoming field contains only elements defined within the SubSet, and may contain less. The argument of SubSet shall be of the type declared for the field in which the SubSet mechanism is used.

EXAMPLE:

```
template MySetOfType MyTemplate1:= subset ( 1, 2, 3 );
// any sequence of integers matches which contains zero or one occurrence of the numbers
// 1, 2 and 3 in any order and positions
```

B.1.3 Matching mechanisms inside values

B.1.3.0 General

The following matching mechanisms may be used inside explicit values of strings, records, records of, sets, sets of and arrays.

B.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.

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.

B.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 B.1.5). For example: "abcdxyz", "abccxyz" "abcxxyz", etc., will all match **`pattern`** "abc?xyz". However, "abcxyz", "abcdefxyz", etc., will not.

B.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. The "*" symbol matches the longest sequence of elements possible, according to the pattern as specified by the symbols surrounding the "*".

EXAMPLE:

```
template Mymessage MyTemplate:=
{ :
   field2 := "abcxyz",
   field3 := '10*11'B,  // where "*" may be any sequence of bits (possibly empty)
   field4 := {*, 2, 3}  // where "*" may be any number of integer values 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.

B.1.3.2.1 Using multiple character wildcards

If it is required to express the "*" wildcard in character strings it shall be done using character patterns (see B.1.5). For example: "abcxyz", "abcdefxyz", "abcabcxyz", etc. will all match **pattern** "abc*xyz".

B.1.3.3 Permutation

Permutation is an operation for matching that shall be used only on values of **record of** types. Permutation is denoted by the keyword **permutation**. Expressions and *AnyElement* and *AnyElementsOrNone* are allowed as permutation elements. Each element listed in the permutation shall be of the type replicated by the **record of** type.

Permutation in place of a single element means that any series of elements is acceptable provided it contains the same elements as the value list in the permutation, though possibly in a different order. If both permutation and *AnyElementsOrNone* are used inside a value, they shall be evaluated jointly.

AnyElementsOrNone used inside permutation replaces none or any number of elements within the segment of the record of value matched by permutation. *AnyElementsOrNone* used inside a permutation shall be evaluated last (when all other elements of the permutation list have matched an element in the evaluated list already).

NOTE 1 – *AnyElementsOrNone* used inside permutation has a different effect as *AnyElementsOrNone* used in conjunction with permutation as in the latter *AnyElementsOrNone* replaces consecutive elements only. For example, {**permutation**(1,2,*)} is equivalent to ({*,1,*,2,*},{*,2,*,1,*}), while {**permutation**(1,2),*} is equivalent to ({1,2},{2,1},*).

NOTE 2 – When *AnyElementsOrNone* is used in conjunction with permutation a length attribute may be applied to *AnyElementsOrNone* to restrict the number of elements matched by *AnyElementsOrNone* (see also B.1.4.1). On the contrary, no length attribute shall be added to *AnyElementsOrNone* used inside a permutation (but can be applied to the whole permutation instead).

EXAMPLE:

```
type record of integer MySequenceOfType;

template MySequenceOfType MyTemplate1 := { permutation ( 1, 2, 3 ), 5 };
// matches any of the following sequences of 4 integers: 1,2,3,5; 1,3,2,5; 2,1,3,5;
// 2,3,1,5; 3,1,2,5; or 3,2,1,5

template MySequenceOfType MyTemplate2 := { permutation ( 1, 2, ? ), 5 };
// matches any sequence of 4 integers that ends with 5 and contains 1 and 2 at least once in
// other positions

template MySequenceOfType MyTemplate3 := { permutation ( 1, 2, 3 ), * };
// matches any sequence of integers starting with 1,2,3; 1,3,2; 2,1,3; 2,3,1; 3,1,2 or 3,2,1

template MySequenceOfType MyTemplate4 := { *, permutation ( 1, 2, 3 )};
// matches any sequence of integers ending with 1,2,3; 1,3,2; 2,1,3; 2,3,1; 3,1,2 or 3,2,1

template MySequenceOfType MyTemplate5 := { *, permutation ( 1, 2, 3 ),* };
// matches any sequence of integers containing any of the following substrings at any position:
// 1,2,3; 1,3,2; 2,1,3; 2,3,1; 3,1,2 or 3,2,1

template MySequenceOfType MyTemplate6 := { permutation ( 1, 2, * ), 5 };
// matches any sequence of integers that ends with 5 and containing 1 and 2 at least once in
// other positions
```

```
template MySequenceOfType MyTemplate7 := { permutation ( 1, 2, 3 ), * length (0..5)};
// matches any sequence of three to eight integers starting with 1,2,3; 1,3,2; 2,1,3; 2,3,1;
// 3,1,2 or 3,2,1

template MySequenceOfType MyTemplate9 := { permutation ( 1, 2, *) length (3..5), 5 };
// matches any sequence of four to six integers that ends with 5 and contains 1 and 2 at least
// once in other position
```

B.1.4 Matching attributes of values

B.1.4.0 General

The following attributes may be associated with matching mechanisms.

B.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, record of** or array structure. It shall be used only as an attribute of the following mechanisms: *AnyValue*, *AnyValueOrNone*, *AnyElement* and *AnyElementsOrNone* (but not inside permutation), permutation, superset and subset. It can also be used in conjunction with the complement matching mechanism and with the **ifpresent** attribute. The syntax for **length** can be found in 6.2.3 and 6.3.3.

NOTE – When both the complement and the length restriction matching mechanisms are used for a template or template field, restrictions implied by them shall apply to the template or template field independently.

The units of length are to be interpreted according to Table 4 in the case of string values. For **set of**, **record of** types and arrays 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.

It is allowed to use a length restriction in conjunction with the special value **omit**; however, in this case, the length attribute has no effect (i.e., with **omit** it is redundant). With *AnyValueOrNone* and **ifpresent** it places a restriction on the incoming value, if any.

EXAMPLE:

```
template Mymessage MyTemplate:=
{
  field1 := complement ({4,5},{1,4,8,9}) length (1 .. 6), // any value containing 1, 2, 3, 4,
                 // 5 or 6 elements is accepted provided it is not {4,5} or {1,4,8,9}
  field2 := "ab*ab" length(13) // max length of the AnyElementsOrNone string is 9 characters
  :
}
```

B.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.

EXAMPLE:

```
template Mymessage:MyTemplate:=
{ :
  field2 := "abcd" ifpresent, // matches "abcd" if not omitted
  :
  :
}
```

NOTE – *AnyValueOrNone* has exactly the same meaning as **? ifpresent**.

B.1.5 Matching character pattern

B.1.5.0 General

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 (e.g., ? and * within a character pattern means matching any character and any number of any character respectively).

EXAMPLE 1:

```
template charstring MyTemplate:= pattern "ab??xyz*0";
```

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 (including any number of "0"s) before the closing character "0".

If it is required to interpret any metacharacter literally, it should be preceded with the metacharacter '\'.

EXAMPLE 2:

