

ETSI ES 201 873-1 V2.2.0 (2002-05)

ETSI Standard

**Methods for Testing and Specification (MTS);
The Testing and Test Control Notation version 3;
Part 1: TTCN-3 Core Language**



Reference

RES/MTS-00063-1 [2]

Keywords

ASN.1, methodology, MTS, testing, TTCN

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Important notice

Individual copies of the present document can be downloaded from:

<http://www.etsi.org>

The present document may be made available in more than one electronic version or in print. In any case of existing or perceived difference in contents between such versions, the reference version is the Portable Document Format (PDF). In case of dispute, the reference shall be the printing on ETSI printers of the PDF version kept on a specific network drive within ETSI Secretariat.

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

<http://portal.etsi.org/tb/status/status.asp>

If you find errors in the present document, send your comment to:

editor@etsi.fr

Copyright Notification

No part may be reproduced except as authorized by written permission.
The copyright and the foregoing restriction extend to reproduction in all media.

© European Telecommunications Standards Institute 2002.
All rights reserved.

DECT™, **PLUGTESTS™** and **UMTS™** are Trade Marks of ETSI registered for the benefit of its Members.
TIPHON™ and the **TIPHON logo** are Trade Marks currently being registered by ETSI for the benefit of its Members.
3GPP™ is a Trade Mark of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners.

Contents

Intellectual Property Rights	11
Foreword.....	11
1 Scope	12
2 References	12
3 Definitions and abbreviations.....	13
3.1 Definitions	13
3.2 Abbreviations	15
4 Introduction	15
4.0 General	15
4.1 The core language and presentation formats	16
4.2 Unanimity of the specification	17
4.3 Conformance	17
5 Basic language elements	17
5.0 General	17
5.1 Ordering of language elements.....	18
5.1.1 Forward references	18
5.2 Parameterization	18
5.2.0 Static and dynamic parameterization	18
5.2.1 Parameter passing by reference and by value	19
5.2.1.0 General	19
5.2.1.1 Parameters passed by reference.....	19
5.2.1.2 Parameters passed by value.....	20
5.2.2 Formal and actual parameter lists	20
5.2.3 Empty formal parameter list	20
5.2.4 Nested parameter lists	20
5.3 Scope rules	21
5.3.0 General.....	21
5.3.1 Scope of formal parameters	22
5.3.2 Uniqueness of identifiers	22
5.4 Identifiers and keywords	23
6 Types and values	23
6.0 General	23
6.1 Basic types and values.....	24
6.1.0 Simple basic types and values.....	24
6.1.1 Basic string types and values	25
6.1.2 Accessing individual string elements.....	27
6.2 Sub-typing of basic types	27
6.2.0 General.....	27
6.2.1 Lists of values	27
6.2.2 Ranges	27
6.2.2.0 General	27
6.2.2.1 Infinite ranges	28
6.2.2.2 Mixing lists and ranges	28
6.2.3 String length restrictions	28
6.3 Structured types and values	28
6.3.0 General.....	28
6.3.1 Record type and values	29
6.3.1.0 General	29
6.3.1.1 Referencing fields of a record type	30
6.3.1.2 Optional elements in a record.....	30
6.3.2 Set type and values	30
6.3.2.0 General	30
6.3.2.1 Referencing fields of a set type	31

6.3.2.2	Optional elements in a set	31
6.3.3	Records and sets of single types	31
6.3.4	Enumerated type and values	32
6.3.5	Unions	33
6.3.5.0	General	33
6.3.5.1	Referencing fields of a union type	34
6.3.5.2	Optionality and union	34
6.4	The anytype	34
6.5	Arrays	34
6.6	Recursive types	35
6.7	Type compatibility	35
6.7.0	General	35
6.7.1	Type compatibility of non-structured types	36
6.7.2	Type compatibility of structured types	36
6.7.2.0	General	36
6.7.2.1	Type compatibility of enumerated types	36
6.7.2.2	Type compatibility of record and record of types	36
6.7.2.3	Type compatibility of union types	38
6.7.2.4	Type compatibility of set and set of types	38
6.7.2.5	Compatibility between slices	39
6.7.3	Type compatibility of communication operations	39
6.7.4	Type conversion	39
7	Modules	39
7.0	General	39
7.1	Naming of modules	39
7.2	Module parameters	40
7.2.0	General	40
7.2.1	Default values for module parameters	40
7.3	Module definitions part	40
7.3.0	General	40
7.3.1	Groups of definitions	41
7.4	Module control part	41
7.5	Importing from modules	42
7.5.0	General	42
7.5.1	Structure of importable definitions	42
7.5.2	Rules on using import	44
7.5.3	Recursive import	46
7.5.4	Importing single definitions	47
7.5.5	Importing all definitions of a module	48
7.5.6	Importing groups	48
7.5.7	Importing definitions of the same kind	49
7.5.8	Handling name clashes on import	49
7.5.9	Handling multiple references to the same definition	50
7.5.10	Import definitions from non-TTCN modules	50
8	Test configurations	50
8.0	General	50
8.1	Port communication model	51
8.2	Restrictions on connections	52
8.3	Abstract test system interface	54
8.4	Defining communication port types	55
8.4.0	General	55
8.4.1	Mixed ports	55
8.5	Defining component types	56
8.5.0	General	56
8.5.1	Declaring local variables and timers in a component	56
8.5.2	Defining components with arrays of ports	57
8.6	Addressing entities inside the SUT	57
8.7	Component references	57
8.8	Defining the test system interface	58
9	Declaring constants	59

10	Declaring variables.....	59
11	Declaring timers	60
11.0	General	60
11.1	Timers as parameters.....	60
12	Declaring messages	60
13	Declaring procedure signatures	60
13.0	General	60
13.1	Signatures for blocking and non-blocking communication.....	61
13.2	Parameters of procedure signatures.....	61
13.3	Value returning remote procedures	61
13.4	Specifying exceptions.....	61
14	Declaring templates	62
14.0	General	62
14.1	Declaring message templates	62
14.1.0	General.....	62
14.1.1	Templates for sending messages.....	62
14.1.2	Templates for receiving messages	63
14.2	Declaring signature templates	63
14.2.0	General.....	63
14.2.1	Templates for invoking procedures	64
14.2.2	Templates for accepting procedure invocations.....	64
14.3	Template matching mechanisms	64
14.4	Parameterization of templates	66
14.4.0	General.....	66
14.4.1	Parameterization with matching attributes.....	67
14.5	Passing templates as parameters.....	67
14.6	Modified templates.....	67
14.6.0	General.....	67
14.6.1	Parameterization of modified templates	68
14.6.2	In-line modified templates	68
14.7	Changing template fields.....	68
14.8	Match Operation.....	69
14.9	Value of Operation	69
15	Operators	69
15.0	General	69
15.1	Arithmetic operators.....	71
15.2	String operators	71
15.3	Relational operators.....	71
15.4	Logical operators.....	73
15.5	Bitwise operators.....	73
15.6	Shift operators	74
15.7	Rotate operators.....	75
16	Functions and altsteps	75
16.1	Functions	75
16.1.0	General.....	75
16.1.1	Parameterization of functions	76
16.1.2	Invoking functions	76
16.1.3	Predefined functions	77
16.2	Altsteps.....	78
16.2.0	General.....	78
16.2.1	Parameterization of altsteps	78
16.2.2	Local definitions in altsteps	78
16.2.2.0	General	78
16.2.2.1	Restrictions for the initialization of local definitions in altsteps.....	79
16.2.3	Invocation of altsteps	79
16.3	Functions and altsteps for different component types	80
17	Test cases.....	80

17.0	General	80
17.1	Parameterization of test cases.....	81
18	Overview of program statements and operations	81
19	Basic program statements.....	83
19.0	General	83
19.1	Expressions.....	83
19.1.0	General.....	83
19.1.1	Boolean expressions	83
19.2	Assignments	83
19.3	The Log statement	83
19.4	The Label statement	84
19.5	The Goto statement	84
19.6	The If-else statement	85
19.7	The For statement.....	86
19.8	The While statement.....	86
19.9	The Do-while statement	86
19.10	The Stop execution statement.....	86
20	Behavioural program statements	87
20.0	General	87
20.1	Alternative behaviour	87
20.1.0	General.....	87
20.1.1	Execution of alternative behaviour	89
20.1.2	Selecting/deselecting an alternative	90
20.1.3	Else branch in alternatives	90
20.1.4	Void	91
20.1.5	Re-evaluation of alt statements.....	91
20.1.6	Invocation of altsteps as alternatives	91
20.2	The Repeat statement	91
20.3	Interleaved behaviour	92
20.4	The Return statement.....	93
21	Default Handling	94
21.0	General	94
21.1	The default mechanism	94
21.2	Default references.....	94
21.3	The activate operation	95
21.3.0	General.....	95
21.3.1	Activation of parameterized altsteps.....	95
21.4	The deactivate operation	95
22	Configuration operations	96
22.0	General	96
22.1	The Create operation	96
22.2	The Connect and Map operations.....	97
22.2.0	General.....	97
22.2.1	Consistent connections and mappings	98
22.3	The Disconnect and Unmap operations.....	98
22.4	The MTC, System and Self operations.....	99
22.5	The Start test component operation	99
22.6	The Stop test component operation	99
22.7	The Running operation	100
22.8	The Done operation	101
22.9	Using component arrays	101
22.10	Summary of the use of any and all with components	102
23	Communication operations.....	102
23.0	General	102
23.1	General format of communication operations	103
23.1.0	General.....	103
23.1.1	General format of the sending operations.....	103
23.1.2	General format of the receiving operations.....	104

23.2	Message-based communication.....	105
23.2.0	General.....	105
23.2.1	The Send operation	105
23.2.2	The Receive operation	106
23.2.2.0	General	106
23.2.2.1	Receive any message.....	107
23.2.2.2	Receive on any port.....	107
23.2.3	The Trigger operation	107
23.2.3.0	General	107
23.2.3.1	Trigger on any message	108
23.2.3.2	Trigger on any port.....	108
23.3	Procedure-based communication.....	108
23.3.0	General.....	108
23.3.1	The Call operation	109
23.3.1.0	General	109
23.3.1.1	Handling responses and exceptions to a Call	110
23.3.1.2	Handling timeout exceptions to the Call	110
23.3.1.3	Calling blocking procedures without return value, out parameters, inout parameters and exceptions.....	111
23.3.1.4	Calling non-blocking procedures	111
23.3.2	The Getcall operation.....	111
23.3.2.0	General	111
23.3.2.1	Accepting any call.....	112
23.3.2.2	Getcall on any port.....	113
23.3.3	The Reply operation.....	113
23.3.4	The Getreply operation	113
23.3.4.0	General	113
23.3.4.1	Get any reply	114
23.3.4.2	Get a reply on any port.....	114
23.3.5	The Raise operation	114
23.3.6	The Catch operation.....	115
23.3.6.0	General	115
23.3.6.1	The Timeout exception	116
23.3.6.2	Catch any exception	116
23.3.6.3	Catch on any port.....	116
23.4	The Check operation	116
23.4.0	General.....	116
23.4.1	The Check any operation	117
23.4.2	Check on any port.....	117
23.5	Controlling communication ports.....	117
23.5.0	General.....	117
23.5.1	The Clear port operation.....	117
23.5.2	The Start port operation	118
23.5.3	The Stop port operation	118
23.6	Use of any and all with ports.....	118
24	Timer operations	118
24.0	General	118
24.1	The Start timer operation.....	119
24.2	The Stop timer operation.....	119
24.3	The Read timer operation.....	120
24.4	The Running timer operation.....	120
24.5	The Timeout operation	120
24.6	Summary of use of any and all with timers	121
25	Test verdict operations	121
25.0	General	121
25.1	Test case verdict.....	121
25.2	Verdict values and overwriting rules.....	122
25.2.0	General.....	122
25.2.1	Error verdict.....	122
26	External actions	122

27	Module control part	123
27.0	General	123
27.1	Execution of test cases	123
27.2	Termination of test cases	123
27.3	Controlling execution of test cases	124
27.4	Test case selection	124
27.5	Use of timers in control	125
28	Specifying attributes	125
28.0	General	125
28.1	Display attributes	126
28.2	Encoding of values	126
28.2.0	General	126
28.2.1	Encode attributes	126
28.2.2	Variant attributes	127
28.2.3	Special strings	127
28.2.4	Invalid encodings	128
28.3	Extension attributes	128
28.4	Scope of attributes	128
28.5	Overwriting rules for attributes	129
28.6	Changing attributes of imported language elements	130
Annex A (normative): BNF and static semantics		131
A.1	TTCN-3 BNF	131
A.1.0	General	131
A.1.1	Conventions for the syntax description	131
A.1.2	Statement terminator symbols	131
A.1.3	Identifiers	131
A.1.4	Comments	131
A.1.5	TTCN-3 terminals	132
A.1.6	TTCN-3 syntax BNF productions	133
A.1.6.0	TTCN module	133
A.1.6.1	Module definitions part	134
A.1.6.1.0	General	134
A.1.6.1.1	Typedef definitions	134
A.1.6.1.2	Constant definitions	135
A.1.6.1.3	Template definitions	135
A.1.6.1.4	Function definitions	137
A.1.6.1.5	Signature definitions	137
A.1.6.1.6	Testcase definitions	137
A.1.6.1.7	Altstep definitions	138
A.1.6.1.8	Import definitions	138
A.1.6.1.9	Group definitions	139
A.1.6.1.10	External function definitions	139
A.1.6.1.11	External constant definitions	139
A.1.6.1.12	Module parameter definitions	139
A.1.6.2	Control part	139
A.1.6.2.0	General	139
A.1.6.2.1	Variable instantiation	140
A.1.6.2.2	Timer instantiation	140
A.1.6.2.3	Component operations	140
A.1.6.2.4	Port operations	140
A.1.6.2.5	Timer operations	142
A.1.6.3	Type	142
A.1.6.4	Value	143
A.1.6.5	Parameterization	144
A.1.6.6	With statement	144
A.1.6.7	Behaviour statements	145
A.1.6.8	Basic statements	145
A.1.6.9	Miscellaneous productions	147
Annex B (normative): Matching incoming values		148

B.1	Template matching mechanisms	148
B.1.0	General	148
B.1.1	Matching specific values	148
B.1.2	Matching mechanisms instead of values	148
B.1.2.0	General.....	148
B.1.2.1	Value list.....	148
B.1.2.2	Complemented value list	149
B.1.2.3	Omitting values.....	149
B.1.2.4	Any value.....	149
B.1.2.5	Any value or none.....	149
B.1.2.6	Value range.....	150
B.1.2.7	SuperSet.....	150
B.1.2.8	SubSet.....	150
B.1.3	Matching mechanisms inside values	151
B.1.3.0	General.....	151
B.1.3.1	Any element.....	151
B.1.3.1.1	Using single character wildcards.....	151
B.1.3.2	Any number of elements or no element	151
B.1.3.2.1	Using multiple character wildcards.....	151
B.1.4	Matching attributes of values	152
B.1.4.0	General.....	152
B.1.4.1	Length restrictions	152
B.1.4.2	The IfPresent indicator.....	152
B.1.5	Matching character pattern	153
B.1.5.0	General.....	153
B.1.5.1	Set expression	154
B.1.5.2	Reference expression	154
B.1.5.3	Match expression n times	154
Annex C (normative): Pre-defined TTCN-3 functions		155
C.1	Integer to character.....	155
C.2	Character to integer	155
C.3	Integer to universal character	155
C.4	Universal character to integer.....	155
C.5	Bitstring to integer.....	155
C.6	Hexstring to integer	156
C.7	Octetstring to integer.....	156
C.8	Charstring to integer.....	156
C.9	Integer to bitstring	156
C.10	Integer to hexstring.....	156
C.11	Integer to octetstring.....	157
C.12	Integer to charstring.....	157
C.13	Length of string type	157
C.14	Number of elements in a structured type.....	158
C.15	The IsPresent function.....	158
C.16	The IsChosen function.....	159
C.17	The Regexp function	159
C.18	Bitstring to charstring.....	159
C.19	Hexstring to charstring	160

C.20	Octetstring to character string	160
C.21	Character string to octetstring	160
C.22	Bitstring to hexstring	160
C.23	Hexstring to octetstring	161
C.24	Bitstring to octetstring	161
C.25	Hexstring to bitstring	161
C.26	Octetstring to hexstring	162
C.27	Octetstring to bitstring	162
C.28	Integer to float	162
C.29	Float to integer	162
C.30	The random number generator function	162
C.31	The Substring function	163
Annex D (normative): Using other data types with TTCN-3		164
D.1	Using ASN.1 with TTCN-3	164
D.1.0	General	164
D.1.1	ASN.1 and TTCN-3 type equivalents	164
D.1.1.0	General	164
D.1.1.1	Identifiers	165
D.1.2	ASN.1 data types and values	165
D.1.2.0	General	165
D.1.2.1	Scope of ASN.1 identifiers	168
D.1.3	Parameterization in ASN.1	168
D.1.4	Defining ASN.1 message templates	169
D.1.4.0	General	169
D.1.4.1	ASN.1 receive messages using the TTCN-3 template syntax	169
D.1.4.2	Ordering of template fields	170
D.1.5	Encoding information	170
D.1.5.0	General	170
D.1.5.1	ASN.1 encoding attributes	170
D.1.5.2	ASN.1 variant attributes	171
Annex E (informative): Library of Useful Types		173
E.1	Limitations	173
E.2	Useful TTCN-3 types	173
E.2.1	Useful simple basic types	173
E.2.1.0	Signed and unsigned single byte integers	173
E.2.1.1	Signed and unsigned short integers	173
E.2.1.2	Signed and unsigned long integers	173
E.2.1.3	Signed and unsigned longlong integers	174
E.2.1.4	IEEE 754 floats	174
E.2.2	Useful character string types	174
E.2.2.0	UTF-8 character string "utf8string"	174
E.2.2.1	BMP character string "bmpstring"	175
E.2.2.2	UTF-16 character string "utf16string"	175
E.2.2.3	ISO/IEC 8859 character string "iso8859string"	175
E.2.3	Useful structured types	176
E.2.3.0	Fixed-point decimal literal	176
Annex F (informative): Bibliography		177
History		178

Intellectual Property Rights

IPRs essential or potentially essential to the present document may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: "*Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards*", which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (<http://webapp.etsi.org/IPR/home.asp>).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Foreword

This ETSI Standard (ES) has been produced by ETSI Technical Committee Methods for Testing and Specification (MTS), and is now submitted for the ETSI standards Membership Approval Procedure.