```
template charstring MyTemplate:= pattern "ab?\?xyz*";
```

This template would match any character string which consists of the characters 'ab', followed by any character, followed by the characters '?xyz', followed by any number of any characters.

The list of metacharacters for TTCN-3 patterns is shown in Table B.1. Metacharacters shall not contain whitespaces except a whitespace preceded by a newline character before or inside a set expression.

Table B.1/Z.140 – List of TTCN-3 pattern metacharacters

Metacharacter	Description
?	Match any character (see Notes 1 and 2)
*	Match any character zero or more times; shall match the longest possible number of characters (see example 1 above) (see Notes 1 and 2)
\	Cause the following metacharacter to be interpreted as a literal (see Note 3). When preceding a character without defined metacharacter meaning '\' and the character together match the character following the '\' (see Note 4)
[ ]	Match any character within the specified set (see B.1.5.1 for more details)
-	Has a metacharacter meaning inside a pair of square brackets ("[" and "]") only, except the first and last positions within the bracket. Allows to specify a range of characters (see B.1.5.1 for more details)
^	Has a metacharacter meaning as the first character following the opening square bracket inside a pair of square brackets ("[" and "]") only and cause to match any character complementing the set of characters following this metacharacter (see B.1.5.1 for more details)
\q{group,plane,row,cell}	Match the universal character specified by the quadruple
{reference}	Insert the referenced user defined string and interpret it as a regular expression (see B.1.5.2 for more details)
\N{reference}	Match any character within the set of characters, where the set is defined by the referenced definition (see B.1.5.4 for more details)
\d	Match any numerical digit (equivalent to [0-9])
\w	Match any alphanumeric character (equivalent to [0-9a-zA-Z])
\t	Match the C0 control character HT(9) (see ISO/IEC 6429 [11])
\n	Match any of the following C0 control characters: LF(10), VT(11), FF(12), CR(13) (see ISO/IEC 6429 [11]) (jointly called newline characters)
\r	Match the C0 control character CR (see ISO/IEC 6429 [11])
\s	Match any one of the following C0 control characters: HT(9), LF(10), VT(11), FF(12), CR(13), SP(32) (see ISO/IEC 6429 [11], ITU-T Rec. T.50 [9]) (jointly called white-space characters)
\b	Match a word boundary (any graphical character except SP or DEL is preceded or followed by any of the whitespace or newline characters)
\"	Match the double quote character
""	Match the double quote character

Table B.1/Z.140 – List of TTCN-3 pattern metacharacters

<table>
<tr><th>Metacharacter</th><th>Description</th></tr>
<tr><td>|</td><td>Used to denote two alternative expressions</td></tr>
<tr><td>( )</td><td>Used to group an expression</td></tr>
<tr><td>#(n, m)</td><td>Match the preceding expression at least n times but no more than m times (postfix) (see B.1.5.3 for more details)</td></tr>
<tr><td>#n</td><td>Match the previous expression exactly n times (where n is a single digit) (postfix); the same as #(n)</td></tr>
<tr><td>+</td><td>Match the preceding expression one or several times (postfix); the same as #(1,)</td></tr>
<tr><td colspan="2">NOTE 1 – Metacharacters ? and * are able to match any characters of the character set of the root type of the template or template field in which they are used (i.e., not considering type constraints applied). However, it shall not be forgotten that receiving operations require type checking of the received message before attempting to match it. Therefore received values not complying with the subtype specification of the template or template field are never provided for matching.
NOTE 2 – In some other languages/notations ? and * have different meanings as metacharacters. However, in TTCN these characters are traditionally used for matching in the sense as specified in this table.
NOTE 3 – Consequently the backslash character can be matched by a pair of backslash characters without a space between them (\\), e.g., the pattern '\\d' will match the string '\d'; opening or closing square brackets can be matched by '\[' and '\]' respectively, etc.
NOTE 4 – Such use of the metacharacter '\' is deprecated as further metacharacters can be defined later.</td></tr>
</table>

B.1.5.1 Set expression

A list of characters enclosed by a pair of '[' and ']' matches any single character in that list. The set expression is delimited by the '[' ']' symbols. In addition to character literals, it is possible to specify character ranges using the hyphen '-' as separator. The range consists of the character immediately before the separator, the character immediately after it and all characters with a character code between the codes of the two bordering characters. A hyphen character '-' inside the list but without preceding or following character loses its special meaning.

The set expression can also be negated by placing the caret '^' character as the first character after the opening square bracket. Negation takes precedence over character ranges. Therefore a hyphen '-' immediately following a negating caret '^' shall be processed as a literal character.

An empty list and an empty negated list are not allowed. Therefore a closing square bracket ']' immediately following an opening square bracket '[' or a caret following the opening square bracket '[' and immediately followed by a closing square bracket ']' shall be processed as literal characters.

All metacharacters, except those listed below, lose their special meaning inside the list:

- ']' not at the first position and not immediately following a '^' at the first position;
- '-' not at the first or last positions in the list;
- '^' at the first position in the list except when immediately followed by a closing square bracket;
- '\', \d, \t, \w, \r, \n, \s and \b;
- \q{group,plane,row,cell};
- **\N**{reference}.

NOTE 1 – Embedded lists are not allowed (for example in pattern '[ab[r-z]]' the second '[' denotes a literal '[', the first ']' closes the list and the second ']' causes an error as there is no related opening bracket in the pattern).

NOTE 2 – To include a literal caret character "^", place it anywhere except in the first position or precede it with a backslash. To include a literal hyphen "-", place it first or last in the list, or precede it with a backslash. To include a literal closing square bracket "]", place it first or precede it with a backslash. If the first character in the list is the caret "^", then the characters "-" and "]" also match themselves when they immediately follow that caret.

EXAMPLE:

```
template charstring RegExp1:= pattern '[a-z]';  // this will match any character from a to z

template charstring RegExp2:= pattern '[^a-z]';  // this will match any character except a to z

template charstring RegExp3:= pattern '[AC-E][0-9][0-9][0-9]YKE';

// RegExp3 will match a string which starts with the letter A or a letter between
// C and E (but not e.g. B) then has three  digits and the letters YKE
```

B.1.5.2 Reference expression

In addition to direct string values, it is also possible within the pattern to use references to existing templates, constants, variables or module parameters. The reference is enclosed within the "{" "}" characters and reference shall resolve to one of the character string types. Contents of the referenced templates, constants or variables shall be handled as a regular expression. Each expression shall be dereferenced only once.

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 character. In effect, any character string following the **pattern** keyword either explicitly or by reference will be interpreted following the rules defined in this clause.

```
template universal charstring MyTemplate1:= pattern '{MyString}de\q{1, 1, 13, 7}';
```

This template would match any character string which consists of the characters 'ab', followed by any character, followed by the characters 'de', followed by the character in ISO/IEC 10646 with group=1, plane=1, row=13 and cell=7.

If a reference expression refers to a template, constant or variable which contains one or more reference expressions, then the references in the referred template, constant or variable shall recursively be dereferenced before inserting their contents into the referring pattern.

EXAMPLE:

```
const charstring MyConst2 := pattern "ab";
template charstring RegExp1 := pattern "{MyConst2}";
   // matches the string "ab"
template charstring RegExp2 := pattern "{RegExp1}{RegExp1}";
   // matches the string "abab"
template charstring RegExp3 := pattern "c{RegExp2}d";
   // matches the string "cababd"

template charstring RegExp4 := pattern "{Reg";
template charstring RegExp5 := pattern "Exp1}";
template charstring RegExp6 := pattern "{RegExp4}{RegExp5}";
   // matches the string "{RegExp1}" only (i.e. shall not be handled as a reference expression
   // to the template RegExp1)
```

B.1.5.3 Match expression n times

To specify that the preceding expression should be matched a number of times, one of the following syntaxes shall be used: '#(n, m)', '#(n,)', '#(, m)', '#(n)', '#n' or '+'. The form '#(n, m)' specifies that the preceding expression must be matched at least n times but not more than m times. The metacharacter postfix '#(n,)' specifies that the preceding expression must be matched at least n times while '#(, m)' indicates that the preceding expression shall be matched not more than m times. Metacharacters (postfixes) '#(n)' and '#n' specify that the preceding expression must be matched exactly n times (they are equivalent to '#(n, n)'). In the form '#n', n shall be a single digit. The metacharacter postfix '+' denotes that the preceding expression must be matched at least 1 time (equivalent to '#(1,)').

EXAMPLE:

```
template charstring RegExp4:= pattern '[a-z]#(9, 11)'; // match at least 9 but no more than 11
                                                        // characters from a to z
template charstring RegExp5a:= pattern '[a-z]#(9)';    // match exactly 9
                                                        // characters from a to z
template charstring RegExp5b:= pattern '[a-z]#9';      // match exactly 9
                                                        // characters from a to z
template charstring RegExp6:= pattern '[a-z]#(9, )';   // match at least 9
                                                        // characters from a to z
template charstring RegExp7:= pattern '[a-z]#(, 11)';  // match no more than 11
                                                        // characters from a to z
template charstring RegExp8:= pattern '[a-z]+';        // match at least 1
                                                        // characters from a to z,
```

B.1.5.4 Match a referenced character set

A notation of the form **"\N{**reference**}"**, where reference is denoting a one-character-length template, constant, variable or module parameter, matches the character in the referenced value or template.

Referencing a template, constant, variable or module parameter that is not of length 1 shall cause an error.

A notation of the form **"\N{**typereference**}"**, where "typereference" is a reference to a **charstring** or **universal charstring** type, matches any character of the character set denoted by the referenced type.

NOTE 1 – Cases when the referenced set of characters is not a true subset of values allowed by the type definition of the template or template field for which the character pattern is used shall not be treated as an error (but, e.g., matching can never occur if the two sets do not overlap).

NOTE 2 – \N{**charstring**} is equivalent to ? when the latter is applied to a template or template field of **charstring** type and \N{**universal charstring**} is equivalent to ? when the latter is applied to a template or template field of **universal charstring** type (but causes an error if applied to a template or template field of **charstring** type).

EXAMPLE:

```
type charstring MyCharRange ('a'..'z');
type charstring MyCharList ('a', 'z');
const MyCharRange myCharR := 'r';

template charstring myTempPatt1 := pattern '\N { myCharR }';
// myTempPatt1 shall match the string 'r' only

template charstring myTempPatt2 := pattern '\N { MyCharRange }';
// myTempPatt2 shall match any string containing a single character from a to z

template MyCharRange myTempPatt3 := pattern '\N { MyCharList }';
// myTempPatt3 and shall match strings 'a' and 'r' only

template MyCharList myTempPatt4 := pattern '\N { MyCharRange }';
// myTempPatt4 shall match strings 'a' and 'r' only
```

B.1.5.5 Type compatibility rules for patterns

For the purpose of referenced patterns (see B.1.5.2) and referenced character sets (see B.1.5.4), specific type compatibility rules apply: a referenced type, template, constant, variable or module parameter of the type **charstring** can always be used in the pattern specification of a template or template field of **universal charstring** type; a referenced type, template or value of the type **universal charstring** can be used in the pattern specification of a template or template field of **charstring** type if all characters used in the referenced template or value, and the character set allowed by the referenced type, has their corresponding characters in the **charstring** type (see definition of corresponding characters in 6.7.1).

Annex C

Pre-defined TTCN-3 functions

This annex defines the TTCN-3 predefined functions.

C.0 General exception handling procedures

Error situations (e.g., input parameter is out of the allowed range, input parameter is of a wrong type, input value contains improper character, etc.), for which no explicit exception-handling rule is defined in the relevant clauses of this annex, shall cause a TTCN-3 compile-time or run-time error. Which error situation causes compile-time and which one run-time error is a tool implementation option.

C.1 Integer to character

```
int2char(integer value) return charstring
```

This function converts an **integer** value in the range of 0 to 127 (8-bit encoding) into a single-character-length **charstring** value. The integer value describes the 8-bit encoding of the character.

C.2 Character to integer

```
char2int(charstring value) return integer
```

This function converts a single-character-length **charstring** value into an integer value in the range of 0 to 127. The integer value describes the 8-bit encoding of the character.

C.3 Integer to universal character

```
int2unichar(integer value) return universal charstring
```

This function converts an **integer** value in the range of 0 to 2 147 483 647 (32-bit encoding) into a single-character-length **universal charstring** value. The integer value describes the 32-bit encoding of the character.

C.4 Universal character to integer

```
unichar2int(universal charstring value) return integer
```

This function converts a single-character-length **universal charstring** value into an integer value in the range of 0 to 2 147 483 647. The integer value describes the 32-bit encoding of the character.

C.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 the least significant, the leftmost bit is the most significant. The bits 0 and 1 represent the decimal values 0 and 1 respectively.

C.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 to F represent the decimal values 0 to 15 respectively.

C.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, an **octetstring** shall be interpreted as a positive base 16 **integer** value. The rightmost hexadecimal digit is the 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 to F represent the decimal values 0 to 15 respectively.

C.8 Charstring to integer

```
str2int(charstring value) return integer
```

This function converts a **charstring** representing an **integer** value to the equivalent **integer**.

EXAMPLE:

```
str2int("66")     // will return the integer value 66

str2int("-66")    // will return the integer value -66

str2int("abc")    // will generate compiler or testcase error

str2int("0")      // will return the integer value 0
```

C.9 Integer to bitstring

```
int2bit(in integer value, in integer 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 the 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.

C.10 Integer to hexstring

```
int2hex(in integer value, in integer 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 the least significant, the leftmost hexadecimal digit is the most significant. The hexadecimal digits 0 to F represent the decimal values 0 to 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.

C.11 Integer to octetstring

```
int2oct(in integer value, in integer 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 the 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 to F represent the decimal values 0 to 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.

C.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).

EXAMPLE:

```
int2str(66)    // will return the charstring value "66"

int2str(-66)   // will return the charstring value "-66"

int2str(0)     // will return the charstring value "0"
```

C.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.

The length of an **universal charstring** shall be calculated by counting each combining character and hangul syllable character (including fillers) on its own (see ISO/IEC 10646 [10], and clauses 23 and 24).

EXAMPLE:

```
lengthof('010'B) // returns 3

lengthof('F3'H) // returns 2

lengthof('F2'O) // returns 1

lengthof (universal charstring : "Length_of_Example") // returns 17
```

C.14 Number of elements in a structured value

```
sizeof(any_type value) return integer
```

This function returns the actual number of elements of a module parameter, constant, variable or **template** of a **record**, **record of**, **set**, **set of** type or array (see Note). In the case of **record of** and **set of** values, templates or arrays, the actual value to be returned is the sequential number of the last defined element (index of that element plus 1).

NOTE – Only elements of the TTCN-3 object, which is the parameter of the function are calculated; i.e., no elements of nested types/values are taken into account at determining the return value.

EXAMPLE:

```
// Given
type record MyPDU
  {  boolean field1  optional,
     integer field2
  };

template MyPDU  MyTemplate
  { field1  omit,
    field2 5
  };

var integer numElements;

// then

numElements := sizeof(MyTemplate);  // returns 1

// Given
type record length(0..10) of integer MyList;
var MyList MyRecordVar;
MyRecordVar := { 0, 1, omit, 2, omit };


// then
numElements := sizeof(MyRecordVar);
// returns 4 without respect to the fact, that the element MyRecordVar[2] is undefined
```

C.15 The IsPresent function

```
ispresent(any_type value) return boolean
```

This function is allowed for **record** and **set** types only and 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 of a **record** or **set** type.

```
// 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
```

C.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
```

C.17 The Regexp function

```
regexp (any_character_string_type instr, charstring expression, integer groupno) return
character_string_type
```

This function returns the substring of the input character string `instr`, which is the content of n-th group matching to the `expression`. In input string `instr` may be of any character string type. The type of the character string returned is the root type of `instr`. The expression is a character pattern as described in B.1.5. The number of the group to be returned is specified by `groupno`, which shall be a positive integer. Group numbers are assigned by the order of occurrences of the opening bracket of a group and counted starting from 0 by step 1. If no substring fulfilling all conditions (i.e., pattern and group number) is found within the input string, an empty string is returned.

EXAMPLE:

```
// Given
var charstring mypattern2 := "
var charstring  myinput := '      date: 2001-10-20 ;  msgno: 17; exp  '
var charstring mypattern := '[ /t]#(,)date:[ \d\-]#(,);[ /t]#(,)msgno: (\d#(1,3)); [exp]#(0,1)'

// Then the expression
var charstring mystring := regexp(myinput, mypattern,1)
//will return the value '17'.
```

C.18 Bitstring to charstring

```
bit2str (bitstring value) return charstring
```

This function converts a single **bitstring** value to a single **charstring**. The resulting **charstring** has the same length as the **bitstring** and contains only the characters '0' and '1'.

For the purpose of this conversion, a **bitstring** should be converted into a **charstring**. Each bit of the **bitstring** is converted into a character '0' or '1' depending on the value 0 or 1 of the bit. The consecutive order of characters in the resulting **charstring** is the same as the order of bits in the **bitstring**.

EXAMPLE:

```
bit2str ('1110101'B) will return "1110101"
```

C.19 Hexstring to charstring

```
hex2str (hexstring value) return charstring
```

This function converts a single hexstring value to a single charstring. The resulting charstring has the same length as the hexstring and contains only the characters '0' to '9' and 'A' to 'F'.

For the purpose of this conversion, a **hexstring** should be converted into a **charstring**. Each hex digit of the **hexstring** is converted into a character '0' to '9' and 'A' to 'F' depending on the value 0 to 9 or A to F of the hex digit. The consecutive order of characters in the resulting **charstring** is the same as the order of digits in the **hexstring**.

EXAMPLE:

```
hex2str ('AB801'H) will return "AB801"
```

C.20 Octetstring to character string

```
oct2str (octetstring invalue) return charstring
```

This function converts an **octetstring** `invalue` to an **charstring** representing the string equivalent of the input value. The resulting **charstring** shall have the same length as the incoming **octetstring**.

For the purpose of this conversion each hex digit of `invalue` is converted into a character '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E' or 'F', echoing the value of the hex digit. The consecutive order of characters in the resulting **charstring** is the same as the order of hex digits in the **octetstring**.

EXAMPLE:

```
oct2str ('4469707379'O) = "4469707379"
```

C.21 Character string to octetstring

```
str2oct (charstring invalue) return octetstring
```

This function converts a string of the type **charstring** to an **octetstring**. The string `invalue` shall contain even number characters and each shall be one of the '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e' 'f', 'A', 'B', 'C', 'D', 'E' or 'F' graphical characters only. The resulting **octetstring** will have the same length as the incoming **charstring**.

EXAMPLE:

```
str2oct ("54696E6B792D57696E6B79") = '54696E6B792D57696E6B79'O
```

C.22 Bitstring to hexstring

```
bit2hex (bitstring value) return hexstring
```

This function converts a single **bitstring** value to a single **hexstring**. The resulting **hexstring** represents the same value as the **bitstring**.

For the purpose of this conversion, a bitstring shall be converted into a hexstring, where the bitstring is divided into groups of four bits beginning with the rightmost bit. Each group of four bits is converted into a hex digit as follows:

'0000'B → '0'H, '0001'B → '1'H, '0010'B → '2'H, '0011'B → '3'H, '0100'B → '4'H, '0101'B → '5'H,
'0110'B → '6'H, '0111'B → '7'H, '1000'B → '8'H, '1001'B → '9'H, '1010'B → 'A'H, '1011'B → 'B'H,
'1100'B → 'C'H, '1101'B → 'D'H, '1110'B → 'E'H, and '1111'B → 'F'H.

When the leftmost group of bits does contain less than 4 bits, this group is filled with '0'B from the left until it contains exactly 4 bits and is converted afterwards. The consecutive order of hex digits in the resulting hexstring is the same as the order of groups of 4 bits in the bitstring.

EXAMPLE:

```
bit2hex ('111010111'B)= '1D7'H
```

C.23 Hexstring to octetstring

```
hex2oct (hexstring value) return octetstring
```

This function converts a single **hexstring** value to a single **octetstring**. The resulting **octetstring** represents the same value as the **hexstring**.

For the purpose of this conversion, a **hexstring** shall be converted into a **octetstring**, where the **octetstring** contains the same sequence of hex digits as the **hexstring** when the length of the **hexstring** modulo 2 is 0. Otherwise, the resulting **octetstring** contains 0 as leftmost hex digit followed by the same sequence of hex digits as in the **hexstring**.

EXAMPLE:

```
hex2oct ('1D7'H)= '01D7'O
```

C.24 Bitstring to octetstring

```
bit2oct (bitstring value) return octetstring
```

This function converts a single **bitstring** value to a single **octetstring**. The resulting **octetstring** represents the same value as the **bitstring**.

For the conversion the following holds: bit2oct(value)=hex2oct(bit2hex(value)).

EXAMPLE:

```
bit2oct ('111010111'B)= '01D7'O
```

C.25 Hexstring to bitstring

```
hex2bit (hexstring value) return bitstring
```

This function converts a single **hexstring** value to a single **bitstring**. The resulting **bitstring** represents the same value as the **hexstring**.

For the purpose of this conversion, a **hexstring** shall be converted into a **bitstring**, where the hex digits of the **hexstring** are converted in groups of bits as follows:

'0'H → '0000'B, '1'H → '0001'B, '2'H → '0010'B, '3'H → '0011'B, '4'H → '0100'B, '5'H → '0101'B,
'6'H → '0110'B, '7'H → '0111'B, '8'H → '1000'B, '9'H → '1001'B, 'A'H → '1010'B, 'B'H → '1011'B,
'C'H → '1100'B, 'D'H → '1101'B, 'E'H → '1110'B, and 'F'H → '1111'B.

The consecutive order of the groups of 4 bits in the resulting **bitstring** is the same as the order of hex digits in the **hexstring**.

EXAMPLE:

```
hex2bit ('1D7'H)= '000111010111'B
```

C.26 Octetstring to hexstring

```
oct2hex (octetstring value) return hexstring
```

This function converts a single **octetstring** value to a single **hexstring**. The resulting **hexstring** represents the same value as the **octetstring**.

For the purpose of this conversion, a **octetstring** shall be converted into a **hexstring** containing the same sequence of hex digits as the **octetstring**.

EXAMPLE:

```
oct2hex ('1D74'O)= '1D74'H
```

C.27 Octetstring to bitstring

```
oct2bit (octetstring value) return bitstring
```

This function converts a single **octetstring** value to a single **bitstring**. The resulting **bitstring** represents the same value as the **octetstring**.

For the conversion, the following holds: oct2bit(value)=hex2bit(oct2hex(value)).

EXAMPLE:

```
oct2bit ('01D7'O)='0000000111010111'B
```

C.28 Integer to float

```
int2float (integer value) return float
```

This function converts an **integer** value into a **float** value.

EXAMPLE:

```
int2float(4) = 4.0
```

C.29 Float to integer

```
float2int (float value) return integer
```

This function converts a **float** value into an **integer** value by removing the fractional part of the argument and returning the resulting **integer**.

EXAMPLE:

```
float2int(3.12345E2) = float2int(312.345) = 312
```

C.30 The random number generator function

```
rnd ([float seed]) return float
```

The **rnd** function returns a (pseudo) random number less than 1 but greater than or equal to 0. The random number generator is initialized by means of an optional seed value. Afterwards, if no new seed is provided, the last generated number will be used as seed for the next random number. Without a previous initialization a value calculated from the system time will be used as seed value when **rnd** is used the first time.

NOTE – Each time the **rnd** function is initialized with the same seed value, it shall repeat the same sequence of random numbers.

To produce a random integer in a given range, the following formula can be used:

```
float2int(int2float(upperbound - lowerbound +1)*rnd()) + lowerbound
// Here, upperbound and lowerbound denote highest and lowest number in range.
```

C.31 The Substring function

```
substr (any_string_type value, in integer index, in integer returncount) return input_string_type
```

This function returns a substring from a value that is of type **bitstring**, **hexstring**, **octetstring**, or any character string. The type of the substring is the root type of the input value. The starting point of substring to return is defined by the second in parameter (index). Indexing starts from zero. The third input parameter defines the length of the substring to be returned. The units of length are as defined in Table 4.

EXAMPLE:

```
substr ('00100110'B, 3, 4)      // returns '0011'B

substr ('ABCDEF'H, 2, 3)        // returns 'CDE'H

substr ('01AB23CD'O, 1, 2)      // returns 'AB23'O

substr ("My name is JJ", 11, 2) // returns "JJ"
```

C.32 Number of elements in a structured type

```
sizeoftype(any_type value) return integer
```

This function returns the declared number of elements of a module parameter, constant, variable or **template** of a **record of** or **set of** type or array (see Note). This function shall be applied to values of types with length restriction. The actual number to be returned is the sequential number of the last element without respect to whether its value is defined or not (i.e., the upper length index of the type definition on which the parameter of the function is based, plus 1).

NOTE – Only elements of the TTCN-3 object, which is the parameter of the function are calculated; i.e., no elements of nested types/values are taken into account at determining the return value.

EXAMPLE:

```
// Given
type record of integer MyPDU1;
type set length(1..8) of integer MyPDU2;
type record length(10) of integer MyPDU3;

var MyPDU1 MyRecordOfVar1;
var MyPDU2 MyRecordOfVar2;
var MyPDU3 MyRecordOfVar3;

var integer numElements;

// then
numElements := sizeoftype(MyRecordOfVar1);  // returns error as MyPDU1 is not constrained
numElements := sizeoftype(MyRecordOfVar2);  // returns 8
numElements := sizeoftype(MyRecordOfVar3);  // returns 10
```

C.33 Character string to float

```
str2float (charstring value) return float
```

This function converts a **charstring** comprising a floating-point number into a **float** value. The format of the number in the **charstring** shall follow rules in 6.1.0 with the following exceptions:

- leading zeros are allowed,
- leading '+' sign before positive values is allowed,
- '-0.0' is allowed.

EXAMPLE:

```
str2float('12345.6') //  is the same as str2float('123.456E+02')
```

C.34 The Replace function

```
replace (in any_string_type str, in integer ind, in integer len, in any_string_type repl)
return any_string_type
```

This function replaces the substring of value **str** at index **ind** of length **len** with the string value **repl** and returns the resulting string. **str** shall not be modified. If **len** is **0**, the string **repl** is inserted. If **ind** is **0**, **repl** is inserted at the beginning of **str**. If **ind** is **lengthof(str)**, **repl** is inserted at the end of **str**. **str** and **repl** shall be of the same string type and shall have as base type **bitstring**, **hexstring**, **octetstring**, or any character string. The returned string is of the same type as **str** and **repl**. Note that indexing in strings starts from zero.

The following error cases will lead to an error at compile or runtime:

- **str** or **repl** are not of string type;
- **str** and **repl** are of different type;
- **ind** is less than **0** or greater than **lengthof(str)**;
- **len** is less than **0** or greater than **lengthof(str)**;
- **ind+len** is greater than **lengthof(str)**.

EXAMPLE:

```
replace ('00000110'B, 1, 3, '111'B)     // returns '01110110'B

replace ('ABCDEF'H, 0, 2, '123'H)       // returns '123CDEF'H

replace ('01AB23CD'O, 2, 1, 'FF96'O)    // returns '01ABFF96CD'O

replace ("My name is JJ", 11, 1, "xx")  // returns "My name is xxJ"

replace ("My name is JJ", 11, 0, "xx")  // returns "My name is xxJJ"

replace ("My name is JJ", 2, 2, "x")    // returns "Myxame is JJ",

replace ("My name is JJ", 12, 2, "xx")  // produces test case error

replace ("My name is JJ", 13, 2, "xx")  // produces test case error

replace ("My name is JJ", 13, 0, "xx")  // returns "My name is JJxx"
```

C.35 Octetstring to character string

```
oct2char (octetstring invalue) return charstring
```

This function converts an **octetstring** `invalue` to a **charstring**. The input parameter `invalue` shall not contain octet values higher than 7F. The resulting **charstring** shall have the same length as the input **octetstring**. The octets are interpreted as ITU-T Rec. T.50 [9] codes (according to the IRV) and the resulting characters are appended to the returned value.

EXAMPLE:

```
oct2char ('4469707379'O) = "Dipsy"
```

NOTE – The character string returned may contain non-graphical characters, which cannot be presented between the double quotes.

C.36 Character string to octetstring

```
char2oct (charstring invalue) return octetstring
```

This function converts a **charstring** `invalue` to an **octetstring**. Each octet of the **octetstring** will contain the ITU-T Rec. T.50 [9] codes (according to the IRV) of the appropriate characters of `invalue`.

EXAMPLE:

```
char2oct ("Tinky-Winky") = '54696E6B792D57696E6B79'O
```

Annex D (informative)

Void

NOTE – The content of this annex has been moved to ITU-T Rec. Z.146 [6].

Annex E (informative)

Library of useful types

E.1 Limitations

Names of types added to this library should be unique within the whole language and within the library (i.e., should not be one of the names defined in Annex C). Names defined in this library should not be used by TTCN-3 users as identifiers of other definitions than given in this annex.

NOTE – Therefore, type definitions given in this annex may be repeated in TTCN-3 modules, but no type distinct from the one specified in this annex can be defined with one of the identifiers used in this annex.

E.2 Useful TTCN-3 types

E.2.1 Useful simple basic types

E.2.1.0 Signed and unsigned single byte integers

These types support integer values of the range from –128 to 127 for the signed type, and from 0 to 255 for the unsigned type. The value notation for these types are the same as the value notation for the integer type. Values of these types shall be encoded and decoded as they were represented on a single byte within the system, independently from the actual representation form used.

NOTE – Encoding of values of these types may be the same or may differ from each other and from the encoding of the integer type (the root type of these useful types) depending on the actual encoding rules used. Details of encoding rules are out of the scope of this Recommendation.

Type definitions for these types are:

```
type integer   byte     (-128 .. 127)     with { variant "8 bit" };

type integer   unsignedbyte (0 .. 255)    with { variant "unsigned 8 bit" };
```

E.2.1.1 Signed and unsigned short integers

These types support integer values of the range from –32 768 to 32 767 for the signed type, and from 0 to 65 535 for the unsigned type. The value notation for these types are the same as the value notation for the integer type. Values of these types shall be encoded and decoded as they were represented on two bytes within the system, independently from the actual representation form used.

NOTE – Encoding of values of these types may be the same or may differ from each other and from the encoding of the integer type (the root type of these useful types) depending on the actual encoding rules used. Details of encoding rules are out of the scope of this Recommendation.

Type definitions for these types are:

```
type integer   short       (-32768 .. 32767)   with { variant "16 bit" };

type integer   unsignedshort  (0 .. 65535)     with { variant "unsigned 16 bit" };
```

E.2.1.2 Signed and unsigned long integers

These types support integer values of the range from –2 147 483 648 to 2 147 483 647 for the signed type, and from 0 to 4 294 967 295 for the unsigned type. The value notation for these types are the same as the value notation for the integer type. Values of these types shall be encoded and decoded as they were represented on four bytes within the system, independently from the actual representation form used.

NOTE – Encoding of values of these types may be the same or may differ from each other and from the encoding of the integer type (the root type of these useful types) depending on the actual encoding rules used. Details of encoding rules are out of the scope of this Recommendation.

Type definitions for these types are:

```
type integer   long (-2147483648 .. 2147483647)
               with { variant "32 bit" };

type integer   unsignedlong (0 .. 4294967295)
               with { variant "unsigned 32 bit" };
```

E.2.1.3 Signed and unsigned longlong integers

These types support integer values of the range from –9 223 372 036 854 775 808 to 9 223 372 036 854 775 807 for the signed type, and from 0 to 18 446 744 073 709 551 615 for the unsigned type. The value notation for these types are the same as the value notation for the integer type. Values of these types shall be encoded and decoded as they were represented on eight bytes within the system, independently from the actual representation form used.

NOTE – Encoding of values of these types may be the same or may differ from each other and from the encoding of the integer type (the root type of these useful types) depending on the actual encoding rules used. Details of encoding rules are out of the scope of this Recommendation.

Type definitions for these types are:

```
type integer   longlong(-9223372036854775808 .. 9223372036854775807)
               with { variant "64 bit" };

type integer   unsignedlonglong (0 .. 18446744073709551615)
               with { variant "unsigned 64 bit" };
```

E.2.1.4 IEEE 754 floats

These types support the ANSI/IEEE Standard 754 (see Bibliography) for binary floating-point arithmetic. The type IEEE 754 float supports floating-point numbers with base 10, exponent of size 8, mantissa of size 23 and a sign bit. The type IEEE 754 double supports floating-point numbers with base 10, exponent of size 11, mantissa of size 52 and a sign bit. The type IEEE 754 `extfloat` supports floating-point numbers with base 10, minimal exponent of size 11, minimal mantissa of size 32 and a sign bit. The type IEEE 754 `extdouble` supports floating-point numbers with base 10, minimal exponent of size 15, minimal mantissa of size 64 and a sign bit.

Values of these types shall be encoded and decoded according to the IEEE 754 definitions. The value notation for these types are the same as the value notation for the float type (base 10).

NOTE – Precise encoding of values of this type depends on the actual encoding rules used. Details of encoding rules are out of the scope of this Recommendation.

Type definitions for these types are:

```
type  float    IEEE754float     with { variant "IEEE754 float" };

type  float    IEEE754double    with { variant "IEEE754 double" };

type  float    IEEE754extfloat  with { variant "IEEE754 extended float" };

type  float    IEEE754extdouble    with { variant "IEEE754 extended double" };
```

E.2.2 Useful character string types

E.2.2.0 UTF-8 character string "utf8string"

This type supports the whole character set of the TTCN-3 type **`universal charstring`** (see paragraph d of 6.1.1). Its distinguished values are zero, one, or more characters from this set. Values of this type shall entirely (e.g., each character of the value individually) be encoded and decoded according to the UCS Transformation Format 8 (UTF-8) as defined in Annex R of ISO/IEC 10646 [10]. The value notation for this type is the same as the value notation for the **`universal charstring`** type.

The type definition for this type is:

```
type universal charstring utf8string with { variant "UTF-8" };
```

E.2.2.1 BMP character string "bmpstring"

This type supports the Basic Multilingual Plane (BMP) character set of ISO/IEC 10646 [10]. The BMP represents all characters of plane 00 of group 00 of the Universal Multiple-octet coded Character Set. Its distinguished values are zero, one, or more characters from the BMP. Values of this type shall entirely (e.g., each character of the value individually) be encoded and decoded according to the UCS-2 coded representation form (see 14.1 of ISO/IEC 10646 [10]). The value notation for this type is the same as the value notation for the **`universal charstring`** type.

NOTE – The type "bmpstring" supports a subset of the TTCN-3 type **`universal charstring`**.

The type definition for this type is:

```
type universal charstring bmpstring ( char ( 0,0,0,0 ) .. char ( 0,0,255,255) )
                          with { variant "UCS-2" };
```

E.2.2.2 UTF-16 character string "utf16string"

This type supports all characters of planes 00 to 16 of group 00 of the Universal Multiple-octet coded Character Set (see ISO/IEC 10646 [10]). Its distinguished values are zero, one, or more characters from this set. Values of this type shall entirely (e.g., each character of the value individually) be encoded and decoded according to the UCS Transformation Format 16 (UTF-16) as defined in Annex Q of ISO/IEC 10646 [10]. The value notation for this type is the same as the value notation for the **universal charstring** type.

NOTE – The type "utf16string" supports a subset of the TTCN-3 type **universal charstring**.

The type definition for this type is:

```
type universal charstring utf16string  ( char ( 0,0,0,0 ) .. char ( 0,16,255,255) )
                          with { variant "UTF-16" };
```

E.2.2.3 ISO/IEC 8859 character string "iso8859string"

This type supports all characters in all alphabets defined in the multiparty standard ISO/IEC 8859 (see bibliography). Its distinguished values are zero, one, or more characters from the ISO/IEC 8859 character set. Values of this type shall entirely (e.g., each character of the value individually) be encoded and decoded according to the coded representation as specified in ISO/IEC 8859 (an 8-bit coding). The value notation for this type is the same as the value notation for the **universal charstring** type.

NOTE 1 – The type "iso8859string" supports a subset of the TTCN-3 type **universal charstring**.

NOTE 2 – In each ISO/IEC 8859 alphabet the lower part of the character set table (positions 02/00 to 07/14) is compatible with the ITU-T Rec. T.50 [9] character set. Hence, all extra language specific characters are defined for the upper part of the character table only (positions 10/00 to 15/15). As the "iso8859string" type is defined as a subset of the TTCN-3 type universal charstring, any coded character representation of any ISO/IEC 8859 alphabets can be mapped into an equivalent character (a character with the same coded representation when encoded on 8 bits) from the Basic Latin or Latin-1 Supplement character tables of ISO/IEC 10646 [10].

The type definition for this type is:

```
type universal charstring iso8859string ( char ( 0,0,0,0 ) .. char ( 0,0,0,255) )
                          with { variant "8 bit" };
```

E.2.3 Useful structured types

E.2.3.0 Fixed-point decimal literal

This type supports the use of fixed-point decimal literal as defined in the IDL Syntax and Semantics version 2.6 (see bibliography). It is specified by an integer part, a decimal point and a fraction part. The integer and fraction parts both consist of a sequence of decimal (base 10) digits. The number of digits is stored in "digits" and the size of the fraction part is given in "scale". The digits themselves are stored in "value_". Value notation for this type is the same as the value notation for the record type. Values of this type shall be encoded and decoded as IDL fixed point decimal values.

NOTE – Precise encoding of values of this type depends on the actual encoding rules used. Details of encoding rules are out of the scope of this Recommendation.

The type definition for this type is:

```
type record IDLfixed {
              unsignedshort  digits,
              short    scale,
              charstring  value_
                 }
           with { variant "IDL:fixed FORMAL/01-12-01 v.2.6" };
```

E.2.4 Useful atomic string types

E.2.4.1 Single IRV character type

A type whose distinguished values are single characters of the version of ITU-T Rec. T.50 [9] complying to the International Reference Version (IRV) as specified in 8.2/T.50 [9] (see also Note 2 of 6.1.1).

The type definition for this type is:

```
type charstring char length (1);
```

NOTE 1 – The name of this useful type is the same as the TTCN-3 keyword used to denote **universal charstring** values in the quadraple form. In general, it is disallowed to use TTCN-3 keywords as identifiers. The "char" useful type is a solitary exception and allowed only for backward compatibility with previous versions of the TTCN-3 standard.

NOTE 2 – The special string "8 bit" defined in 28.2.3 may be used with this type to specify a given encoding for its values. Also, other properties of the base type can be changed by using attribute mechanisms.

E.2.4.2 Single universal character type

A type whose distinguished values are single characters from ISO/IEC 10646 [10].

The type definition for this type is:

```
type universal charstring uchar length (1);
```

NOTE – Special strings defined in 28.2.3, except "8 bit", may be used with this type to specify a given encoding for its values. Also, other properties of the base type can be changed by using attribute mechanisms.

E.2.4.3 Single bit type

A type whose distinguished values are single binary digits.

The type definition for this type is:

```
type bitstring bit length (1);
```

E.2.4.4 Single hex type

A type whose distinguished values are single hexadecimal digits.

The type definition for this type is:

```
type hexstring hex length (1);
```

E.2.4.5 Single octet type

A type whose distinguished values are pairs of hexadecimal digits.

The type definition for this type is:

```
type octetstring octet length (1);
```

Annex F (informative)

Operations on TTCN-3 active objects

F.1 General

This annex describes in a short form the semantics of operations on active objects in TTCN-3, these being test components, timers and ports. This dynamic behaviour is written in the form of state machines with:

- the states being named and identified as nodes;
- the initial state being identified by an incoming arrow;
- transitions between states connecting two states (not necessarily different states) and identified as arrows;
- transitions being marked with the enabling condition for that transition (i.e., operation or statement calls) and the resulting condition (for example a test case error), both are separated by '/':
  - operation and statement calls are the TTCN-3 operations and statements applicable to the object (written in bold);
  - error as a resulting condition means testcase error (written in bold);
  - null as a resulting condition means that except for a possible state change, no other results apply (written in bold);
  - match/no match refers to the matching result of a transition (written in bold);
  - concrete values are Boolean or float results (written in bold italics);
  - all other resulting conditions are textually described (written in standard font);
- notes are used to explain further details of the state machine.

For further details, please refer to the operational semantics of TTCN-3 [3]. In case of any contradiction between this annex and the operational semantics of TTCN-3 [3], the latter takes precedence.

F.2 Test components

F.2.1 Test component references

Variables of test component types, the **`self`** and **`mtc`** operations are used to reference test components. The **`start`**, **`stop`**, **`done`** and **`running`** operations are not directly applied on test components but on component references. The test system shall decide if the operation requested shall effect the component object itself or if the other action is appropriate (e.g., an error occurs when the reference of a stopped PTC is used in a component start operation). The **`create`** operation used to create PTCs returns a unique reference to the created PTC, which is typically bound to a test component variable. The behaviour related to test component variables themselves is shown in Figure F.1.

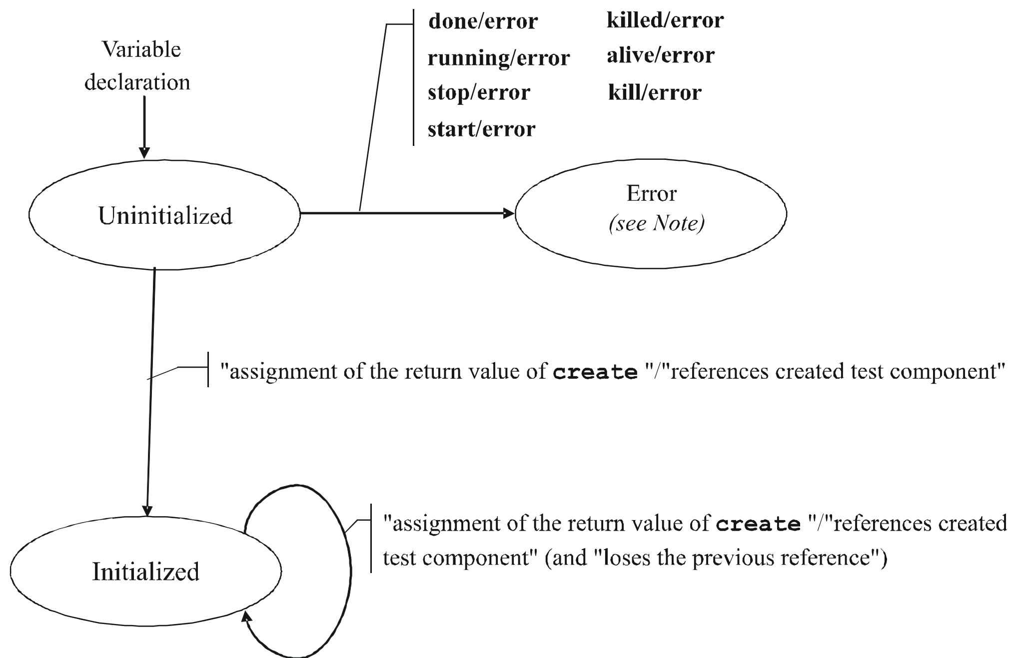
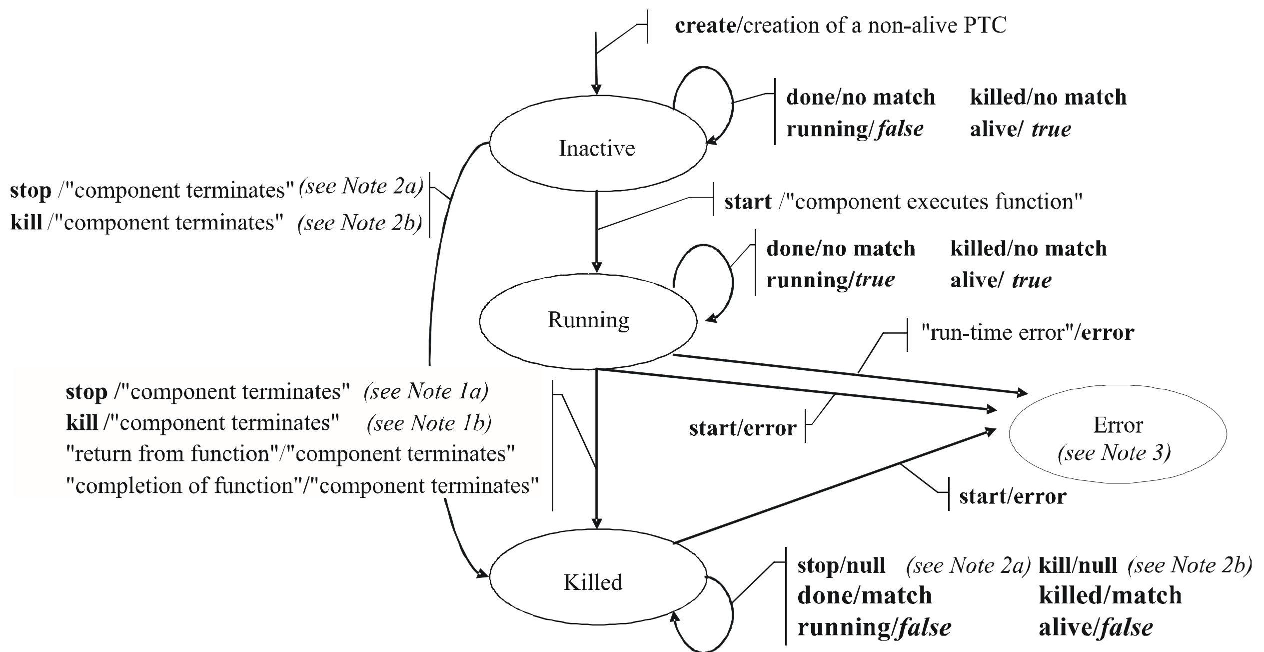


Figure F.1/Z.140 – Handling of test component references

F.2.2 Dynamic behaviour of PTCs

PTCs can be of non-alive type or alive-type. Non-alive type PTCs can be in Inactive, Running and Killed states. Their dynamic behaviour is shown in Figure F.2.



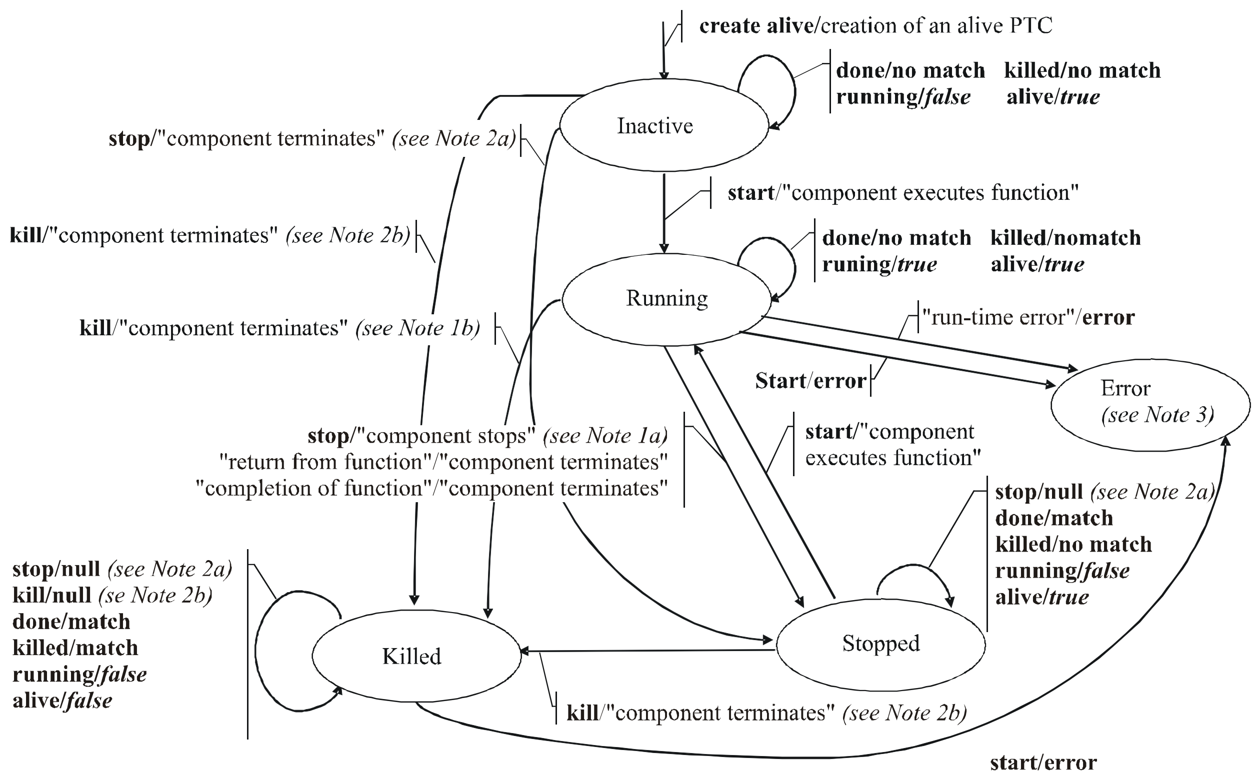
NOTE 1 – a) Stop can be either a stop, self.stop or a stop from another test component;
b) Kill can be either a kill, self.kill, a kill from another test component or a kill from the test system (in error cases).

NOTE 2 – a) Stop can be from another test component only;
b) Kill can be from another test component or from the test system (in error cases) only.

NOTE 3 – Whenever a test component enters its error state, the error verdict is assigned to its local verdict, the test case terminates and the overall test case result will be an error.

Figure F.2/Z.140 – Dynamic behaviour of non-alive type PTCs

Alive-type PTCs can be in Inactive, Running, Stopped and Killed states. Their dynamic behaviour is shown in Figure F.3.



NOTE 1 – a) Stop can be either a stop, self.stop or a stop from another test component;
b) Kill can be either a kill, self.kill, a kill from another test component or a kill from the test system (in error cases).

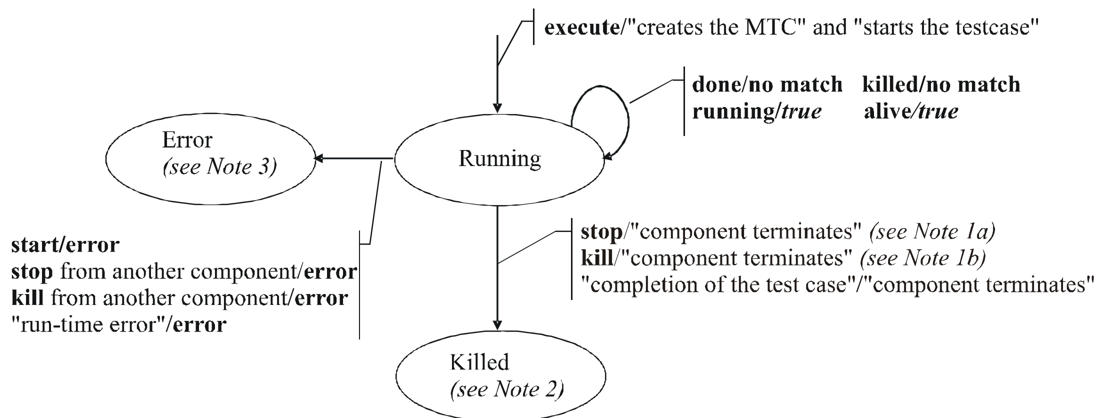
NOTE 2 – a) Stop can be from another test component only;
b) Kill can be from another test component or from the test system (in error cases) only.

NOTE 3 – Whenever a test component enters its error state, the error verdict is assigned to its local verdict, the test case terminates and the overall test case result will be an error.

Figure F.3/Z.140 – Dynamic behaviour of alive-type PTCs

F.2.3 Dynamic behaviour of the MTC

The MTC can be in Running or Killed state. The dynamic behaviour of the MTC is shown in Figure F.4.