The present document is part 1 of a multi-part deliverable covering the Testing and Test Control Notation version 3, as identified below:

- Part 1: "TTCN-3 Core Language";**
- Part 2: "TTCN-3 Tabular presentation Format (TFT)";
- Part 3: "TTCN-3 Graphical presentation Format (GFT)";
- Part 4: "TTCN-3 Operational Semantics".

1 Scope

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

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

The present document defines a normative way of using of ASN.1 as defined in the ITU-T Recommendation X.680 series [7], [8], [9] and [10] with TTCN-3. The harmonization of other languages with TTCN-3 is not covered by the present document.

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

2 References

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

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

- [1] ETSI ES 201 873-2 (V2.2.0): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 2: TTCN-3 Tabular Presentation Format (TFT)".
- [2] ETSI TR 101 873-3 (V1.1.2): "Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3; Part 3: TTCN-3 Graphical Presentation Format (GFT)".
- [3] ISO/IEC 9646-1 (1994): "Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 1: General concepts".
- [4] ISO/IEC 9646-3 (1998): "Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation (TTCN)".
- [5] ISO/IEC 646 (1991): "Information technology - ISO 7-bit coded character set for information interchange".
- [6] ISO/IEC 10646: "Information technology - Universal Multiple-Octet Coded Character Set (UCS)".
- [7] ITU-T Recommendation X.680 (1997): "Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation".
- [8] ITU-T Recommendation X.681 (1997): "Information technology - Abstract Syntax Notation One (ASN.1): Information object specification".
- [9] ITU-T Recommendation X.682 (1997): "Information technology - Abstract Syntax Notation One (ASN.1): Constraint specification".
- [10] ITU-T Recommendation X.683 (1997): "Information technology - Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications".

- [11] ITU-T Recommendation X.690 (1997): "Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)".
- [12] ITU-T Recommendation X.691 (1997): "Information technology - ASN.1 encoding rules: Specification of Packed Encoding Rules (PER)".
- [13] ISO/IEC 6429 (1992): "Information technology - Control functions for coded character sets".
- [14] ITU-T Recommendation T.100 (1988): "International information exchange for interactive videotex".
- [15] ITU-T Recommendation T.101 (1994): "International interworking for videotex services".
- [16] ITU-T Recommendation X.660 (1992): "Information technology - Open Systems Interconnection - Procedures for the operation of OSI Registration Authorities: General procedures".

3 Definitions and abbreviations

3.1 Definitions

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

actual parameter: value, template or name reference (identifier) to be passed as 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.

basic types: set of predefined TTCN-3 types described in clauses 6.1.0 and 6.1.1 of ES 201 873-1

NOTE: Basic types are referenced by their names.

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 clause 6.7 are met.

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.

data types : common name for simple basic types, basic string types, structured types, the special data type and all user defined types based on them (see table 3 of ES 201 873-1)

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 another TTCN-3 module

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

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

formal parameter: value or template name 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).

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

Implementation Conformance Statement (ICS): See ISO/IEC-9646-1.

Implementation eXtra Information for Testing (IXIT): See ISO/IEC-9646-1.

Implementation Under Test (IUT): See ISO/IEC-9646-1.

known types: set of defined types, imported ASN.1 and other imported external types

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

Main Test Component (MTC): See ISO/IEC 9646-3.

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.

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 effect on the actual arguments as seen by the caller.

Parallel Test Component (PTC): See ISO/IEC 9646-3.

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

NOTE: In case of types based on imported ASN.1 types, the root type is determined from the associated TTCN-3 type (see clause D.1.2).

static parameterization: kind of parameterization, in which actual parameters are independent of run-time events; i.e. known at compile time or in case of module parameters are 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)

strong typing: strict enforcement of type compatibility by type name equivalence with no exceptions

System Under Test (SUT): See ISO/IEC-9646-1.

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

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

test case: See ISO/IEC-9646-1.

test case error: See ISO/IEC-9646-1.

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

test system: See ISO/IEC-9646-1.

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

type compatibility: language feature, which allows to use values 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 or template shall be compatible with the other type.

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.

user-defined type: type which is defined by subtyping of a basic type, declaring a structured type or constraining the anytype to a single type by the dot notation

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

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

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

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

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.

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

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

4.1 The core language and presentation formats

Historically, TTCN has always been associated with conformance testing. In order to open the language to a wider range of testing applications in both the standards domain and the industrial domain the present document separates the specification of TTCN-3 into several parts. The first part, defined in the present document, is the core language. The second part, defined in ES 201 873-2 [1], is the tabular presentation format, similar in appearance and functionality to earlier versions of TTCN. The third part, defined in TR 101 873-3 [2] is the graphical presentation format. The fourth part contains the operational semantics of the language.

The core language serves three purposes:

- as a generalized text-based test language in its own right;
- as a standardized interchange format of TTCN test suites between TTCN tools;
- as the semantic basis (and where relevant, the syntactical basis) for various presentation formats.

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

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

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

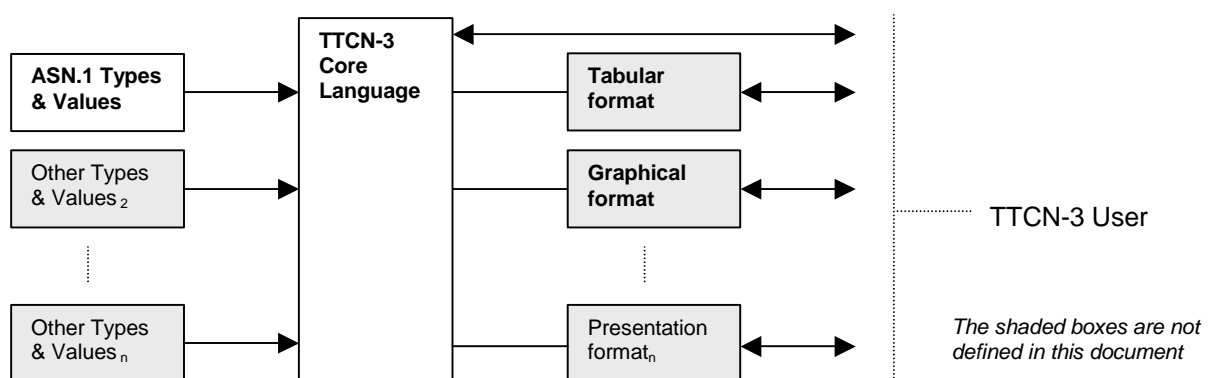


Figure 1: User's view of the core language and the various presentation formats

The core language is defined by a complete syntax (see annex A) and operational semantics (see part 4 of the present document). It contains minimal static semantics (provided in the body of the present document and in annex A) which do not restrict the use of the language due to some underlying application domain or methodology. Functionality of previous versions of TTCN, such as test suite indexes, which can be achieved using proprietary tools is not part of TTCN-3.

4.2 Unanimity of the specification

The language is specified syntactically and semantically in terms of a textual description in the body of the current document (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

The present document does not specify levels of implementation for the language. However, for an implementation claiming to conform to this version of the language, all implemented features of this document shall be consistent with the requirements given in the present document.

NOTE: This does not prevent any conformant implementation to realize extra features not specified in the present document.

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 parameter lists to give a form of test suite parameterization similar to the PICS and PIXIT parameterization mechanisms of TTCN-2.

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

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

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

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: 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
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			
Function definitions	function	Yes			
Altstep definitions	altstep	Yes			
Test case definitions	testcase	Yes			
Variable declarations	var		Yes	Yes	Yes
Timer declarations	timer		Yes	Yes	Yes

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;
    :
}
:
:
```

5.1.1 Forward references

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

Forward references are only allowed for declarations in the module definitions part. Forward references shall never be made inside the module control part, test case definitions, functions and altsteps. This means forward references to local variables, local timers and local constants shall never occur. The only exceptions to this rule are labels. Forward references to labels maybe used in **goto** statements to jump forwards (see clause 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: 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	<i>Values of:</i> all basic types, all user-defined types and address type.
type (note 1)	Static at compile-time	<i>Values of:</i> all basic types, all user-defined types and address type.
template	Dynamic at run-time	<i>Values of:</i> all basic types, all user-defined types, address type and template .
function	Dynamic at run-time	<i>Values of:</i> all basic types, all user-defined types, address type, component type, port type, default , template and timer .
altstep	Dynamic at run-time	<i>Values of:</i> all basic types, all user-defined types, address type, component type, port type, default , template and timer .
testcase	Dynamic at run-time	<i>Values of:</i> all basic types and of all user-defined types, address type and template .
signature	Dynamic at run-time	<i>Values of:</i> all basic types, all user-defined types and address type and component type.
NOTE 1: record of , set of , enumerated , port , component and subtype 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 the present document.		

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 and are identified by the keywords **timer** and **port**. 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** called explicitly, **functions**, **signatures** and **testcase** may contain pass-by-reference parameters;

NOTE: There are further restrictions on how to use pass-by-reference parameters in signatures (see clause 23).

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

EXAMPLE:

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

function MyFunction(out boolean MyReferenceParameter){ ... };
// MyReferenceParameter is passed by reference. The actual 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

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

EXAMPLE:

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

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

```

}

// A 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
{
    :
    TC001(MyTemplate(7));
    :
}

```

5.3 Scope rules

5.3.0 General

TTCN-3 provides six basic units of scope:

- a) module definition part;
- b) control part of a module;
- c) component types;
- d) functions;
- f) test cases;
- e) altsteps;
- g) "blocks of statements and declarations" within compound statements.

NOTE 1: Additional scoping rule for groups are given in clause 7.3.1.

NOTE 2: Additional scoping rule for counters of **for** loops are given in clause 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 definition 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 clause 16.3) by a **runs on**-clause.

Test cases, altsteps and functions are individual scope units without any hierarchical relation between them, i.e. declarations made at the beginning of their body have local visibility and may be used in the given test case, **altstep** or function only (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 relation 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.

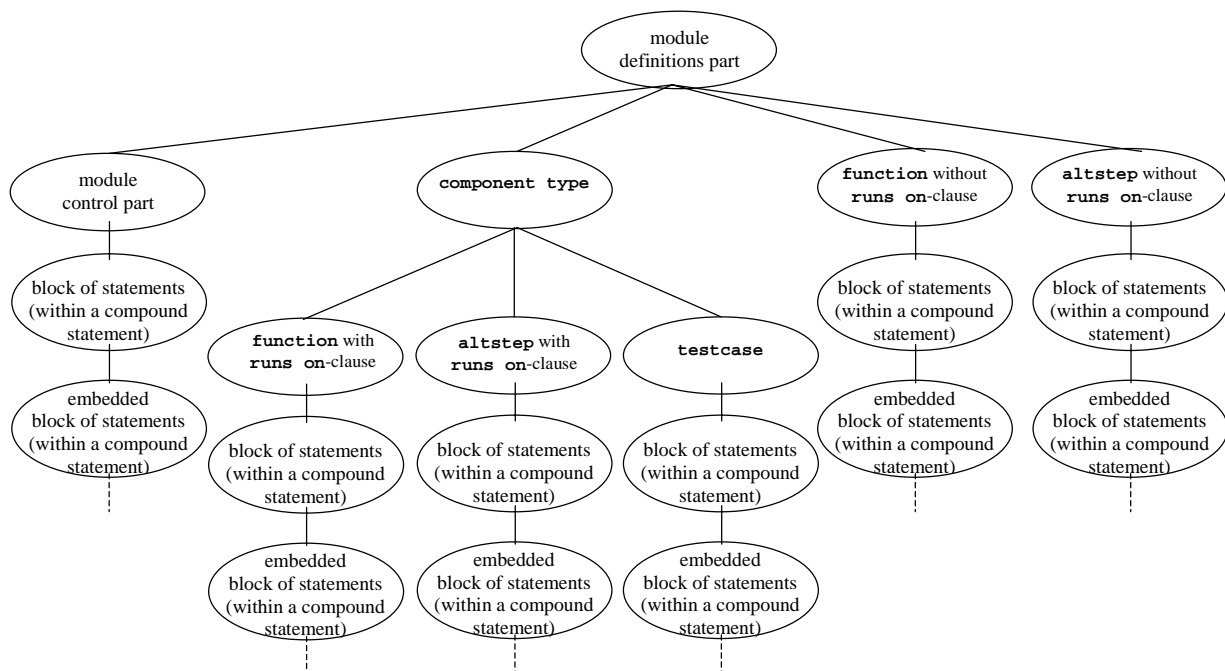


Figure 2: Hierarchy of scope units

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 clause 5.4).

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 reused 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 and TTCN-3 keywords shall be written in all lowercase letters (see annex A). TTCN-3 keywords shall not be used neither 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 **objid** and **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 types 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: Overview of TTCN-3 types

Class of type	Keyword	Sub-type
Simple basic types	integer	range, list
	char	range, list
	universal char	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
	universal charstring	range, list, length
Structured types	record	list
	record of	list, length
	set	list
	set of	list, length
	enumerated	list
	union	list
Special data types	anytype	list
Special configuration types	address	
	port	
	component	
Special default types	default	

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 the positive and negative whole numbers, including zero.

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

- b) **char**: a type whose distinguished values are characters of the version of ISO/IEC 646 [5] complying to the International Reference Version (IRV) as specified in clause 8.2 of ISO/IEC 646 [5].

NOTE 1: The IRV version of ISO/IEC 646 [5] is equivalent to the IRV version of the International Reference Alphabet (former International Alphabet No.5 - IA5), described in ITU-T Recommendation T.50 (see annex F).

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

Relational operators equality (==) and non-equality (!=) can be used to compare values of type **char**.

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

Values of the type **universal char** may be given enclosed in double quotes ("), calculated using a predefined conversion function with the positive integer value of their encoding as argument or by a "quadruple". 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 [6], 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 "ú").

NOTE 2: Control characters can be denoted by using the quadruple form only.

By default **universal char** shall conform to the UCS-4 coded representation form specified in clause 14.2 of ISO/IEC 10646 [6]. This default encoding can be overridden using the defined encoding attributes (see clause 28.2.1).

NOTE 3: UCS-4 is an encoding format, which represents any UCS character on a fixed, 32 bits-length field.

Relational operators equality (==) and non-equality (!=) can be used to compare values of type **universal char**.

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

Floating point numbers are represented as: $\langle mantissa \rangle \times \langle base \rangle^{\langle exponent \rangle}$

Where $\langle mantissa \rangle$ a positive or negative integer, $\langle base \rangle$ a positive integer (in most cases 2, 10 or 16) and $\langle exponent \rangle$ a positive or negative integer.

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

- the normal notation with a dot in a sequence of numbers like, 1.23 (which represents 123×10^{-2}), 2.783 (i.e. 2783×10^{-3}) or -123.456789 (which represents $-123456789 \times 10^{-6}$); 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 12.3×10^4) or -12.3E-4 (which represents -12.3×10^{-4}).

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

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

f) **objid**: a type whose distinguished values are the set of all object identifiers conforming to clause 6.2 of ITU-T Recommendation X.660 [16].

EXAMPLE:

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

or alternatively {itu-t identified-organization etsi}

or alternatively { 0 4 0 }

g) **verdicttype**: a type for use with test verdicts consisting of 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 zeros and ones, 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:

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

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

EXAMPLE 2:

'AB01D'H

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

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

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

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

EXAMPLE 3:

'FF96'O

- d) **charstring**: are types whose distinguished values are zero, one, or more characters of the version of ISO/IEC 646 [5] complying to the International Reference Version (IRV) as specified in clause 8.2 of ISO/IEC 646 [5] (see note 1 of clause 6.1.0 b).

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

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

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

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

The **universal charstring** type can also be denoted by an arbitrary number (possibly zero) of characters from the relevant character set, preceded and followed by double quote (") or by a "quadruple". 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 [6], 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 2: Control characters can be denoted by using the quadruple form only.

By default **universal charstring** shall conform to the UCS-4 coded representation form specified in clause 14.2 of ISO/IEC 10646 [6].

NOTE 3: 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 encoding attributes (see clause 28.2.1). 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 simple basic and basic string types according to table 3.

6.2.1 Lists of values

TTCN-3 permits the specification of a list of distinguished values of any given type as listed in table 3. The values in the list shall be of the 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 universal char SpecialLetter (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 a range of values of type **integer**, **char**, **universal char** and **float** (or derivations of these types). The subtype defined by this range restricts the allowed values of the subtype to the values in the range including the lower boundary and the upper boundary.

EXAMPLE 1:

```
type integer MyIntegerRange (0 .. 255);
type char MyCharRange ("a" .. "z");
type float piRange (3.14 .. 3142E-3);
```

The **char** range specification can also be used within a **charstring** subtype definitions and **universal char** range within the **universal charstring** subtype definitions. In these cases the range restricts the allowed values for each separate character in the strings.

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 specified using
// the double quote notation
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

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

EXAMPLE:

```
type integer MyIntegerRange (1, 2, 3, 10 .. 20, 99, 100);
type char MyCharRange ("a", "b", "c", "0" .. "9");
```

Within **charstring** and **universal charstring** subtype definitions, lists and ranges shall not be mixed in the same subtype definition.

6.2.3 String length restrictions

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

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: Units of length used in field length specifications

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

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

6.3 Structured types and values

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. 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,
    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. Omitting a field causes the value of the relevant field becomes undefined whatever value it has before. The **omit** keyword shall not be used for mandatory fields.

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.

```

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 1:

```

type record MyEmptyRecord { }

```

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

EXAMPLE 2:

```

var integer MyIntegerValue := 1;

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

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

```

Or using an value list.

EXAMPLE 3:

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

Optional fields shall be omitted using the omit symbol.

EXAMPLE 4:

```
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

```

6.3.1.1 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.MyRecord2.MyElement2;
```

6.3.1.2 Optional elements in a record

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

EXAMPLE:

```

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

```

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 clause 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.MySet2.myElement2;
```

6.3.2.2 Optional elements in a set

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

6.3.3 Records and sets of single types

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

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

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 clause 6.5).

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 from 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 notated 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 };

MyVar := MyRecordVar[0]; // the first element of the 'record of' value 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

// The following two assignments

MyRecordVar := { 0, 1, -, 2, omit };
MyRecordVar[6] := 6;

//will result in { 0, 1, <unchanged>, 2, <undefined>, <undefined>, 6 };
// Note also, that the 3rd element would remain undefined if had no assigned value before.
// and the 6th element (with index 5) had no assigned value before this assignment.
```

NOTE: This makes 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 clause 6.5).

EXAMPLE 3:

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

// Then the variable myRecordOfArray will have similar attributes then 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;
```

6.3.4 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 consequently visible within the context of the given type only. Enumeration identifiers shall only be reused 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 clause 5.4.0).

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 also may be used by the system to encode/decode enumerated values. This, however is outside of the scope of this document (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+1};

```

```
ageInMonths := age × 12;
```

This is equivalent to writing:

```
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 clause 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.MyUnion2.myChoice2;
```

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.

6.4 The anytype

The special type **anytype** is defined as a shorthand for the union of all *known types* in a TTCN-3 module. The definition of the term known types is given in clause 3.1.

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

EXAMPLE:

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

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

MyVarThree := MyVarOne × 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: 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 has to 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.

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

EXAMPLE 2:

```
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 x 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.

EXAMPLE 3:

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

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

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

EXAMPLE 4:

```
// 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 value "b" is called the actual value to be assigned, passed as parameter etc., type "B" is called the type of value "b" and type "A" is called the type definition of the value, which is to obtain the actual value of value "b".

6.7.1 Type compatibility of non-structured types

For non-structured variables, constants, templates etc. the value "b" is compatible to type "A" if type "B" resolves to the same root type as type "A" (i.e. **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
```

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, type, and optionality of the fields at the textual order of definition are identical and values of each existing field of the value "b" is compatible with the type of its corresponding field in type "A". Values 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 (0..10) a optional,
  integer (0..10) b optional,
  boolean c
}

type record BType {
  integer a optional,
  integer (0..10) b 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, than the compatibility requirement is always fulfilled.

Values of a **record** type also can 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 equals to the number of element 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 union types

For **union** types the effective value structures shall be compatible between the active field in value "b" and type "A".

EXAMPLE:

```

// Given
type union JType
{
  Atype   A,
  Btype   B,
  Htype   H
};
var JType MyVarJ := { A := { 1, 1, true}};

// Then
MyVarB := MyVarJ
// is a valid assignment as the type of the active field in MyVarJ (Atype) and
// Btype are compatible and the actual values of MyVarJ does not violates subtyping

```

6.7.2.4 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 to **record** and **record of** types.

NOTE: 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.

EXAMPLE:

```

// Given
type set FType {
  integer a optional,
  integer b optional,
  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

```

6.7.2.5 Compatibility between slices

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

EXAMPLE:

```
// If considering 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 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.

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

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

7.1 Naming of modules

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

NOTE 1: The module identifier is the informal text name of the module.

NOTE 2: Module names may differ in the object identifier part only. However, in this case due precaution has to be exercised at import to avoid name clash as prefixing of identifiers (see clause 7.5.8) is unable to resolve such kind of clashes.

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 listing their identifiers and types between a pair of curly brackets following the keyword **parameters**. 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
{
  parameters { integer TS_Par0, TS_Par1; boolean TS_Par2 };
  :
  template MyType Mytemplate
  {
    field TS_Par3
  };
  parameters { hexstring TS_Par3 };
}

```

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

7.2.1 Default values for module parameters

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.

EXAMPLE:

```

module MyModuleDefaultParameter
{
  parameters { integer TS_Par0 := 0, TS_Par1; 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 definition part and imported declarations are given in clause 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 test suite 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 on the same level of hierarchy shall be unique and sub-groups in a lower level of hierarchy shall not have the same name as a group in a higher level of hierarchy. 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 attribute are given in clause 28.4.

EXAMPLE:

```
// 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 repetitious) of the actual test cases. A test case shall be defined in the module definitions part and called in the control part.

EXAMPLE:

```
module MyTestSuite
{ // This module contains definitions ...
  :
  const integer MyConstant := 1;
  type record MyMessageType { ... }
  template MyMessageType MyMessage := { ... }
}
```

```

:
function MyFunction1() { ... }
function MyFunction2() { ... }
:
testcase MyTestcase1() runs on MyMTCType { ... }
testcase MyTestcase2() runs on MyMTCType { ... }
:
// ... and a control part so it is executable
control
{
    var boolean MyVariable; // Local control variable
    :
    MyTestCase1(); // sequential execution of test cases
    MyTestCase2();
    :
}
}

```

7.5 Importing from modules

7.5.0 General

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

If the object identifier is provided as part of the module name (from which the definitions are imported from) in the import statement, this object identifier shall be used to identify the correct module.

All definitions that are imported from one module shall be referenced in one **import** statement only.

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 clause 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:

function	Name MyFunction	Specification (inout MyType1 MyPar) return MyType2 runs on MyCompType	Behaviour description { const MyType3 MyConst := ...; : // further behaviour }
type	record	Name MyRecordType	Specification { field1 MyType4, field2 integer }
template	MyType5	Name MyTemplate	Specification { field1 := 1, field2 := MyConst, // MyConst is a module constant field3 := ModulePar // ModulePar is module parameter }

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	MyType3, 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: 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 definitions* 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 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) As default, the identifiers of referenced definitions are not automatically imported. If the identifiers of the referenced definitions are wished to be implicitly imported, the **recursive** directive (see clause 7.5.3) shall be used.

NOTE 3: If the referenced definitions are wished to be used in the importing module when the default import mechanism is used (e.g. for variable instantiation), it 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.

EXAMPLE:

```

module ModuleONE {
    parameters {
        integer ModPar1, ModPar2 := 7
    }

    type record RecordType_T1 {
        integer Field1_T1,
        boolean Field2_T1
    }

    type record RecordType_T2 {
        MyRecordType_T1 Field1_T2, // Use of RecordType_T1
        MyRecordType_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 {
        parameters 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 inout 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 Recursive import

The TTCN-3 default import mechanism imports referenced definitions without their identifier. This means, a referenced definition cannot be used within the importing module for, e.g. declaring a variable or for being sent over a port. Even though this default import mechanism avoids cluttering up of the name space of the importing module, in some cases it is desired to import all referenced definitions together with their identifiers. In TTCN-3, the **recursive** keyword provides this feature.

On using **import** together with a **recursive** directive the following rules shall be applied:

- a) The rules a), b), c) and f) of clause 7.5.2 remain valid.
- b) A recursively imported definition is imported together with all referenced definitions, i.e. the identifier of all referenced definitions becomes visible and usable in the importing module.

NOTE 1: Recursive import statements are transitive within source modules, e.g. if a module A imports a definition recursively from module B that uses a type T also defined in module B, then type T is automatically imported into module A.

NOTE 2: Recursive import statements are not transitive over module boundaries, e.g. if a module A imports a definition recursively from module B that uses a type T defined in module C, the type T is not automatically imported into module A. Type T has to be imported explicitly from module C, i.e. from its source module.

EXAMPLE:

```

// The module ModuleONE and ModuleTHREE are defined as in the examples
// for clause 7.5.2.

module ModuleFIVE {

    import from ModuleONE recursive {
        template Template_T2
    }

    // The recursive import of Template_T2 will also import the definitions of
    // RecordType_T2, RecordType_T1, MyConst and ModPar1 from ModuleONE. Due to the
    // import of the types RecordType_T2 and RecordType_T1, the field names of these types
    // Field1_T1, Field2_T1, Field1_T2, Field2_T2 and Field3_T3 will become visible
    // in ModuleFIVE

} // end module ModuleFIVE

module ModuleSIX {

    import from ModuleTHREE recursive {
        testcase MyTestCase
    }

```

```

}

// Will cause an ERROR, if the module does not include a further import statement
// that imports RecordType_T2 recursively from ModuleONE! The recursive import of
// MyTestCase from ModuleTHREE requires the recursive import of RecordType_T2 and
// MyCompType from their source modules. The source module of RecordType_T2 is module
// ModuleONE. Even though the source module of MyCompType is ModuleTHREE, its
// recursive import will also cause an error, because this definition also requires
// definitions from ModuleONE.

} // end ModuleSIX

module ModuleSEVEN {

    import from ModuleONE recursive {
        parameters ModPar2;
        type RecordType_T2
    }

    // Imports ModPar2, RecordType_T2 and RecordType_T1 (RecordType_T1 is used by
    // RecordType_T2) from ModuleONE. Through the recursive import the field names
    // Field1_T1, Field2_T1, Field1_T2, Field2_T2 and Field3_T3 of RecordType_T2 and
    // RecordType_T1 will also become visible.

    import from ModuleTHREE recursive {
        testcase MyTestCase
    }

    // Imports MyTestCase, MyCompType (used by MyTestCase) and MyPortType (used by
    // MyCompType) from ModuleTHREE. Through the recursive import of MyTestCase and
    // MyCompType the identifiers MyPar (defined in MyTestCase), MyComponentVar and
    // MyPort (both defined in MyCompType) become visible. Definitions from ModuleOne
    // that are needed for the recursive import of MyTestCase are recursively imported
    // by the previous import statement.

} // end ModuleSEVEN

module ModuleEIGHT {

    import from ModuleONE {
        parameters ModPar2;
        type RecordType_T2
    }

    import from ModuleTHREE recursive {
        testcase MyTestCase
    }

    // Will cause an error, because for the complete recursive import of MyTestCase
    // also the type RecordType_T1 needs to be imported completely from ModuleONE, or
    // in other words, RecordType_T2 needs to be imported recursively.

} // end ModuleEIGHT

```

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 is 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 clause 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.

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, import of groups or import of recursive definitions. Name clashes shall be resolved by prefixing the imported definition (which causes the name clash) by the identifier of the module from which it is imported. The prefix and the identifier shall be separated by a dot (.).

In cases where there are no ambiguities the prefixing need not (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 the present document 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 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. Language identifiers specified for ASN.1 modules are given in clause D.1.

EXAMPLE:

```
import from MyASN1Module language "ASN.1:1997" {
  type MyASN1Type
}
```

NOTE: The import mechanism is designed to allow the re-use of TTCN-3 and ASN.1 definitions from other TTCN-3 or ASN.1 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. Import rules for languages other than TTCN-3 and ASN.1 are not covered by this document.

8 Test configurations

8.0 General

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

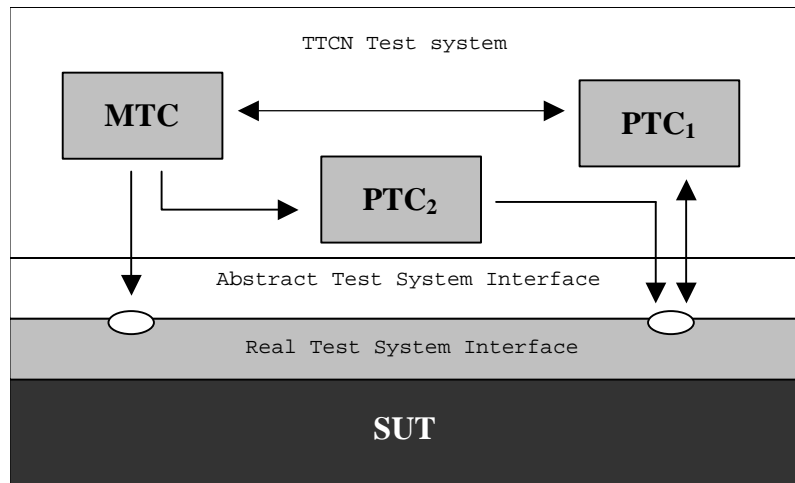


Figure 3: Conceptual view of a typical TTCN-3 test configuration

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 clause 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 clause 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 clause 25.2.1).

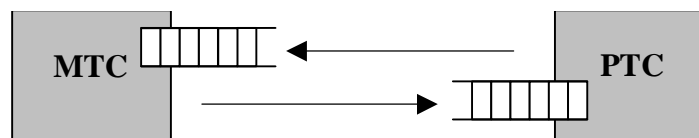


Figure 4: The TTCN-3 communication port model

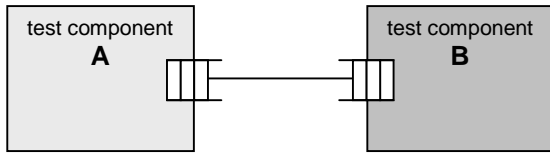
8.2 Restrictions on connections

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)).

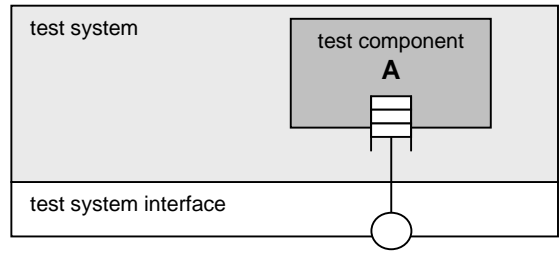
The following connections are not allowed:

- A port owned by a component A shall not be connected with two or more ports owned by the same component (figure 6 (a) and figure 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 figure 6 (b) and figure 6 (d) are not allowed.
- Connections within the test system interface are not allowed (see figure 6 (f)).

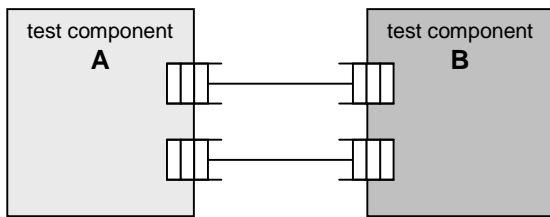
Since TTCN-3 allows dynamic configurations and addresses, the restrictions on connections cannot always be checked at compile-time. The checks shall be made at run-time and shall lead to a test case error when failing.



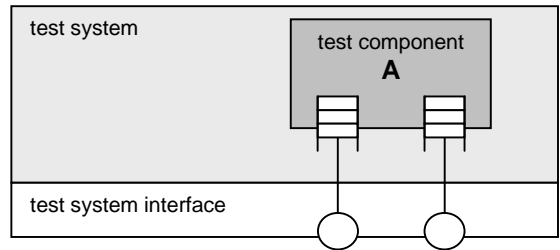
(a)



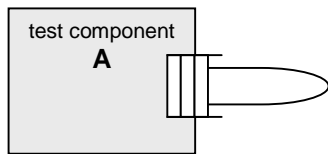
(b)



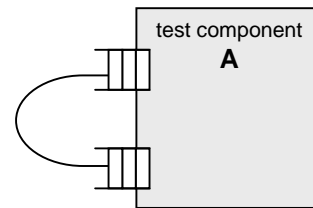
(c)



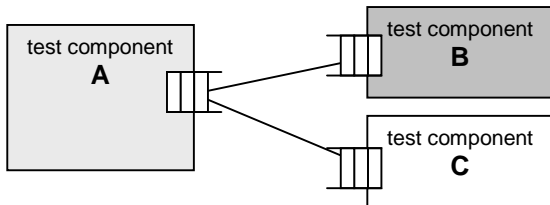
(d)



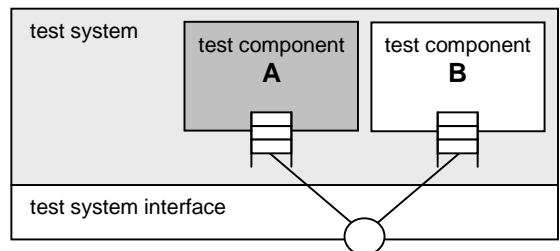
(e)



(f)



(g)



(h)

Figure 5: Allowed connections

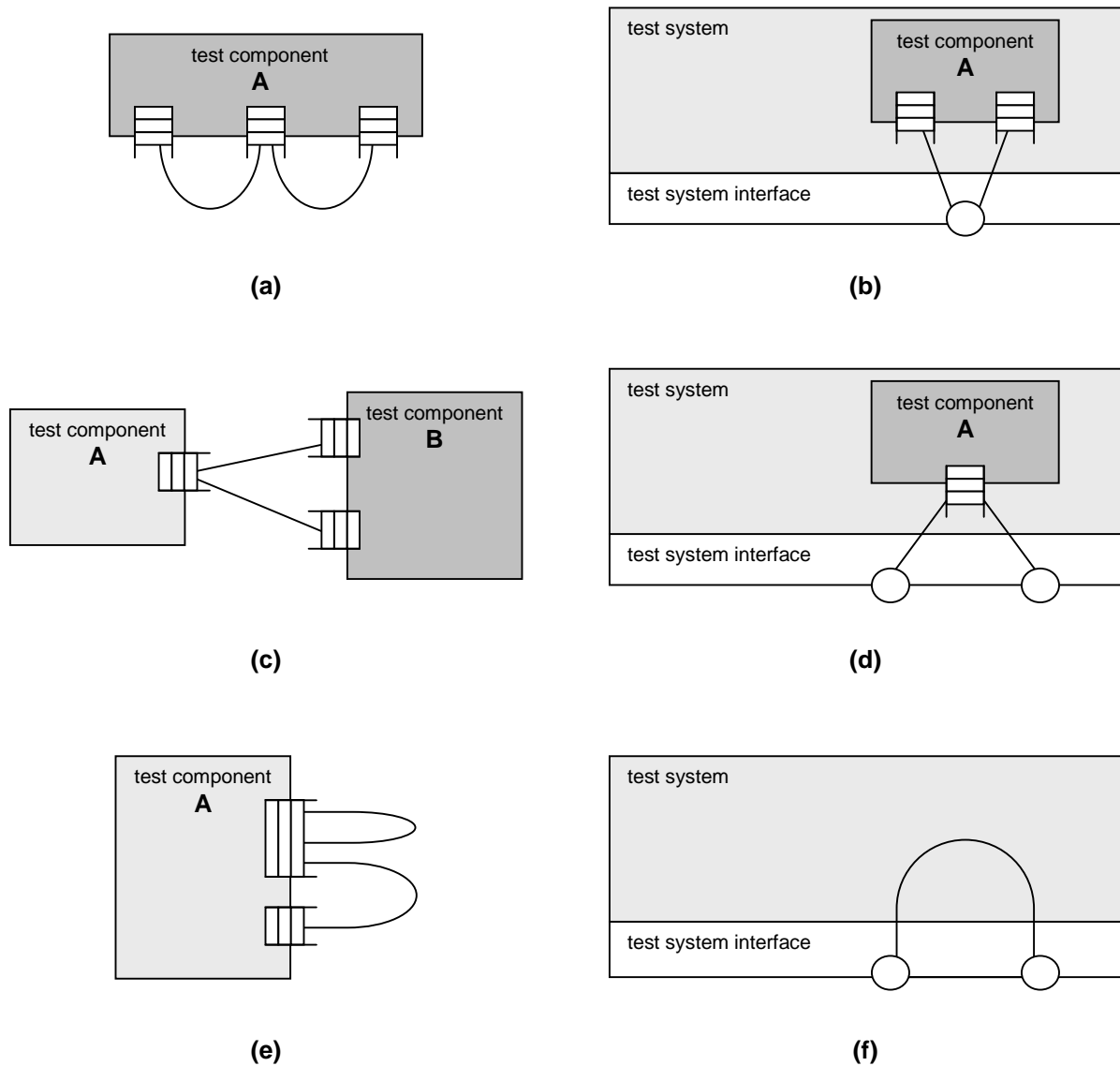


Figure 6: NOT allowed connections

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 the present document the term SUT is used in a general way to mean either SUT or IUT.

In a real test environment test cases need to communicate with the SUT. However, the specification of the real physical connection is outside the scope of TTCN-3. Instead, a well defined (but abstract) test system interface 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 clauses 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 by clause 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 directional. The directions are specified by the keywords **in** (for the in direction), **out** (for the out direction) and **inout** (for both directions). Each port type definition shall have one or more lists indicating the allowed collection of (message) types and/or procedures together with the allowed communication direction.

EXAMPLE 1:

```
// 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.

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

EXAMPLE 2:

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

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. The **all** keyword in this case denotes all types and all procedure signatures defined in the module. No separation is made in the definition.

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

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

```

type port MyAllMixedPortType mixed
{
    inout    all
}

```

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

Operations used to control ports (see clause 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.

NOTE: TTCN-3 differs from TTCN-2 in this respect, where the test configuration is static and declaring test components, PCOs and ASPs implies their automatic connection when the test case execution is initiated.

EXAMPLE:

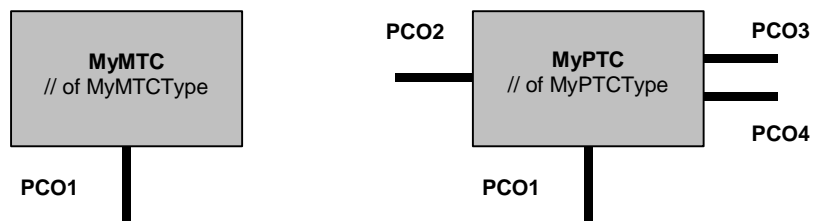


Figure 7: Typical components

```

type component MyMTCType
{
    port MyMessageType PCO1
}

type component MyPTCType
{
    port MyMessageType PCO1, PCO4;
    port MyProcedurePortType PCO2;
    port MyAllMessagesPortType PCO3
}

```

8.5.1 Declaring local variables 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 MyMessageType PCO1
}

```


These declarations are visible to all 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 clause 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 clause 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 clause 22.9).

EXAMPLE:

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

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 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 clause 22.1). In addition component references are returned by the predefined functions **system** (returns the component reference to identify the ports of the test system interface), **mtc** (returns the component reference of the MTC) and **self** (returns the component reference of the component in which **self** is called).

Component references are used in the configuration operations **connect**, **map** and **start** (see clause 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 PC01;
  port PortTypeTwo PC02
}

// Declaring two variables 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:MyPC01, MyCompInst:PC01);
map(MyCompInst:PC02, system:ExtPC01);
MyCompInst.start(MyBehavior(self)); // self is passed as a parameter to MyBehavior

// Usage of component references in from- and to- clauses
MyPC01.receive from MyCompInst;
:
MyPC02.receive(integer?) -> sender MyCompInst;
:
MyPC01.receive(MyTemplate) from MyCompInst;
:
MPC02.send(integer:5) to MyCompInst;

// The following example explains the case of a one-to-many connection at a Port PC01
// 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 {
  [] PC01.receive(M1:?) from MyInst1 -> value MyMessage sender MyInst1 {}
  [] PC01.receive(M1:?) from MyInst2 -> value MyMessage sender MyInst2 {}
  [] PC01.receive(M1:?) from MyInst3 -> value MyMessage sender MyInst3 {}
}
:
MyResult := MyMessageHandling(MyMessage); // some result is retrieved from a function
:
if (MyInst1 != null) {PC01.send(MyResult) to MyInst1};
if (MyInst2 != null) {PC01.send(MyResult) to MyInst2};
if (MyInst3 != null) {PC01.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).

```

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 i.e. when the test case is called from the control part of the module.

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 module definitions part, component type definitions, the module control part, test cases, functions and altsteps. Constant definitions are denoted by the keyword **const**. The value of the constant shall be assigned at the point of declaration.

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

10 Declaring variables

Variables are denoted by the keyword **var**. Variables can be declared and used in 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). A variable declaration may have an optional initial value assigned to it.

EXAMPLE:

```

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

```

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

11 Declaring timers

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 or equal 0.0) where the base unit is seconds.

EXAMPLE:

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

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

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. it needs 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 co-ordination and control messages specific to the relevant test configuration.

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

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 a **noblock** keyword, shall only have **in** parameters (see clause 13.2) and shall have no return value (see clause 13.3), but may raise exceptions (see clause 13.4). By default, signature definitions without **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 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 therefore beyond the scope of the present document.

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

EXAMPLE:

```
signature MyRemoteProcFive (inout float Par1) return integer
exception (ExceptionType1, ExceptionType2);
// MyRemoteProcFive will be used for blocking procedure-based communication. It may return 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 Parl) noblock
    exception (integer, float);
// MyRemoteProcSix will be used for non-blocking procedure-based communication. In case of
// an unsuccessful termination, MyRemoteProcSix may raise 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 provide the following possibilities:

- a) they are a way to organize and to re-use test data, including a simple form of inheritance;
- b) they can be parameterized;
- c) they allow matching mechanisms;
- d) they can be used with either message-based or procedure-based communications.

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

14.1 Declaring message templates

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 the special configuration and default types (**port**, **component**, **address** and **default**).

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** 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.

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 the introduction of clause 14.2 ...

// 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 the introduction of clause 14.2 ...

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

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

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

```

14.3 Template matching mechanisms

Generally, matching mechanisms 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**, **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, objid, 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 (i.e. an expression that evaluates to a specific value);
- b) special symbols that can be used instead of values:
 - (...): a list of values;
 - **complement (...)**: complement of a list of values;
 - **omit**: value is omitted;
 - **?**: wildcard for any value;
 - *****: wildcard for any value or no value at all (i.e. an omitted value);
 - (lower **to** upper): a range of integer values between and including the lower- and upper bounds.
- c) special symbols that can be used *inside* values:
 - **?**: wildcard for any single element in a string, array, **record of** or **set of**;
 - *****: wildcard for any number of consecutive elements in a string, array, **record of** or **set of**, or no element at all (i.e. an omitted element).
- d) special symbols which describe *attributes* of values:
 - **length**: restrictions for strings and arrays;
 - **ifpresent**: for matching of optional field values (if not omitted).

The supported matching mechanisms and their associated symbols (if any) and the scope of their application are shown in table 6. The left-hand column of this table lists all the TTCN-3 and ASN.1 equivalent types as defined in the ITU-T Recommendation X.680 series [7], [8], [9] and [10] to which these matching mechanisms apply. A full description of each matching mechanism can be found in annex B.

Table 6: TTCN-3 Matching Mechanisms

Used with values of	Value	Instead of values								Inside values		Attributes	
		V a l u e L i s t	C o m p l e m e n t e d L i s t	O m i t V a l u e	A n y V a l u e (?)	A n y V a l u e O r N o n e (*)	R a n g e	S u p e r s e t	S u b s e t	A n y E l e m e n t (?)	A n y E l e m e n t s O r N o n e (*)	L e n g t h R e s t r i c t i o n	I f P r e s e n t
boolean	Yes	Yes	Yes	Yes	Yes	Yes							Yes
integer	Yes	Yes	Yes	Yes	Yes	Yes	Yes						Yes
char	Yes	Yes	Yes	Yes	Yes	Yes	Yes						Yes
universal char	Yes	Yes	Yes	Yes	Yes	Yes	Yes						Yes
float	Yes	Yes	Yes	Yes	Yes	Yes	Yes						Yes
bitstring	Yes	Yes	Yes	Yes	Yes	Yes				Yes	Yes	Yes	Yes
octetstring	Yes	Yes	Yes	Yes	Yes	Yes				Yes	Yes	Yes	Yes
hexstring	Yes	Yes	Yes	Yes	Yes	Yes				Yes	Yes	Yes	Yes
character strings	Yes	Yes	Yes	Yes	Yes	Yes	Yes			Yes	Yes	Yes	Yes
record	Yes	Yes	Yes	Yes	Yes	Yes							Yes
record of	Yes	Yes	Yes	Yes	Yes	Yes				Yes	Yes	Yes	Yes
array	Yes	Yes	Yes	Yes	Yes	Yes				Yes	Yes	Yes	Yes
set	Yes	Yes	Yes	Yes	Yes	Yes							Yes
set of	Yes	Yes	Yes	Yes	Yes	Yes		Yes	Yes	Yes	Yes	Yes	Yes
enumerated	Yes	Yes	Yes	Yes	Yes	Yes							Yes
union	Yes	Yes	Yes	Yes	Yes	Yes							Yes

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 clause 5.2.

EXAMPLE:

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

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

14.4.1 Parameterization with matching attributes

To enable matching attributes to be passed as parameters the extra keyword **template** shall be added before the type field. This makes the parameter a template and in effect extends the allowed parameters for the associated type to include the appropriate set of matching attributes (see annex 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,
  field2 := pattern "abc*xyz",
  field3 := true
}

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

14.5 Passing templates as parameters

Only **function**, **testcase**, **altstep** and **template** definitions can have templates as formal parameters.

EXAMPLE:

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

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 definition. In cases where small changes are needed to specify a new template it is possible to specify a modified template. A modified template specifies modifications to particular fields of the original template, either directly or indirectly.

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

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

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

EXAMPLE:

```
// Given
template MyRecordType MyTemplate1 :=
{
  field1 := 123,
  field2 := "A string",
  field3 := true
}
// then writing
template MyRecordType MyTemplate2 modifies MyTemplate1 :=
{
  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
}

```

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;
- d) base template fields containing parameterized templates shall not be modified or explicitly omitted in a modified template.

EXAMPLE:

```

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

// then a modification could be
template MyRecordType MyTemplate2(integer MyPar) modifies MyTemplate1 :=
{
    // field1 is parameterized in Template1 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
pc01.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 to be compared with a template. The operation returns a boolean value. If the types of the template and variable are not compatible (see clause 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;
    :
    PC01.receive(integer?:?) -> value RxValue;

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

```

14.9 Value of Operation

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

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: 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: Precedence of Operators

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

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** 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:

- integer** values gives the whole **integer** part of the value resulting from dividing the first **integer** by the second (i.e. fractions are discarded);
- 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 $x \text{ rem } y$ and $x \text{ mod } y$ 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 $x \text{ rem } y$ and $x \text{ mod } y$ have the same result but for negative arguments they differ.

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

$$\begin{aligned}
 x \text{ rem } y &= x - y * (x/y) \\
 x \text{ mod } y &= x \text{ rem } |y| && \text{if } x \geq 0 \\
 &= 0 && \text{if } x < 0 \text{ and } x \text{ rem } |y| = 0 \\
 &= |y| + x \text{ rem } |y| && \text{if } x < 0 \text{ and } x \text{ rem } |y| < 0
 \end{aligned}$$

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

Table 9: 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 operands only 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 fractions which shall equal at all positions are bits, hexadecimal digits or pairs of hexadecimal digits accordingly.

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 clause 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 field can 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 these chosen fields are compatible and the actual values of the chosen fields 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 SetA conSetA1 := { a1 := 0, a3 := 2 };
// Notice that the order of defining values of the field does not matter
const SetB conSetB1 := { b1 := 0, b3 := 2 };
const SetB conSetB2 := { b2 := 0, b3 := 2 };
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 nor a1 neither a2 equals to its counterpart
// ( not the corresponding element is 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**.

15.5 Bitwise operators

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

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

Their operands shall be of type **bitstring**, **hexstring**, **octetstring**. 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.

NOTE: An error verdict shall be assigned if a system dependent overflow occurs when applying the shift left operation to the left-hand 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. This is denoted by the **return** keyword followed by a type identifier. The keyword **return**, when used in the body of the function with a return type defined in its header, always shall be followed by a value, a constant or variable reference or an expression presenting the return value. The type of the return value shall be compatible with the return 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; // return the integer value 7 when the function terminates
}
```

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

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

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

NOTE 1: The TTCN-3 functions replace Test Steps and Test Suite Procedural Definitions, external functions replace Test Suite Operations in TTCN-2. Informal functions may be declared as external functions with explanatory comments or by using an empty formal function with comments.

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 if all component-wide information used within the function is passed in 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
    PC01.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 clause 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 2: 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 clause 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 parameter. 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 clause 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 the same

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

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

Special restrictions apply to functions bound to test components using the **start** test component operation. These restrictions are described in clause 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: List of TTCN-3 predefined functions

Category	Function	Keyword
Conversion functions	Convert integer value to char value	int2char
	Convert integer value to universal char value	int2uchar
	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 char value to integer value	char2int
	Convert universal char value to integer value	uchar2int
	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 charstring value to integer value	str2int	
Convert charstring value to octetstring value	str2oct	
Length/size functions	Return the length of a value of any string type	lengthof
	Return the number of elements in a record , record of , template , set , set of or array	sizeof
Presence/choice functions	Determine if an optional field in a record , record of , template , set or set of is present	ispresent
	Determine which choice has been made in a union type	ischosen
String functions	Returns part of the input string matching the specified pattern description	regexp
	Returns the specified portion of the input string	substr
Other functions	Generate a random integer 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.

16.2 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 forms 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 PC01, PC02;
  port MyPortTypeTwo PC03;
}

// Altstep definition using PC01, PC02, MyIntVar and MyTimer of MyComponentType

altstep AltSet_A(in integer MyPar1) runs on MyComponentType {
  [] PC01.receive(MyTemplate(MyPar1, MyIntVar) {
    setverdict(inconc);
  }
  [] PC02.receive {
    repeat
  }
  [] MyTimer.timeout {
    setverdict(fail);
    stop
  }
}
```

Altsteps may invoke functions and altsteps or activate altsteps as defaults. An altstep without **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 default shall only have value parameters, i.e. **in** parameters. An altstep that only is 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 clause 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
  [] PC01.receive(MyTemplate(MyPar1, MyLocalVar) {
    setverdict(inconc);
  }
  [] PC02.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 the following operations shall not be called during the initialization of a local definition:

- The **done** operation.
- All port operations, i.e. **start** (port), **stop** (port), **clear**, **send**, **receive**, **trigger**, **call**, **getcall**, **reply**, **getreply**, **raise**, **catch**, **check**.

NOTE 1: The execution of the operations **done**, **start** (port), **stop** (port), **clear**, **receive**, **trigger**, **getcall**, **getreply**, **catch** and **check** can cause conflicts with the actual snapshot. Their execution may remove information from port queues, restrict the access to port queues and/or cause a new snapshot within the evaluation of the actual snapshot.

NOTE 2: The operations **send**, **call**, **reply** and **raise**, shall be avoided for readability purposes, i.e. all communication shall be made explicit and not as a side effect during communication.

- The timer operations **start** (timer), **timeout** and **stop** (timer).

NOTE 3: It is allowed to use the **readtimer** and timer **running** operation.

NOTE 4: These restrictions for the initialization of local definitions in altsteps are the same restrictions as the restrictions to avoid side effects for the Boolean expressions used to select and deselect alternatives in **alt** statement or within altsteps.

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 clause 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 has to 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 alternative.