NOTE 1 – a) Stop can be either a stop, self.stop, a stop from another test component;
b) Kill can be either a kill, self.kill, a kill from another test component or a kill from the test system (in error cases).

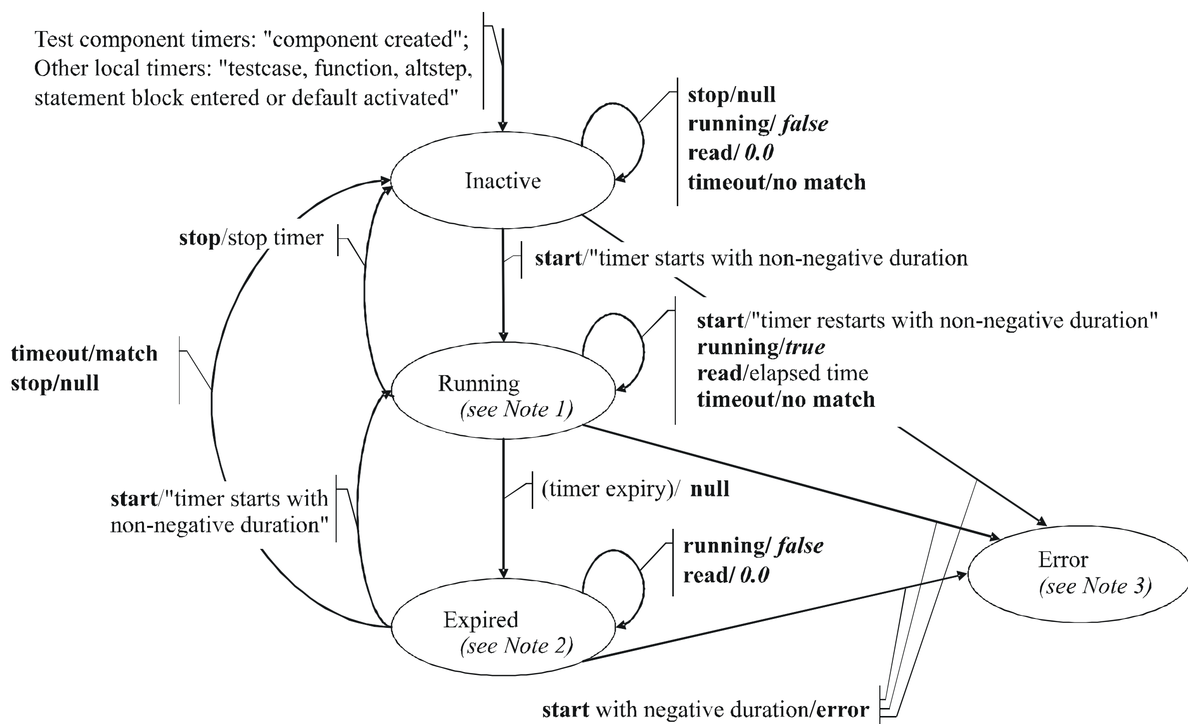
NOTE 2 – All remaining PTCs shall be killed as well and the testcase terminates.

NOTE 3 – Whenever the MTC enters its error state, the error verdict is assigned to its local verdict, the test case terminates and the overall test case result will be an error.

Figure F.4/Z.140 – Dynamic behaviour of the MTC

F.3 Timers

Timers can be in Inactive, Running or Expired state. The dynamic behaviour of a timer is shown in Figure F.5.



NOTE 1 – For any scope unit, all timers in that scope being in Running state constitute the running-timer list.

NOTE 2 – For any scope unit, all timers in that scope being in Expired state constitute the timeout-list.

NOTE 3 – Whenever a timer enters its error state, the test component it belongs to enters also its error state, assigns a local error verdict, the test case terminates and the overall test case result will be an error.

Figure F.5/Z.140 – Dynamic behaviour of timers

F.4 Ports

Ports can be in Started or Stopped state. As their behaviour is rather complex, the state machine has been split into a state machine giving the dynamic behaviour of configuration operations (i.e., connect, disconnect, map and unmap), of port controlling operations (i.e., start, stop, and clear) and of communication operations (i.e., send, receive, call, getcall, raise, catch, reply, getreply, and check). As trigger is a shorthand for an alt together with receive, it is not considered here.

F.4.1 Configuration operations

The port configuration operations (i.e., connect, disconnect, map, and unmap) are indifferent to the state of the port. They show the behaviour shown in Figure F.6.

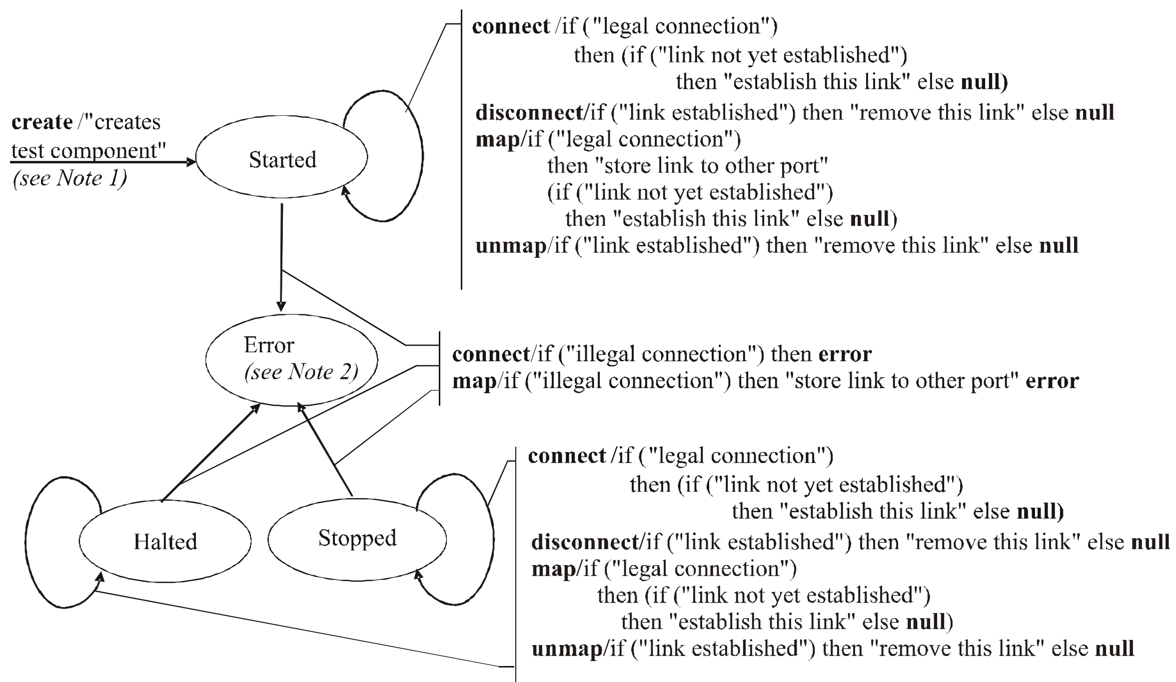
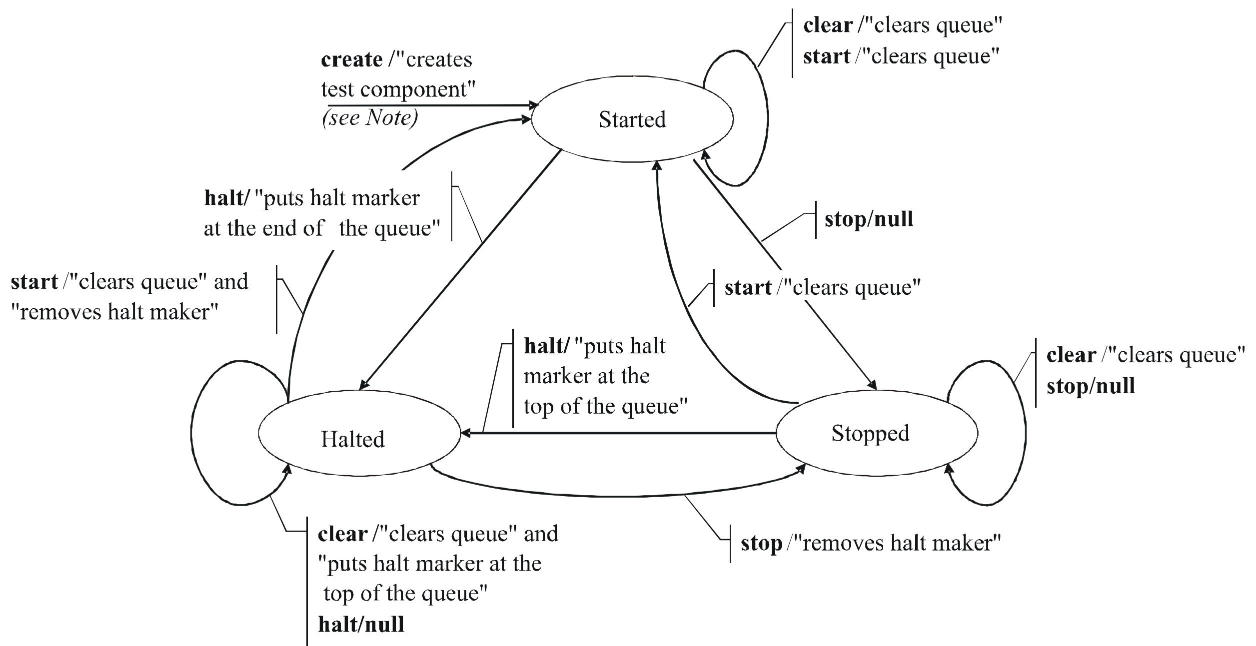


Figure F.6/Z.140 – Dynamic behaviour of ports: port configuration operations

The transitions do not change the main state of the port; i.e., the port remains in the Started or Stopped state.

F.4.2 Port controlling operations

The results of port controlling operations are shown in Figure F.7.

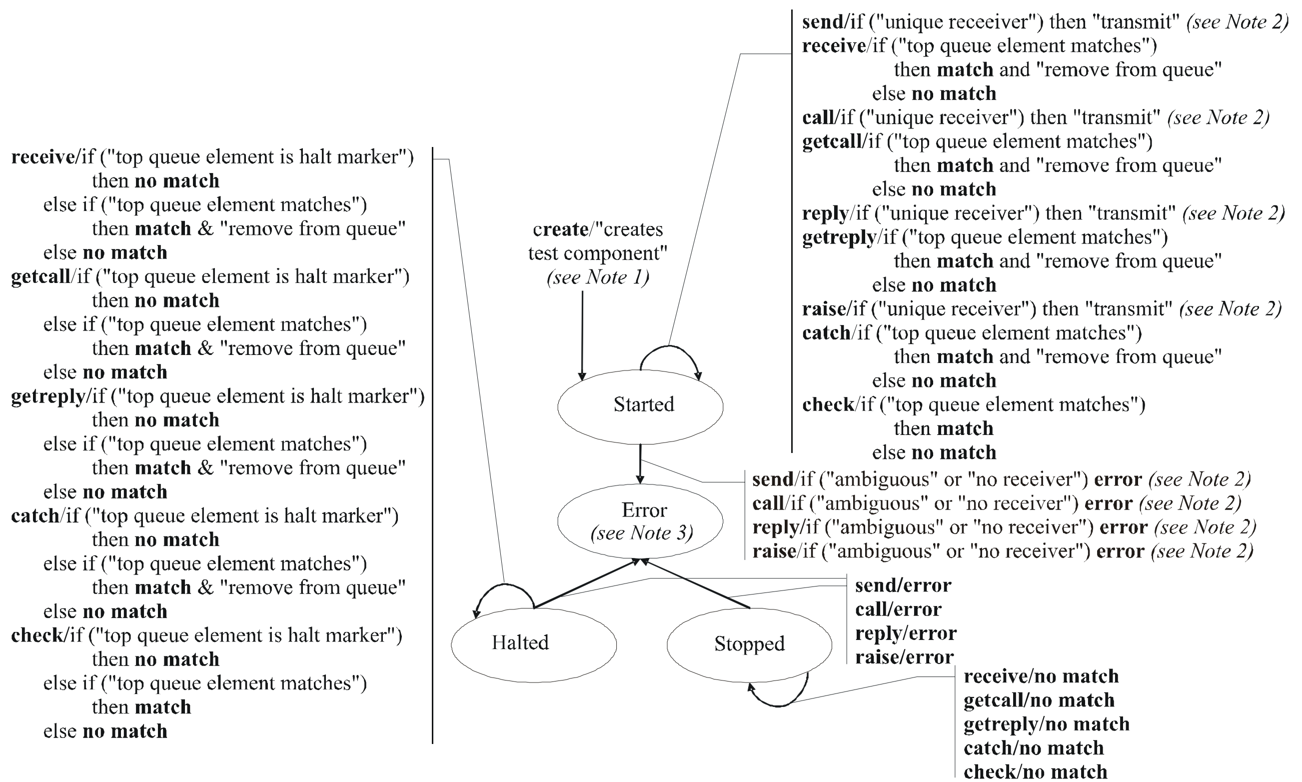


NOTE – When creating a PTC, the ports of that PTC are created and started; when creating the MTC, the ports of the MTC and the ports of the TSI are created and started.

Figure F.7/Z.140 – Dynamic behaviour of ports: port controlling operations

F.4.3 Communication operations

The results of the communication operations send, receive, call, getcall, raise, catch, reply, getreply, check are shown in Figure F.8.



NOTE 1 – When creating a PTC, the ports of that PTC are created and started; when creating a MTC, the ports of the MTC and the ports of the TSI are created and started.
NOTE 2 – A unique receiver exists if there is only one link for this port or if the to address expression references a test component whose port is linked to this port (a terminated test component is not a legal receiver).
NOTE 3 – Whenever a port enters its error state, the test component it belongs to also enters its error state, assigns a local error verdict, the test case terminates and the overall test case result will be an error.
NOTE 4 – As trigger is a shorthand for an alt together with receive, it is not considered here.

Figure F.8/Z.140 – Dynamic behaviour of ports: communication operations

Annex G (informative)

Deprecated language features

G.1 Group style definition of module parameters

The previous version of this Recommendation required the use of a group-like syntax shown in the example below to declare module parameters. The module parameter syntax has been unified with constant and variable declaration syntax in this version but group-like syntax is not fully removed to leave a time period for tool providers and users to change from the old syntax to the new one. The group-like syntax of module parameter declarations is planned to be fully removed in the next published edition of this Recommendation.

EXAMPLE (superfluous syntax):