EXAMPLE 2:

```

:
alt {
  [] PC03.receive {
    ...
  }
  [] AnotherAltStep(); // explicit call of altstep AnotherAltStep as 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 clause 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 clause 20.2) or a continuation immediately after the **alt** statement, i.e. the selected top alternative of the **altstep** ends without explicit **repeat**.

An **altstep** can also be called as 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

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

"Let C1 and C2 be two component types and FT be a function or altstep which refers to C1 in its **runs on** clause. An instance of component type C2 may use FT if the type definition C2 includes the entire type definition of C1. This means, C2 includes at least the same port, variable and timer instances and constant declarations as C1. The "same instance" means here, that both the type and the identifier shall be identical".

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 clause 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 clause 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 basic program statements such as expressions, assignments, loop constructs etc., behavioural statements such as sequential behaviour, alternative behaviour, interleaving, defaults etc., and operations such as **send**, **receive**, **create**, etc.

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

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

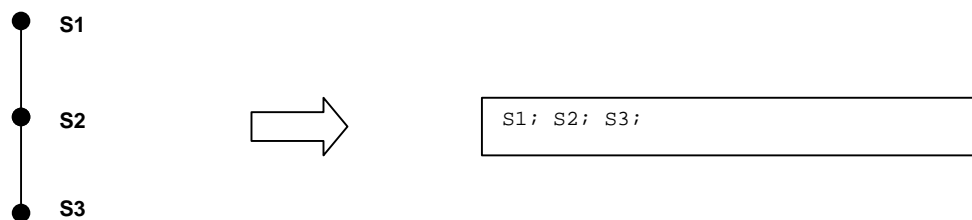


Figure 8: Illustration of sequential behaviour

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: Overview of TTCN-3 statements and operations

Statement	Associated keyword or symbol	Can be used in module control	Can be used in functions, test cases and altsteps
Basic program statements			
Expressions	(...)	Yes	Yes
Assignments	:=	Yes	Yes
Logging	log	Yes	Yes
Label and Goto	label / goto	Yes	Yes
If-else	if (...) {...} else {...}	Yes	Yes
For loop	for (...) {...}	Yes	Yes
While loop	while (...) {...}	Yes	Yes
Do while loop	do {...} while (...)	Yes	Yes
Stop execution	stop	Yes	Yes
Behavioural program statements			
Alternative behaviour	alt {...}	Yes (see note 1)	Yes
Re-evaluation of alternative behaviour	repeat	Yes (see note 1)	Yes
Interleaved behaviour	interleave {...}	Yes (see note 1)	Yes
Returning control	return		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 to component	connect		Yes
Disconnect two components	disconnect		Yes
Map port to test interface	map		Yes
Unmap port from test system interface	unmap		Yes
Get MTC address	mtc		Yes
Get test system interface address	system		Yes
Get own address	self		Yes
Start execution of test component	start		Yes
Stop execution of test component	stop		Yes
Check termination of a PTC	running		Yes
Wait for termination of a PTC	done		Yes
Communication operations			
Send message	send		Yes
Invoke procedure call	call		Yes
Reply to procedure call from remote entity	reply		Yes
Raise exception (to an accepted call)	raise		Yes
Receive message	receive		Yes
Trigger on message	trigger		Yes
Accept procedure call from remote entity	getcall		Yes
Handle response from a previous call	getreply		Yes
Catch exception (from called entity)	catch		Yes
Check (current) message/call received	check		Yes
Clear port	clear		Yes
Clear and give access to port	start		Yes
Stop access (receiving & sending) at port	stop		Yes
Timer operations			
Start timer	start	Yes	Yes
Stop timer	stop	Yes	Yes
Read elapsed time	read	Yes	Yes
Check if timer running	running	Yes	Yes
Timeout event	timeout	Yes	Yes
Verdict operations			
Set local verdict	setverdict		Yes
Get local verdict	getverdict		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.			

19 Basic program statements

19.0 General

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

Table 12: Overview of TTCN-3 basic program statements

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

19.1 Expressions

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 an element of the same 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 a character string to some logging device associated with test control or the test component in which the statement is used.

EXAMPLE:

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

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

19.4 The Label statement

The **label** statement allows the specification of labels in test cases, functions, 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 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);
goto L1; // Jump to label L1
}
}
:
```

19.5 The Goto statement

The **goto** statement can be used in functions, test cases, altsteps and the control part of a TTCN 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 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 statementblock_x refers to a block of statements.

EXAMPLE:

```

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

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

```

A more complex scheme could be:

```

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

```

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

19.7 The For statement

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

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

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 statement which immediately follows the **for** loop.

The index variable of a **for** loop can be declared before being used in the for statement or can be declared and 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 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 test execution. 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{
    :
    stop // stops a test component
  }
  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 clause 22.6).

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: 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:

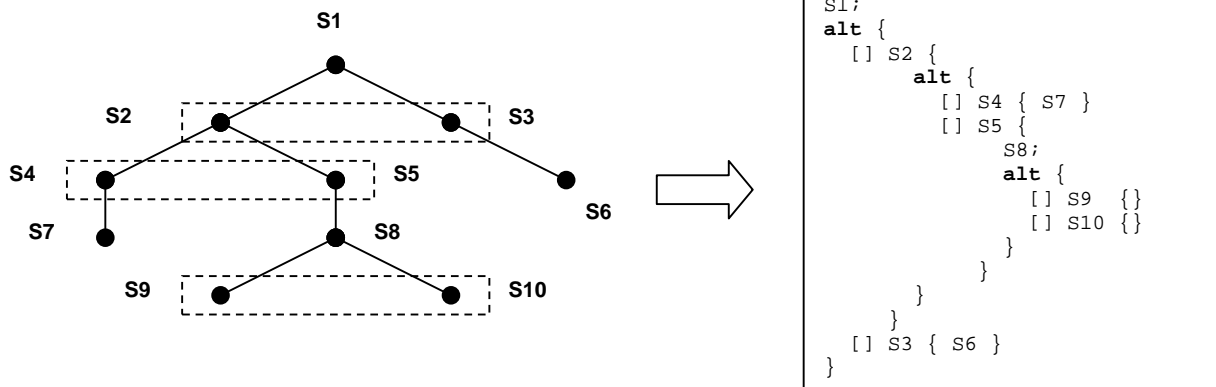


Figure 9: Illustration of alternative behaviour

The **alt** statement denotes branching of test behaviour due to the reception and handling of communication and/or timer events and/or the termination of parallel test components, i.e. it is related to the use of the TTCN-3 operations **receive**, **trigger**, **getcall**, **getreply**, **catch**, **check**, **timeout** and **done**. The **alt** statement denotes a set of possible events that are to be matched against a particular snapshot (see clause 20.1.1).

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

- it is not possible to examine the port queue by using the **boolean** expression and then to disable an alternative;
- It is not possible to call a function as an alternative in the **alt** statement, except in the case where an else guard (i.e. [**else**]) is the last choice in the alternative (see clause 20.1.3).

EXAMPLE:

```

// Use of nested alternative statements
:
alt {
  [] L1.receive(DL_REL_CO:*) {
    setverdict(pass);
    TAC.stop;
    TNOAC.start;
    alt {
      [] L1.receive(DL_EST_IN) {
        TNOAC.stop;
        setverdict(pass);
      }
      [] TNOAC.timeout {
        L1.send(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 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 (part 4 of the present document (see bibliography)).

NOTE 1: Snapshots are only conceptual means for describing the behaviour of the **alt** statement. The concrete algorithms for the snapshot handling can be found in part 4 of the present document (see bibliography).

NOTE 2: The TTCN-3 semantics assumes that taking a snapshot is instantaneously, 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 standard.

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 activation order 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 clause 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*), **timeout** operation (*timeout-branch*) or an receiving operation (*receiving-branch*), i.e. **receive**, **trigger**, **getcall**, **getreply** or **catch**. 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 *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 timeout 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 thus, are 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.

NOTE 4: The repetitive procedure of taking a complete snapshot and re-evaluate 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 standard.

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 (clause 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

The last branch in an **alt** statement can be defined as an else branch by including the **else** keyword between the open and close 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 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)
:
alt {
  [x>1] L2.receive {
    setverdict(pass);
  }
  [x<=1] L2.receive {
    setverdict(inconc);
  }
  [else] {                          // else branch
    MyErrorHandlerling();
    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 as last statement of an **else** branch.

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 clause 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 clause 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 causes the re-evaluation of an **alt** statement, i.e. a new snapshot is taken and the alternatives of the **alt** statement are evaluated in the order of their specification. The **repeat** statement shall only be used as last statement of an alternative in an **alt** statement or as last statement of a top alternative in an altstep definition.

If a **repeat** statement is used as last statement of an alternative in an **alt** statement, it causes a new snapshot and the re-evaluation of the **alt** statement.

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 as last statement of 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 clause 20.1.6).

EXAMPLE 2:

```
// Usage of repeat in an altstep
altstep AnotherAltStep() runs on MyComponentType {
  [] PC01.receive{
    setverdict(inconc);
    repeat           // usage of repeat
  }
  [] PC02.receive {}
}
```

20.3 Interleaved behaviour

The **interleave** statement allows to specify the interleaved occurrence and handling of the statements **done**, **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 part 4 of the present document (see bibliography).

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** 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 part 4 of the present document (see bibliography).

EXAMPLE:

```
// The following TTCN-3 code fragment
:
interleave {
  [] PC01.receive(MySig1)
  { PC01.send(MySig2);
    PC01.receive(MySig3);
  }
  [] PC02.receive(MySig4)
  { PC02.send(MySig5);
    PC02.send(MySig6);
    PC02.receive(MySig7);
  }
}
:

// can be interpreted as a shorthand for
:
alt {
  [] PC01.receive(MySig1)
  { PC01.send(MySig2);
    alt {
      [] PC01.receive(MySig3)
      { PC02.receive(MySig4);
        PC02.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 and returns control to the point from which the function was called. A **return** statement may be optionally associated with a return value. A **return** statement shall only be used in a function.

EXAMPLE:

```

function MyFunction() return boolean {
:
    if (date == "1.1.2000") {
        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 clause 16.2) as defaults. For each test component the defaults, i.e. activated altsteps, are stored in form of a list. The defaults are listed in the order of their activation. The TTCN-3 operations **activate** (see clause 21.3) and **deactivate** (see clause 21.4) operate on the list of defaults. An **activate** appends a new default to the end of the list and a **deactivate** removes a default from the list. A default in the default list can be identified by means of default reference that is generated as a result of the corresponding **activate** operation.

Table 14: 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 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 clause 16.2) defining the default behaviour could be selected, successful means that one of the top alternatives has been selected and executed.

In 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 clause 20.1).

In 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 clause 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 only is responsible for the default handling. The only requirement is that defaults are called in the 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 clause 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 clause 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 append the referenced altstep to 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, a test component cannot activate a default in another test component.

EXAMPLE:

```
:
// 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
:
```

21.3.1 Activation of parameterized altsteps

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

EXAMPLE:

```
:
var default MyDefaultVar := null;
:
MyDefaultVar := activate(MyAltStep2(5, MyVar);
// 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, 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 a not initialized 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 our case MySecondAltStep
           // and MyThirdAltStep
:

```

22 Configuration operations

22.0 General

Configuration operations 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: Overview of TTCN-3 configuration operations

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

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. 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.

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


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

The **create** operation shall return the unique component reference of the newly created instance. The unique reference to the component will typically be stored in a variable (see clause 8.7) and can be used for connecting instances and for communication purposes such as sending and receiving.

Components can be created at any point in a behaviour definition providing full flexibility with regard to dynamic configurations (i.e. any component can create any other 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.

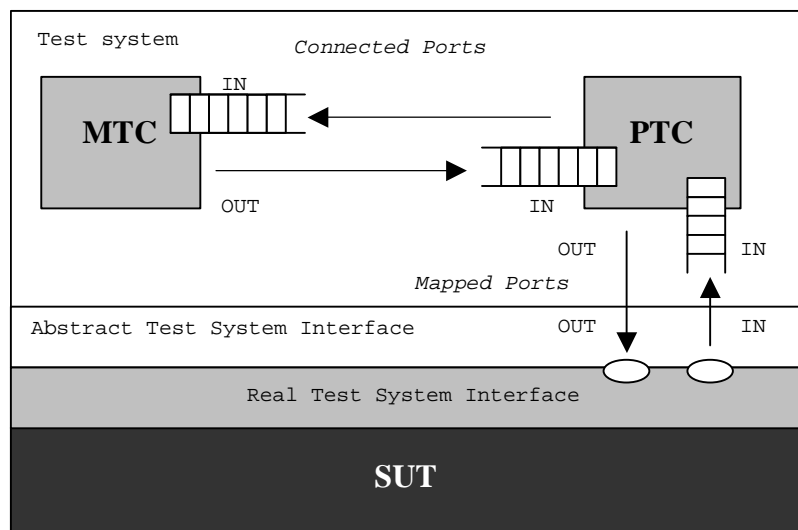


Figure 10: Illustration of the connect and map operations

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 and the operation **system** identifies the test system interface (see clause 22.4). Both 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 PC01 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:PC01);
:
:
// 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 PC01 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 has only an effect if the connection or mapping to be removed has been created beforehand.

EXAMPLE:

```
:
:
connect(MyNewComponent:Port1, mtc:Port3);
map(MyNewComponent:Port2, system:PC01);
:
:
disconnect(MyNewComponent:Port1, mtc:Port3); // disconnect previously made connection
```

```
unmap(MyNewComponent:Port2, system:PC01); // unmap previously made mapping
```

22.4 The MTC, System and Self operations

The component reference (see clause 8.7) has two operations, **mtc** and **system** which return the reference of the master test component and the test system interface respectively. In addition, the operation **self** can be used to return the reference of the component in which it is called.

EXAMPLE:

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

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

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 (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.

EXAMPLE:

```
// It is assumed that the ports Port1, Port2, Port3 and PC01 are properly defined and declared
// in the corresponding port type and component type definitions
:
var MyComponentType MyNewPTC;
:
MyNewPTC := MyComponentType.create; // Creation of a new test component.
:
connect(MyNewPTC:Port1, mtc:Port3); // Connection of the new component with
map(MyNewPTC:Port2, system:PC01); // its environment.
:
:
MyNewPTC.start(MyPTCBehaviour()); // Start of the new 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 the same component type as the newly created component.
- Ports and timers shall not be passed into this function.

NOTE: As **in** and **inout** ports starts listening when the component is created, at the moment, when it starts execution there may be messages in the incoming queues of such ports already waiting to be processed.

22.6 The Stop test component operation

By using the **stop** test component statement a test component can stop its own execution or the execution of another test component. If a component does not stop itself, but 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 itself by using a simple **stop** execution statement (clause 19.10) or by addressing itself in the **stop** operation, e.g. by using the **self** operation. The **stop** test component operation has no arguments.

EXAMPLE 1:

NOTE 1: While the **create**, **start**, **running** and **done** operations can be used for PTC(s) only, the **stop** operation can be used for both the MTC and to PTC(s).

```

var MyComponentType MyComp := MyComponentType.create; // A new test component is created
MyComp.start(CompBehaviour()); // The new component is started
:
if (date == "1.1.2003") {
    MyComp.stop // The new component is stopped on the 1.1.2003
}

:
if ( cond1 ) {
    : // Some behaviour
    self.stop // The test component stops itself by using the self operation
}
else {
    stop // The test component stops itself by using a simple stop operation
}
:

```

If the test component that is stopped is the MTC, all remaining PTCs that are still running shall also be stopped and the test case terminates (see clause 27.2).

NOTE 2: A PTC can stop the test case execution by stopping the MTC.

All resources shall be released when a test component terminates, either explicitly using the **stop** operation or through reaching a **return** statement in the function that originally started the test component or implicitly when the component reaches the end of its behaviour definition. Any variables storing a stopped component reference shall be undefined.

NOTE 3: Undefined means that the value cannot be used for any computation, i.e. it refers to nothing and that its value is not considered to be **null**.

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. In this case, the MTC will not be stopped. It continues its execution after the stop statement.

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 the present document.

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 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 is executing its 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 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 **all** keyword is used with the **done** operation, it will return **true** if no one PTC is executing its behaviour. It also returns true if no PTC has been created or started. Otherwise it returns **false**.

When the **any** keyword is used with the **done** operation, it will return **true** if at least one PTC started but not stopped explicitly by another component is finished executing its behaviour. Otherwise it returns **false**.

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

EXAMPLE:

```

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

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

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

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

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

```

22.9 Using component arrays

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

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 MyPtcType[11];
  :
  for (i:= 0; i<=10; i:=i+1)
  {
    MyPtc [i] := MyPtcType1.create;
    connect(self:PtcCoordination, MyPtcAddresses[i]:MtcCoordination);
    MyPtc [i].start(MyPtcBehaviour());
  }
  :
}
```

22.10 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: Any and All with components

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

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: 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	clear	Yes	Yes
Clear and give access to port	start	Yes	Yes
Stop access (receiving & sending) to port	stop	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 value of the information to be transmitted;
- gives an (optional) address expression that uniquely identifies the communication partner in the case of a one-to-many connection.

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

EXAMPLE 1:

Send part			(Optional) response and exception handling part
Port and operation	Value part	(Optional) address expression	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 expression	
MyP1. call	(MyProc: {MyVar1})		<pre>{ [] MyP1.getreply(MyProc: {MyVar2}) {} [] MyP1.catch(MyProc, ExceptionOne) {} }</pre>

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:

- specifies the port at which the operation shall take place;
- defines a matching part which specifies the acceptable input which will match the statement;
- gives an (optional) address expression that uniquely identifies the communication partner (in case of one-to-many connections).

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

23.2.1 The Send operation

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

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

EXAMPLE 1:

```

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
  
```

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

EXAMPLE 2:

```
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.
```

23.2.2 The Receive operation

23.2.2.0 General

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

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

If the match is not successful, the top message shall not be removed from the port queue i.e. if the **receive** operation is 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 may either be derived from a template or from the value resulting from an expression (which of course can be an explicit value). 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: Encoding attributes are also participating 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 depending 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 and the address of the component that sent the message, can be retrieved and stored in a variable. This is denoted by the symbol '**->**' and the keyword **value**.

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**.

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 MyVar 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.

NOTE: This is equivalent to the TTCN-2 OTHERWISE statement.

A message received by *ReceiveAnyMessage* shall not be assigned to a variable.

EXAMPLE:

```

MyPort.receive;           // Removes the top value from MyPort.

MyPort.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** keyword.

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 do 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: The note to clause 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** keyword.

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 only is 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 23.

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 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.

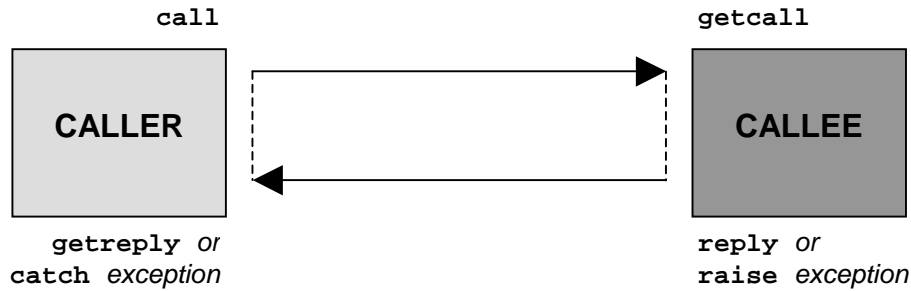


Figure 12: Illustration of blocking procedure-based communication

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.

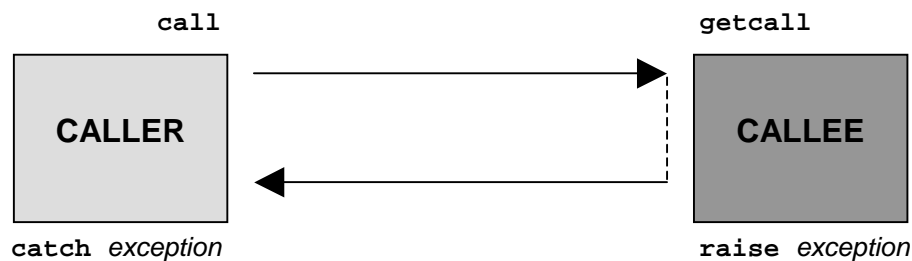


Figure 13: Illustration of non-blocking procedure-based communication

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}) {           // inline 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 or if the **nowait** option is used (see clause 23.3.1.2), the handling of responses or exceptions to **call** operations is done by using **getreply** (see clause 23.3.2) and **catch** (see clause 23.3.6) operations as alternatives in **alt** statements.

In case of blocking procedure-based communication, the handling of response or exception to a call is done in the response and exception handling part of the **call** operation by means of **getreply** (see clause 23.3.2) and **catch** (see clause 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. This means, the usage of **else** branches and the invocation of **altsteps** are 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 clause 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 clause 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** in the 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

```

In where 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** statements.

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(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 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 to 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.

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

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(MyProc3:{5,MyVar} value 20); // Replies to an accepted call of MyProc2.
```

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 **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 MyProc 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 **getcall** 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 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 MyRetVal param(MyPar1,MyPar2);
// The returned value is assigned to variable MyRetVal and the value
// of the two out or inout parameters are assigned to the variables MyPar1 and MyPar2.

MyPort.getreply(MyProc1:{?, ?} value ?) -> value MyRetVal 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 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 a reply message on the top of the incoming port queue (if any) if all other matching criteria are fulfilled. Parameters or return values of responses accepted by *GetAnyReply* shall not be assigned to a variable.

EXAMPLE:

```
MyPort.getreply; // Removes the top 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** keyword.

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.

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

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 MyProc and send it to MyPeer
```

23.3.6 The Catch operation

23.3.6.0 General

The **catch** operation is used to catch exceptions raised by a peer entity as a reaction to a procedure call. The **catch** operation shall only be used at procedure-based (or mixed) ports. The type of the caught exception shall be specified in the signature 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.

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 from MyPartner, assigns its value to MyVar 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 is caught by the **catch** operation. The **timeout** exception is an emergency exit for cases where a called procedure neither replies nor raises an exception within a predetermined time (see clause 23.3.1.2).

EXAMPLE:

```
MyPort.call(MyProc:{5,MyVar}, 20E-3) {
  [] MyPort.getreply(MyProc:{?, ?}) { }
  [] MyPort.catch(timeout) { // timeout exception after 20ms
    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 and without an assignment part. This statement 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 made and the receiving operation specified in the **check** operation is performed on the copy. The **check** operation fails if the receiving function fails i.e. the matching criteria are not fulfilled. In this case the *copy* of the top element of the queue is discarded and test execution continues in the normal manner, i.e. the next statement or alternative to the check operation is evaluated. The **check** operation is successful if the receiving function is successful.

Using the **check** operation in a wrong manner, e.g. check for an exception at a message-based port shall cause a test case error.

NOTE: In most cases the correct usage of the check operation can be checked statically, i.e. before compilation.

EXAMPLE:

```
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 MyPort where the returned value is 20 and
// the values of the two out or inout parameters is 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 *CheckAny* operation allows to distinguish between different senders (in case of one-to-many connections) by using a **from** clause and to retrieve the sender by using a shorthand assignment part with a **sender** clause.

EXAMPLE:

```
MyPort.check;

MyPort.check(from MyPartner);

MyPort.check(-> sender MySenderVar);
```

23.4.2 Check on any port

To **check** on any port use the **any** keyword.

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 an incoming port queue;
- **start**: start listening at and give access to a port;
- **stop**: stop listening and disallow sending operations at a port.

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 action.

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) -> 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 is not executed
```

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: 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		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 has 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 a **timeout** operation is executed.

NOTE 1: The running-timers list and the timeout-list are only a conceptual lists and does not restrict the implementation of timers. Other data structures like 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 into 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: 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 or equal 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
```

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 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. Any entry in a timeout-list for this timer shall be removed from the timeout-list.

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 and to store it into the specified variable. This variable 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 value zero.

24.4 The Running timer operation

The **running** timer operation is used to check whether a timer is listed on the running-timer list of the given scope unit or not (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 of the component or module control are 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).

NOTE: The TTCN-3 **timeout** operation and the TIMEOUT operation in TTCN-2 have identical semantics.

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: 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 **getverdict** and **setverdict** operations respectively. These operations shall only be used in test cases, altsteps and functions.

Table 21: 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 instantiation. It is used to track the individual verdict in each test component (i.e. in the MTC and in each and every PTC).

NOTE: Unlike TTCN-2 no final verdict can be assigned to a test component, therefore assigning a verdict never halts execution of the test component in which the behaviour is executing. If required, this shall be explicitly done by using the **stop** statement.

25.1 Test case verdict

Additionally there is a global verdict that is updated when each test component (i.e. the MTC and each and every PTC) terminates execution. This verdict is not accessible to the **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.

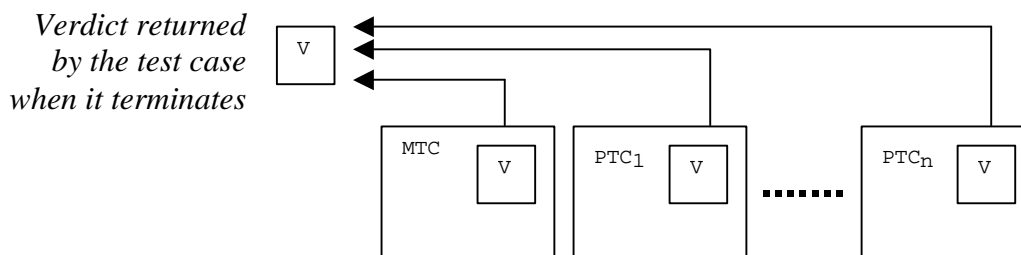


Figure 14: Illustration of the relationship between verdicts

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 clause 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: 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 electrical 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 like the temperature, voltage of the power feeding etc.).

The required action may be defined as a string.

EXAMPLE 1:

```
action("Send MyTemplate on lower PCO"); // Informal description of the SUT action
```

or as a reference to a template which specifies the structure of the message to be sent by the SUT.

EXAMPLE 2:

```
action(MyTemplate); // This is equivalent to the TTCN-2 IMPLICIT SEND statement.
```

In both cases there is no specification of what is done to or by the SUT to trigger this action, only an informal specification of the required action itself.

SUT actions can be specified 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, timers etc. (if any) defined in the control part of a module shall be passed into the test case by parameterization if they are to be used in the behaviour definition of that test case i.e. TTCN-3 does not support global variables or timers 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**, **inconclusive**, **fail** or **error** shall be returned and may be assigned to a variable for further processing.

Optionally, the **execute** statement allows supervision of a test case by means of a timer duration (see clause 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 terminating of the MTC (explicitly or implicitly) all running parallel test components shall be terminated by the test system.

NOTE 1: The concrete mechanism for stopping all PTCs is tool specific and therefore outside the scope of the present document.

The final verdict of a test case is calculated based on the final local verdicts of the different test components according to the rules defined in clause 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** 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 may be run.

EXAMPLE 1:

```

module MyTestSuite () {
  :
  control {
    :
    // Do this test 10 times
    count:=0;
    while (count < 10)
    { execute (MySimpleTestCase1());
      count := count+1;
    }
  }
}

```

If no programming statements are used then, by default, the test cases are executed in the sequential order in which they appear in the module control.

NOTE: This does not preclude the possibility that certain tools may wish to override this default ordering to allow a user or tool to select a different execution order.

Test cases return a single value of type **verdicttype** so it is possible to control the order of execution depending on the outcome of a test case.

EXAMPLE 2:

```

if (execute (MySimpleTestCase()) == pass) { execute (MyGoOnTestCase) }
else { execute (MyErrorRecoveryTestCase) };

```

27.4 Test case selection

Boolean expressions may be used to select and deselect which test cases are to be executed. This includes, of course, the use of functions that return a **boolean** value.

NOTE: This is equivalent to the TTCN-2 named test selection expressions.

EXAMPLE 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 (MySelectionExpression1()) { MyTestCaseGroup2(); }
    :
}
```

27.5 Use of timers in control

Timer may be used to supervise 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 neither declared nor started.

EXAMPLE 1:

```
MyReturnVal := execute (MyTestCase(), 7E-3);
// Where the return verdict will be error if the TestCase does not complete execution
// within 7ms
```

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;
:
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 simply 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;
- c) **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 in, for example, a tabular format.

Special attribute strings related to the display attributes for the tabular (conformance) presentation format can be found in ES 201 873-2 [1].

Special attribute strings related to the display attributes for the graphical presentation format can be found in TR 101 873-3 [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 even may 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.

Special attribute strings related to ASN.1 encoding attributes can be found in annex D.

The manner in which the actual encoding rules are defined (e.g. prose, functions etc.) is outside the scope of the present document. If no specific rules are referenced then encoding shall be a matter for individual implementation.

In most cases encoding attributes will be used in a hierarchical manner. The top-level is the entire module, the next level is a group 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 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.

EXAMPLE:

```

module MyTTCNmodule1
{
  :
  type charstring MyType; // Normally encoded according to the global 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 clause 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 (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 annex F).

The following strings are the predefined (standardized) **variant** attributes for **char**, **universal char**, **charstring** and **universal charstring** (see clause E.2.2):

- a) "UTF-8" means, when applied to universal char and universal charstring types, 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 [6].
- b) "UCS-2" means, when applied to universal char and universal charstring types, that each character of the value shall be individually encoded and decoded according to the UCS-2 coded representation form (see clause 14.1 of ISO/IEC 10646 [6]).
- c) "UTF-16" means, when applied to universal char and universal charstring types, 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 [6].
- d) "8 bit" means, when applied to char, universal char, 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 strings are the predefined (standardized) **variant** attributes for structured types (see clause 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 annex F).

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 clause D.1.5.1) 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.

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 MPDUs will be displayed as PDU and
    extension "MyRule"; // (if not overwritten) have the extension attribute MyRule
}

// is identical to ...
group MyPDUs
{
    type record MyPDU1 { ... } with {display "PDU"; extension "MyRule" }
    type record MyPDU2 { ... } with {display "PDU"; extension "MyRule" }
    group MySpecialPDUs {
        type record MyPDU3 { ... } with {display "PDU"; extension "MySpecialRule" }
        type record MyPDU4 { ... } with {display "PDU"; extension "MySpecialRule" }
    }
}

```

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.

28.6 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 (normative): 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: The syntactic metanotation

::=	is defined to be
abc xyz	abc followed by xyz
	alternative
[abc]	0 or 1 instances of abc
{abc}	0 or more instances of abc
{abc}+	1 or more instances of abc
(...)	textual grouping
Abc	the non-terminal symbol abc
"abc"	a terminal symbol abc

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: 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: List of TTCN-3 terminals which are reserved words

action	fail	noblock	self
activate	false	none	send
address	float		sender
all	for	not	set
alt	from	not4b	setverdict
altstep	function	nowait	signature
and		null	start
and4b	getverdict		stop
any	getcall	objid	subset
anytype	getreply	octetstring	superset
	goto	of	system
bitstring	group	omit	
boolean		on	template
	hexstring	optional	testcase
call		or	timeout
catch	if	or4b	timer
char	ifpresent	out	to
charstring	import	override	trigger
check	in		true
clear	inconc	param	type
complement	infinity	pass	
component	inout	pattern	union
connect	integer	port	universal
const	interleave	procedure	unmap
control			
create	label	raise	value
	language	read	valueof
deactivate	length	receive	var
default	log	record	variant
disconnect			verdicttype
display	map	recursive	
do	match	rem	while
done	message	repeat	with
	mixed	reply	
else	mod	return	
encode	modifies	running	xor
enumerated	module	runs	xor4b
error	mtc		
except			
exception			
execute			
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 module

1. `TTCN3Module ::= TTCN3ModuleKeyword TTCN3ModuleId
BeginChar
[ModuleDefinitionsPart]
[ModuleControlPart]
EndChar
[WithStatement] [SemiColon]`
2. `TTCN3ModuleKeyword ::= "module"`
3. `TTCN3ModuleId ::= ModuleIdentifier [DefinitiveIdentifier]`
4. `ModuleIdentifier ::= Identifier`
5. `DefinitiveIdentifier ::= Dot ObjectIdentifierKeyword "{" DefinitiveObjIdComponentList "}"`
6. `DefinitiveObjIdComponentList ::= {DefinitiveObjIdComponent}+`
7. `DefinitiveObjIdComponent ::= NameForm |
DefinitiveNumberForm |
DefinitiveNameAndNumberForm`
8. `DefinitiveNumberForm ::= Number`
9. `DefinitiveNameAndNumberForm ::= Identifier "(" DefinitiveNumberForm ")"`

A.1.6.1 Module definitions part

A.1.6.1.0 General

```

10. ModuleDefinitionsPart ::= ModuleDefinitionsList
11. ModuleDefinitionsList ::= {ModuleDefinition [SemiColon]}+
12. ModuleDefinition ::= (TypeDef |
    ConstDef |
    TemplateDef |
    ParamDef |
    FunctionDef |
    SignatureDef |
    TestcaseDef |
    AltstepDef |
    ImportDef |
    GroupDef |
    ExtFunctionDef |
    ExtConstDef) [WithStatement]

```

A.1.6.1.1 Typedef definitions

```

13. TypeDef ::= TypeDefKeyword TypeDefBody
14. TypeDefBody ::= StructuredTypeDef | SubTypeDef
15. TypeDefKeyword ::= "type"
16. StructuredTypeDef ::= RecordDef |
    UnionDef |
    SetDef |
    RecordOfDef |
    SetOfDef | EnumDef |
    PortDef |
    ComponentDef
17. RecordDef ::= RecordKeyword StructDefBody
18. RecordKeyword ::= "record"
19. StructDefBody ::= (StructTypeIdentifier [StructDefFormalParList] | AddressKeyword)
    BeginChar
    [StructFieldDef {"," StructFieldDef}]
    EndCharStructTypeIdentifier ::= Identifier
20.
21. StructDefFormalParList ::= "(" StructDefFormalPar {"," StructDefFormalPar} ")"
22. StructDefFormalPar ::= FormalValuePar
/* STATIC SEMANTICS - FormalValuePar shall resolve to an in parameter */
23. StructFieldDef ::= Type StructFieldIdentifier [ArrayDef] [SubTypeSpec] [OptionalKeyword]
24. StructFieldIdentifier ::= Identifier
25. OptionalKeyword ::= "optional"
26. UnionDef ::= UnionKeyword UnionDefBody
27. UnionKeyword ::= "union"
28. UnionDefBody ::= (StructTypeIdentifier [StructDefFormalParList] | AddressKeyword)
    BeginChar
    UnionFieldDef {"," UnionFieldDef}
    EndChar
29. UnionFieldDef ::= Type StructFieldIdentifier [ArrayDef] [SubTypeSpec]
30. SetDef ::= SetKeyword StructDefBody
31. SetKeyword ::= "set"
32. RecordOfDef ::= RecordKeyword [StringLength] OfKeyword StructOfDefBody
33. OfKeyword ::= "of"
34. StructOfDefBody ::= Type (StructTypeIdentifier | AddressKeyword) [SubTypeSpec]
35. SetOfDef ::= SetKeyword [StringLength] OfKeyword StructOfDefBody
36. EnumDef ::= EnumKeyword (EnumTypeIdentifier | AddressKeyword)
    BeginChar
    EnumerationList
    EndChar
37. EnumKeyword ::= "enumerated"
38. EnumTypeIdentifier ::= Identifier
39. EnumerationList ::= Enumeration {"," Enumeration}
40. Enumeration ::= EnumerationIdentifier ["(" Number ")"]
41. EnumerationIdentifier ::= Identifier
42. SubTypeDef ::= Type (SubTypeIdentifier | AddressKeyword) [ArrayDef] [SubTypeSpec]
43. SubTypeIdentifier ::= Identifier
44. SubTypeSpec ::= AllowedValues | StringLength
/* STATIC SEMANTICS - AllowedValues shall be of the same type as the field being subtyped */
45. AllowedValues ::= "(" ValueOrRange {"," ValueOrRange} ")"
46. ValueOrRange ::= RangeDef | SingleConstExpression
/* STATIC SEMANTICS - RangeDef production shall only be used with integer, char, universal char,
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 */
47. RangeDef ::= LowerBound ".." UpperBound

```