```
module MyModuleWithParameters
{
  modulepar { integer TS_Par0, TS_Par1 := 0;
              boolean TS_Par2 := true
            };
  modulepar { hexstring TS_Par3 };
}
```

G.2 Recursive import

The previous version of this Recommendation allowed to import named definitions implicitly, via importing other definitions of the same module using them in a recursive mode. This feature is deprecated in this version and is planned to be fully removed in the next published version.

G.3 Using `all` in port type definitions

The previous version of this Recommendation allowed to use the **`all`** keyword in port type definitions instead of an explicit list of types and signatures allowed via the given port. This feature is deprecated in this version and is planned to be fully removed in the next published version.

BIBLIOGRAPHY

- ETSI ES 201 873-1 V1.1.2 (2001-06), *Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3; Part 1: TTCN-3 Core Language*.
- ETSI ES 201 873-1 V2.2.1 (2003-02), *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*.
- ISO/IEC 8859-1:1998, *Information technology – 8-bit single-byte coded graphic character sets – Part 1: Latin alphabet No. 1*.
- Object Management Group (OMG): *The Common Object Request Broker: Architecture and Specification*, Chapter 3 – IDL Syntax and Semantics. Version 2.6, FORMAL/01-12-01, December 2001.
- IEEE 754 (1985), *Binary Floating-Point Arithmetic*.

SERIES OF ITU-T RECOMMENDATIONS

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

Printed in Switzerland
Geneva, 2006