```

48. StringLength ::= LengthKeyword "(" SingleConstExpression [".." UpperBound] ")"
49. /* 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) */LengthKeyword ::= "length"
50. PortType ::= [GlobalModuleId Dot] PortTypeIdentifier
51. PortDef ::= PortKeyword PortDefBody
52. PortDefBody ::= PortTypeIdentifier PortDefAttribs
53. PortKeyword ::= "port"
54. PortTypeIdentifier ::= Identifier
55. PortDefAttribs ::= MessageAttribs | ProcedureAttribs | MixedAttribs
56. MessageAttribs ::= MessageKeyword
BeginChar
{MessageList [SemiColon]}+
EndChar
57. MessageList ::= Direction AllOrTypeList
58. Direction ::= InParKeyword | OutParKeyword | InOutParKeyword
59. MessageKeyword ::= "message"
60. AllOrTypeList ::= AllKeyword | TypeList
61. AllKeyword ::= "all"
62. TypeList ::= Type {"," Type}
63. ProcedureAttribs ::= ProcedureKeyword
BeginChar
{ProcedureList [SemiColon]}+
EndChar
64. ProcedureKeyword ::= "procedure"
65. ProcedureList ::= Direction AllOrSignatureList
66. AllOrSignatureList ::= AllKeyword | SignatureList
67. SignatureList ::= Signature {"," Signature}
68. MixedAttribs ::= MixedKeyword
BeginChar
{MixedList [SemiColon]}+
EndChar
69. MixedKeyword ::= "mixed"
70. MixedList ::= Direction ProcOrTypeList
71. ProcOrTypeList ::= AllKeyword | (ProcOrType {"," ProcOrType})
72. ProcOrType ::= Signature | Type
73. ComponentDef ::= ComponentKeyword ComponentTypeIdentifier
BeginChar
[ComponentDefList]
EndChar
74. ComponentKeyword ::= "component"
75. ComponentType ::= [GlobalModuleId Dot] ComponentTypeIdentifier
76. ComponentTypeIdentifier ::= Identifier
77. ComponentDefList ::= {ComponentElementDef [SemiColon]}
78. ComponentElementDef ::= PortInstance | VarInstance | TimerInstance | ConstDef
79. PortInstance ::= PortKeyword PortType PortElement {"," PortElement}
80. PortElement ::= PortIdentifier [ArrayDef]
81. PortIdentifier ::= Identifier

```

A.1.6.1.2 Constant definitions

```

82. ConstDef ::= ConstKeyword Type ConstList
83. ConstList ::= SingleConstDef {"," SingleConstDef}
84. SingleConstDef ::= ConstIdentifier [ArrayDef] AssignmentChar ConstantExpression
/* STATIC SEMANTICS - The Value of the ConstantExpression shall be of the same type as the stated
type for the constant s */
85. ConstKeyword ::= "const"
86. ConstIdentifier ::= Identifier

```

A.1.6.1.3 Template definitions

```

87. TemplateDef ::= TemplateKeyword BaseTemplate [DerivedDef] AssignmentChar TemplateBody
88. BaseTemplate ::= (Type | Signature) TemplateIdentifier ["(" TemplateFormalParList ")"]
89. TemplateKeyword ::= "template"
90. TemplateIdentifier ::= Identifier
91. DerivedDef ::= ModifiesKeyword TemplateRef
92. ModifiesKeyword ::= "modifies"
93. TemplateFormalParList ::= TemplateFormalPar {"," TemplateFormalPar}
94. TemplateFormalPar ::= FormalValuePar | FormalTemplatePar
/* STATIC SEMANTICS - FormalValuePar shall resolve to an in parameter */
95. TemplateBody ::= SimpleSpec | FieldSpecList | ArrayValueOrAttrib
/* STATIC SEMANTICS - Within TeplateBody the ArrayValueOrAttrib can be used for array, record,
record of and set of types. */
96. SimpleSpec ::= SingleValueOrAttrib
97. FieldSpecList ::= "{"[FieldSpec {"," FieldSpec}] "}"
98. FieldSpec ::= FieldReference AssignmentChar TemplateBody
99. FieldReference ::= StructFieldRef | ArrayOrBitRef | ParRef
100. StructFieldRef ::= StructFieldIdentifier

```

```

101. ParRef ::= SignatureParIdentifier
/* OPERATIONAL SEMANTICS - SignatureParIdentifier shall be a formal parameter Identifier from the
associated signature definition */
102. SignatureParIdentifier ::= ValueParIdentifier
103. ArrayOrBitRef ::= [" FieldOrBitNumber "]"
/* STATIC SEMANTICS - ArrayRef shall be optionally used for array types and ASN.1 SET OF and
SEQUENCE OF and TTCN record of and set of. The same notation can be used for a Bit reference inside
an ASN.1 or TTCN bitstring type */
104. FieldOrBitNumber ::= SingleExpression
/* STATIC SEMANTICS - SingleExpression will resolve to a value of integer type */
105. SingleValueOrAttrib ::= MatchingSymbol [ExtraMatchingAttributes] |
SingleExpression [ExtraMatchingAttributes] |
TemplateRefWithParList
/* STATIC SEMANTIC - VariableIdentifier (accessed via singleExpression) may only be used in inline
template definitions to reference variables in the current scope */
106. ArrayValueOrAttrib ::= "{" ArrayElementSpecList "}"
107. ArrayElementSpecList ::= ArrayElementSpec {" "," ArrayElementSpec}
108. ArrayElementSpec ::= NotUsedSymbol | TemplateBody
109. NotUsedSymbol ::= Dash
110. MatchingSymbol ::=
Complement |
AnyValue |
AnyOrOmit |
ValueOrAttribList |
Range |
BitStringMatch |
HexStringMatch |
OctetStringMatch |
CharStringMatch |
SubsetMatch |
SupersetMatch
111. ExtraMatchingAttributes ::= LengthMatch | IfPresentMatch
112. BitStringMatch ::= "'" {BinOrMatch} "'" "B"
113. BinOrMatch ::= Bin | AnyValue | AnyOrOmit
114. HexStringMatch ::= "'" {HexOrMatch} "'" "H"
115. HexOrMatch ::= Hex | AnyValue | AnyOrOmit
116. OctetStringMatch ::= "'" {OctOrMatch} "'" "O"
117. OctOrMatch ::= Oct | AnyValue | AnyOrOmit
118. CharStringMatch ::= PatternKeyword CharStringValue
119. PatternKeyword ::= "pattern"
120. Complement ::= ComplementKeyword ValueList
121. ComplementKeyword ::= "complement"
122. ValueList ::= "(" ConstantExpression {" "," ConstantExpression} ")"
123. SubsetMatch ::= SubsetKeyword ValueList
/* STATIC SEMANTIC - Subset matching shall only be used with the set of type */
124. SubsetKeyword ::= "subset"
125. SupersetMatch ::= SupersetKeyword ValueList
/* STATIC SEMANTIC - Superset matching shall only be used with the set of type */
126. SupersetKeyword ::= "superset"
127. AnyValue ::= "?"
128. AnyOrOmit ::= "*"
129. ValueOrAttribList ::= "(" TemplateBody {" "," TemplateBody}+ ")"
130. LengthMatch ::= StringLength
131. IfPresentMatch ::= IfPresentKeyword
132. IfPresentKeyword ::= "ifpresent"
133. Range ::= "(" LowerBound ".." UpperBound ")"
134. LowerBound ::= SingleConstExpression | Minus InfinityKeyword
135. UpperBound ::= SingleConstExpression | InfinityKeyword
/* STATIC SEMANTICS - LowerBound and UpperBound shall evaluate to types integer, char, universal
char or float. In case LowerBound or UpperBound evaluates to types char or universal char, only
SingleConstExpression may be present */
136. InfinityKeyword ::= "infinity"
137. TemplateInstance ::= InLineTemplate
138. TemplateRefWithParList ::= [GlobalModuleId Dot] TemplateIdentifier [TemplateActualParList]
| TemplateParIdentifier
139. TemplateRef ::= [GlobalModuleId Dot] TemplateIdentifier | TemplateParIdentifier
140. InLineTemplate ::= [(Type | Signature) Colon] [DerivedDef AssignmentChar] TemplateBody
/* STATIC SEMANTICS - The type field may only be omitted when the type is implicitly unambiguous */
141. TemplateActualParList ::= "(" TemplateActualPar {" "," TemplateActualPar} ")"
142. TemplateActualPar ::=
TemplateInstance
/* STATIC SEMANTICS - When the corresponding formal parameter is not of template type the
TemplateInstance production shall resolve to one or more SingleExpressions */
143. TemplateOps ::= MatchOp | ValueofOp
144. 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 */
145. MatchKeyword ::= "match"
146. ValueofOp ::= ValueofKeyword "(" TemplateInstance)"
147. ValueofKeyword ::= "valueof"

```


A.1.6.1.4 Function definitions

```

148. FunctionDef ::= FunctionKeyword FunctionIdentifier
                  "(" [FunctionFormalParList] ")" [RunsOnSpec] [ReturnType]
149.              StatementBlockFunctionKeyword ::= "function"
150. FunctionIdentifier ::= Identifier
151. FunctionFormalParList ::= FunctionFormalPar {"," FunctionFormalPar}
152. FunctionFormalPar ::= FormalValuePar |
                          FormalTimerPar |
                          FormalTemplatePar |
                          FormalPortPar
153. ReturnType ::= ReturnKeyword Type
154. ReturnKeyword ::= "return"
155. RunsOnSpec ::= RunsKeyword OnKeyword ComponentType
156. RunsKeyword ::= "runs"
157. OnKeyword ::= "on"
159. MTCKeyword ::= mtc
158. StatementBlock ::= BeginChar [FunctionStatementOrDefList] EndChar
160. FunctionStatementOrDefList ::= {FunctionStatementOrDef [SemiColon]}+
161. FunctionStatementOrDef ::= FunctionLocalDef |
                              FunctionLocalInst |
                              FunctionStatement
162. FunctionLocalInst ::= VarInstance | TimerInstance
163. FunctionLocalDef ::= ConstDef
164. FunctionStatement ::= ConfigurationStatements |
                          TimerStatements |
                          CommunicationStatements |
                          BasicStatements |
                          BehaviourStatements |
                          VerdictStatements |
                          SUTStatements
165. FunctionInstance ::= FunctionRef "(" [FunctionActualParList] ")"
166. FunctionRef ::= [GlobalModuleId Dot] FunctionIdentifier
167. FunctionActualParList ::= FunctionActualPar {"," FunctionActualPar}
168. 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

```

169. SignatureDef ::= SignatureKeyword SignatureIdentifier
                  "(" [SignatureFormalParList] ")" [ReturnType | NoBlockKeyword]
                  [ExceptionSpec]
170. SignatureKeyword ::= "signature"
171. SignatureIdentifier ::= Identifier
172. SignatureFormalParList ::= SignatureFormalPar {"," SignatureFormalPar}
173. SignatureFormalPar ::= FormalValuePar
174. ExceptionSpec ::= ExceptionKeyword "(" ExceptionTypeList ")"
175. ExceptionKeyword ::= "exception"
176. ExceptionTypeList ::= Type {"," Type}
177. NoBlockKeyword ::= "noblock"
178. Signature ::= [GlobalModuleId Dot] SignatureIdentifier

```

A.1.6.1.6 Testcase definitions

```

179. TestcaseDef ::= TestcaseKeyword TestcaseIdentifier
                  "(" [TestcaseFormalParList] ")" ConfigSpec
180.              StatementBlockTestcaseKeyword ::= "testcase"
181. TestcaseIdentifier ::= Identifier
182. TestcaseFormalParList ::= TestcaseFormalPar {"," TestcaseFormalPar}
183. TestcaseFormalPar ::= FormalValuePar |
                          FormalTemplatePar
184. ConfigSpec ::= RunsOnSpec [SystemSpec]
185. SystemSpec ::= SystemKeyword ComponentType
186. SystemKeyword ::= "system"
188. 187. TestcaseInstance ::= ExecuteKeyword "(" TestcaseRef "(" [TestcaseActualParList] ")" ["",
TimerValue] ")" ExecuteKeyword ::= "execute"
189. TestcaseRef ::= [GlobalModuleId Dot] TestcaseIdentifier
190. TestcaseActualParList ::= TestcaseActualPar {"," TestcaseActualPar}
191. 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

```
192. AltstepDef ::= AltstepKeyword AltstepIdentifier
    ("["AltstepFormalParList] ") [RunsOnSpec]
    BeginChar
    AltstepLocalDefList
    AltGuardList
    EndChar
193. AltstepKeyword ::= "altstep"
194. AltstepIdentifier ::= Identifier
195. AltstepFormalParList ::= FunctionFormalParList
/* STATIC SEMANTICS - all formal parameter must be value parameters i.e., in parameters */
196. AltstepLocalDefList ::= {AltstepLocalDef [SemiColon]}
197. AltstepLocalDef ::= VarInstance | TimerInstance | ConstDef
198. AltstepInstance ::= AltstepRef "(" [FunctionActualParList] ")"
199. AltstepRef ::= [GlobalModuleId Dot] AltstepIdentifier
```

A.1.6.1.8 Import definitions

```
200. ImportDef ::= ImportKeyword ImportFromSpec (AllWithExceptions | (BeginChar ImportSpec EndChar))
201. ImportKeyword ::= "import"
202. AllWithExceptions ::= AllKeyword [ExceptionsDef]
203. ExceptionsDef ::= ExceptionKeyword BeginChar ExceptionSpec EndChar
204. ExceptionKeyword ::= "except"
205. ExceptionSpec ::= {ExceptionElement [SemiColon]}
/* STATIC SEMANTICS: Any of the production components (ExceptionGroupSpec, ExceptionTypeDefSpec
etc.) may be present only once in the ExceptionSpec production */
206. ExceptionElement ::= ExceptionGroupSpec |
    ExceptionTypeDefSpec |
    ExceptionTemplateSpec |
    ExceptionConstSpec |
    ExceptionTestcaseSpec |
    ExceptionAltstepSpec |
    ExceptionFunctionSpec |
    ExceptionSignatureSpec |
    ExceptionParamSpec
207. ExceptionGroupSpec ::= GroupKeyword (ExceptionGroupRefList | AllKeyword)
208. ExceptionTypeDefSpec ::= TypeDefKeyword (TypeRefList | AllKeyword)
209. ExceptionTemplateSpec ::= TemplateKeyword (TemplateRefList | AllKeyword)
210. ExceptionConstSpec ::= ConstKeyword (ConstRefList | AllKeyword)
211. ExceptionTestcaseSpec ::= TestcaseKeyword (TestcaseRefList | AllKeyword)
212. ExceptionAltstepSpec ::= AltstepKeyword (AltstepRefList | AllKeyword)
213. ExceptionFunctionSpec ::= FunctionKeyword (FunctionRefList | AllKeyword)
214. ExceptionSignatureSpec ::= SignatureKeyword (SignatureRefList | AllKeyword)
215. ExceptionParamSpec ::= ParamKeyword (ParamRefList | AllKeyword)
216. ImportSpec ::= {ImportElement [SemiColon]}
217. ImportElement ::= ImportGroupSpec |
    ImportTypeDefSpec |
    ImportTemplateSpec |
    ImportConstSpec |
    ImportTestcaseSpec |
    ImportAltstepSpec |
    ImportFunctionSpec |
    ImportSignatureSpec |
    ImportParamSpec
218. ImportFromSpec ::= FromKeyword ModuleId [RecursiveKeyword]
219. ModuleId ::= GlobalModuleId [LanguageSpec]
/* STATIC SEMANTICS - LanguageSpec may only be omitted if the referenced module contains TTCN-3
notation */
220. LanguageKeyword ::= "language"
221. LanguageSpec ::= LanguageKeyword FreeText
222. GlobalModuleId ::= ModuleIdentifier [Dot ObjectIdentifierValue]
223. RecursiveKeyword ::= "recursive"
224. ImportGroupSpec ::= GroupKeyword (GroupRefList | AllGroupsWithException)
225. GroupRefList ::= FullGroupIdentifier {"", " FullGroupIdentifier})
226. AllGroupsWithException ::= AllKeyword [ExceptionKeyword GroupRefList]
FullGroupIdentifier := GroupIdentifier {Dot GroupIdentifier} [ExceptionSpec]
227. ExceptionGroupRefList ::= ExceptionFullGroupIdentifier {"", " ExceptionFullGroupIdentifier})
228. ExceptionFullGroupIdentifier ::= GroupIdentifier {Dot GroupIdentifier}
228'. ImportTypeDefSpec ::= TypeDefKeyword (TypeRefList | AllTypesWithException)
229. TypeRefList ::= TypeDefIdentifier {"", " TypeDefIdentifier}
230. AllTypesWithException ::= AllKeyword [ExceptionKeyword TypeRefList]
231. TypeDefIdentifier ::= StructTypeIdentifier |
```

```

EnumTypeIdentifier |
PortTypeIdentifier |
ComponentTypeIdentifier |
SubTypeIdentifier
232. ImportTemplateSpec ::= TemplateKeyword (TemplateRefList | AllTemplsWithException)
233. TemplateRefList ::= TemplateIdentifier {"", " TemplateIdentifier}
234. AllTemplsWithException ::= AllKeyword [ExceptionKeyword TemplateRefList]
235. ImportConstSpec ::= ConstKeyword (ConstRefList | AllConstsWithException)
236. ConstRefList ::= ConstIdentifier {"", " ConstIdentifier}
237. AllConstsWithException ::= AllKeyword [ExceptionKeyword ConstRefList]
238. ImportAltstepSpec ::= AltstepKeyword (AltstepRefList | AllAltstepsWithException)
239. AltstepRefList ::= AltstepIdentifier {"", " AltstepIdentifier}
240. AllAltstepsWithException ::= AllKeyword [ExceptionKeyword AltstepRefList]
241. ImportTestcaseSpec ::= TestcaseKeyword (TestcaseRefList | AllTestcasesWithException)
242. TestcaseRefList ::= TestcaseIdentifier {"", " TestcaseIdentifier}
243. AllTestcasesWithException ::= AllKeyword [ExceptionKeyword TestcaseRefList]
244. ImportFunctionSpec ::= FunctionKeyword (FunctionRefList) | AllFunctionsWithException)
245. FunctionRefList ::= FunctionIdentifier {"", " FunctionIdentifier}
246. AllFunctionsWithException ::= AllKeyword [ExceptionKeyword FunctionRefList]
247. ImportSignatureSpec ::= SignatureKeyword (SignatureRefList | AllSignaturesWithException)
248. SignatureRefList ::= SignatureIdentifier {"", " SignatureIdentifier}
249. AllSignaturesWithException ::= AllKeyword [ExceptionKeyword SignatureRefList]
250. ImportParamSpec ::= ParamKeyword (ParamRefList | AllParamWithException)
251. ParamRefList ::= ModuleParIdentifier {"", " ModuleParIdentifier}
252. AllParamWithException ::= AllKeyword [ExceptionKeyword ParamRefList]

```

A.1.6.1.9 Group definitions

```

253. GroupDef ::= GroupKeyword GroupIdentifier
    BeginChar
    [ModuleDefinitionsPart]
    EndChar
254. GroupKeyword ::= "group"
255. GroupIdentifier ::= Identifier

```

A.1.6.1.10 External function definitions

```

256. ExtFunctionDef ::= ExtKeyword FunctionKeyword ExtFunctionIdentifier ("["FunctionFormalParList]
    ")" [ReturnType]
257. ExtKeyword ::= "external"
258. ExtFunctionIdentifier ::= Identifier

```

A.1.6.1.11 External constant definitions

```

259. ExtConstDef ::= ExtKeyword ConstKeyword Type ExtConstIdentifier
260. ExtConstIdentifier ::= Identifier

```

A.1.6.1.12 Module parameter definitions

```

261. ParamDef ::= ParamKeyword "{" ModuleParList "}"
262. ParamKeyword ::= "parameters"
263. ModuleParList ::= ModulePar {SemiColon ModulePar}
264. ModulePar ::= ModuleParType ModuleParIdentifier [AssignmentChar ConstantExpression] {"", "
    ModuleParIdentifier [AssignmentChar ConstantExpression]}
265. /* STATIC SEMANTICS - The Value of the ConstantExpression shall be of the same type as the
    stated type for the Parameter */ModuleParType ::= Type
266. ModuleParIdentifier ::= Identifier

```

A.1.6.2 Control part

A.1.6.2.0 General

```

267. ModuleControlPart ::= ControlKeyword
    BeginChar
    ModuleControlBody
    EndChar
    [WithStatement] [SemiColon]
268. ControlKeyword ::= "control"
269. ModuleControlBody ::= [ControlStatementOrDefList]
271. 270. ControlStatementOrDefList ::= {ControlStatementOrDef
    [SemiColon]}+ControlStatementOrDef ::= FunctionLocalInst |
    ControlStatement |
    FunctionLocalDef
272. ControlStatement ::= TimerStatements |
    BasicStatements |
    BehaviourStatements |

```

SUTStatements

A.1.6.2.1 Variable instantiation

```

273. VarInstance ::= VarKeyword Type VarList
274. VarList ::= SingleVarInstance {"," SingleVarInstance}
275. SingleVarInstance ::= VarIdentifier [ArrayDef] [AssignmentChar VarInitialValue]
276. VarInitialValue ::= Expression
277. VarKeyword ::= "var"
278. VarIdentifier ::= Identifier
279. VariableRef ::= (VarIdentifier | ValueParIdentifier) [ExtendedFieldReference]

```

A.1.6.2.2 Timer instantiation

```

280. TimerInstance ::= TimerKeyword TimerList
281. TimerList ::= SingleTimerInstance{"," SingleTimerInstance}
282. SingleTimerInstance ::= TimerIdentifier [ArrayDef] [AssignmentChar TimerValue]
283. TimerKeyword ::= "timer"
284. TimerIdentifier ::= Identifier
285. TimerValue ::= SingleExpression
/* STATIC SEMANTICS - SingleExpression must resolve to a value of type float */
286. TimerRef ::= TimerIdentifier [ArrayOrBitRef]|
    TimerParIdentifier [ArrayOrBitRef]

```

A.1.6.2.3 Component operations

```

287. ConfigurationStatements ::=    ConnectStatement |
    MapStatement |
    DisconnectStatement |
    UnmapStatement |
    DoneStatement |
    StartTCStatement |
    StopTCStatement
288. ConfigurationOps ::= CreateOp | SelfOp | SystemOp | MTCOp | RunningOp
289. CreateOp ::= ComponentType Dot CreateKeyword
290. SystemOp ::= SystemKeyword
291. SelfOp ::= "self"
292. MTCOp ::= MTCKeyword
293. DoneStatement ::= ComponentId Dot DoneKeyword
294. ComponentId ::= ComponentIdentifier | (AnyKeyword | AllKeyword) ComponentKeyword
295. DoneKeyword ::= "done"
296. RunningOp ::= ComponentId Dot RunningKeyword
297. RunningKeyword ::= "running"
298. CreateKeyword ::= "create"
299. ConnectStatement ::= ConnectKeyword PortSpec
300. ConnectKeyword ::= "connect"
301. PortSpec ::= "(" PortRef "," PortRef ")"
302. PortRef ::= ComponentRef Colon Port
303. ComponentRef ::= ComponentIdentifier | SystemOp | SelfOp | MTCOp
304. DisconnectStatement ::= DisconnectKeyword PortSpec
305. DisconnectKeyword ::= "disconnect"
306. MapStatement ::= MapKeyword PortSpec
307. MapKeyword ::= "map"
308. UnmapStatement ::= UnmapKeyword PortSpec
309. UnmapKeyword ::= "unmap"
310. StartTCStatement ::= ComponentIdentifier Dot StartKeyword "(" FunctionInstance ")"
/* STATIC SEMANTICS the Function instance may only have in parameters */

/* STATIC SEMANTICS the Function instance shall not have timer parameters */
311. StartKeyword ::= "start"
312. StopTCStatement ::= StopKeyword | ComponentIdentifier Dot StopKeyword |
    AllKeyword ComponentKeyword Dot StopKeyword
313. ComponentIdentifier ::= VariableRef | FunctionInstance
/* STATIC SEMANTICS the variable associated with VariableRef or the Return type associated with
FunctionInstance must be of component type */

```

A.1.6.2.4 Port operations

```

314. Port ::= (PortIdentifier | PortParIdentifier) [ArrayOrBitRef]
315. CommunicationStatements ::=    SendStatement |
    CallStatement |
    ReplyStatement |
    RaiseStatement |
    ReceiveStatement |
    TriggerStatement |
    GetCallStatement |

```

```

        GetReplyStatement |
        CatchStatement |
        CheckStatement |
        ClearStatement |
        StartStatement |
        StopStatement
316. SendStatement ::= Port Dot PortSendOp
317. PortSendOp ::= SendOpKeyword "(" SendParameter ")" [ToClause]
318. SendOpKeyword ::= "send"
319. SendParameter ::= TemplateInstance
320. ToClause ::= ToKeyword AddressRef
321. ToKeyword ::= "to"
322. AddressRef ::= VariableRef | FunctionInstance
/* STATIC SEMANTICS - VariableRef and FunctionInstance return must be of address or component type
*/
323. CallStatement ::= Port Dot PortCallOp [PortCallBody]
324. PortCallOp ::= CallOpKeyword "(" CallParameters ")" [ToClause]
325. CallOpKeyword ::= "call"
326. CallParameters ::= TemplateInstance ["," CallTimerValue]
/* STATIC SEMANTICS only out parameters may be omitted or specified with a matching attribute */
327. CallTimerValue ::= TimerValue | NowaitKeyword
/* STATIC SEMANTICS Value must be of type float */
328. NowaitKeyword ::= "nowait"
329. PortCallBody ::= BeginChar
        CallBodyStatementList
        EndChar
330. CallBodyStatementList ::= {CallBodyStatement [SemiColon]}+
331. CallBodyStatement ::= CallBodyGuard StatementBlock
332. CallBodyGuard ::= AltGuardChar CallBodyOps
333. CallBodyOps ::= GetReplyStatement | CatchStatement
334. ReplyStatement ::= Port Dot PortReplyOp
335. PortReplyOp ::= ReplyKeyword "(" TemplateInstance [ReplyValue]" )" [ToClause]
336. ReplyKeyword ::= "reply"
337. ReplyValue ::= ValueKeyword Expression
338. RaiseStatement ::= Port Dot PortRaiseOp
339. PortRaiseOp ::= RaiseKeyword "(" Signature "," TemplateInstance ")" [ToClause]
340. RaiseKeyword ::= "raise"
341. ReceiveStatement ::= PortOrAny Dot PortReceiveOp
342. PortOrAny ::= Port | AnyKeyword PortKeyword
343. PortReceiveOp ::= ReceiveOpKeyword [ "(" ReceiveParameter ")" ] [FromClause] [PortRedirect]
/* STATIC SEMANTICS: the PortRedirect option may only be present if the ReceiveParameter option is
also present */
344. ReceiveOpKeyword ::= "receive"
345. ReceiveParameter ::= TemplateInstance
346. FromClause ::= FromKeyword AddressRef
347. FromKeyword ::= "from"
348. PortRedirect ::= PortRedirectSymbol (ValueSpec [SenderSpec] | SenderSpec)
349. PortRedirectSymbol ::= "->"
350. ValueSpec ::= ValueKeyword VariableRef
351. ValueKeyword ::= "value"
352. SenderSpec ::= SenderKeyword VariableRef
/* STATIC SEMANTIC Variable ref must be of address or component type */
353. SenderKeyword ::= "sender"
354. TriggerStatement ::= PortOrAny Dot PortTriggerOp
355. PortTriggerOp ::= TriggerOpKeyword [ "(" ReceiveParameter ")" ] [FromClause] [PortRedirect]
/* STATIC SEMANTICS: the PortRedirect option may only be present if the ReceiveParameter option is
also present */
356. TriggerOpKeyword ::= "trigger"
357. GetCallStatement ::= PortOrAny Dot PortGetCallOp
358. PortGetCallOp ::= GetCallOpKeyword [ "(" ReceiveParameter ")" ] [FromClause]
[PortRedirectWithParam]
/* STATIC SEMANTICS: the PortRedirectWithParam option may only be present if the ReceiveParameter
option is also present */
359. GetCallOpKeyword ::= "getcall"
360. PortRedirectWithParam ::= PortRedirectSymbol RedirectSpec
361. RedirectSpec ::= ValueSpec [ParaSpec] [SenderSpec] |
        ParaSpec [SenderSpec] |
        SenderSpec
362. ParaSpec ::= ParaKeyword ParaAssignmentList
363. ParaKeyword ::= "param"
364. ParaAssignmentList ::= "(" (AssignmentList | VariableList) ")"
365. AssignmentList ::= VariableAssignment {"," VariableAssignment}
366. VariableAssignment ::= VariableRef AssignmentChar ParameterIdentifier
/* STATIC SEMANTICS: the parameterIdentifiers must be from the corresponding signature definition */
367. ParameterIdentifier ::= ValueParIdentifier |
        TimerParIdentifier |
        TemplateParIdentifier |
        PortParIdentifier

```

```

368. VariableList ::= VariableEntry {"," VariableEntry}
369. VariableEntry ::= VariableRef | NotUsedSymbol
370. GetReplyStatement ::= PortOrAny Dot PortGetReplyOp
371. PortGetReplyOp ::=      GetReplyOpKeyword ["(" ReceiveParameter [ValueMatchSpec] ")"]
      [FromClause] [PortRedirectWithParam]
/* STATIC SEMANTICS: the PortRedirectWithParam option may only be present if the ReceiveParameter
option is also present */
372. GetReplyOpKeyword ::= "getreply"
373. ValueMatchSpec ::= ValueKeyword TemplateInstance
374. CheckStatement ::= PortOrAny Dot PortCheckOp
375. PortCheckOp ::= CheckOpKeyword ["(" CheckParameter ")"]
376. CheckOpKeyword ::= "check"
377. CheckParameter ::=      CheckPortOpsPresent | FromClausePresent | RedirectPresent
378. CheckPortOpsPresent ::= CheckPortOps [FromClause] [PortRedirectSymbol SenderSpec]
379. FromClausePresent ::= FromClause [PortRedirectSymbol SenderSpec]
380. RedirectPresent ::= PortRedirectSymbol SenderSpec
381. CheckPortOps ::= PortReceiveOp | PortGetCallOp | PortGetReplyOp | PortCatchOp
382. CatchStatement ::= PortOrAny Dot PortCatchOp
383. PortCatchOp ::= CatchOpKeyword ["(" CatchOpParameter ")"] [FromClause] [PortRedirect]
/* STATIC SEMANTICS: the PortRedirect option may only be present if the CatchOpParameter option is
also present */
384. CatchOpKeyword ::= "catch"
385. CatchOpParameter ::= Signature "," TemplateInstance | TimeoutKeyword
386. ClearStatement ::= PortOrAll Dot PortClearOp
387. PortOrAll ::= Port | AllKeyword PortKeyword
388. PortClearOp ::= ClearOpKeyword
390. 389. ClearOpKeyword ::= clearStartStatement ::= PortOrAll Dot PortStartOp
391. PortStartOp ::= StartKeyword
392. StopStatement ::= PortOrAll Dot PortStopOp
393. PortStopOp ::= StopKeyword
395. StopKeyword ::= "stop"
394. AnyKeyword ::= "any"

```

A.1.6.2.5 Timer operations

```

396. TimerStatements ::= StartTimerStatement | StopTimerStatement | TimeoutStatement
397. TimerOps ::= ReadTimerOp | RunningTimerOp
398. StartTimerStatement ::= TimerRef Dot StartKeyword ["(" TimerValue ")"]
399. StopTimerStatement ::= TimerRefOrAll Dot StopKeyword
400. TimerRefOrAll ::= TimerRef | AllKeyword TimerKeyword
401. ReadTimerOp ::= TimerRef Dot ReadKeyword
402. ReadKeyword ::= "read"
403. RunningTimerOp ::= TimerRefOrAny Dot RunningKeyword
404. TimeoutStatement ::= TimerRefOrAny Dot TimeoutKeyword
405. TimerRefOrAny ::= TimerRef | AnyKeyword TimerKeyword
406. TimeoutKeyword ::= "timeout"

```

A.1.6.3 Type

```

407. Type ::= PredefinedType | ReferencedType
408. PredefinedType ::= BitStringKeyword |
      BooleanKeyword |
      CharStringKeyword |
      UniversalCharString |
      CharKeyword |
      UniversalChar |
      IntegerKeyword |
      OctetStringKeyword |
      ObjectIdentifierKeyword |
      HexStringKeyword |
      VerdictTypeKeyword |
      FloatKeyword |
      AddressKeyword |
      DefaultKeyword |
      AnyTypeKeyword
409. BitStringKeyword ::= "bitstring"
410. BooleanKeyword ::= "boolean"
411. IntegerKeyword ::= "integer"
412. OctetStringKeyword ::= "octetstring"
413. ObjectIdentifierKeyword ::= "objid"
414. HexStringKeyword ::= "hexstring"
415. VerdictTypeKeyword ::= "verdicttype"
416. FloatKeyword ::= "float"
417. AddressKeyword ::= "address"
418. DefaultKeyword ::= "default"
419. AnyTypeKeyword ::= "anytype"
420. CharStringKeyword ::= "charstring"

```

```

421. UniversalCharString ::= UniversalKeyword CharStringKeyword
422. UniversalKeyword ::= "universal"
423. CharKeyword ::= "char"
424. UniversalChar ::= UniversalKeyword CharKeyword
425. ReferencedType ::= [GlobalModuleId Dot] TypeReference [ExtendedFieldReference]
426. TypeReference ::= StructTypeIdentifier[TypeActualParList] |
    EnumTypeIdentifier |
    SubTypeIdentifier |
    ComponentTypeIdentifier
427. TypeActualParList ::= "(" TypeActualPar {" ," TypeActualPar } ")"
428. TypeActualPar ::= ConstantExpression
429. ArrayDef ::= {"[" ArrayBounds [".." ArrayBounds] "]" }+
430. ArrayBounds ::= SingleConstExpression
/* STATIC SEMANTICS - ArrayBounds will resolve to a non negative value of integer type */

```

A.1.6.4 Value

```

431. Value ::= PredefinedValue | ReferencedValue
432. PredefinedValue ::= BitStringValue |
    BooleanValue |
    CharStringValue |
    IntegerValue |
    OctetStringValue |
    ObjectIdentifierValue |
    HexStringValue |
    VerdictTypeValue |
    EnumeratedValue |
    FloatValue |
    AddressValue |
    OmitValue
433. BitStringValue ::= Bstring
434. BooleanValue ::= "true" | "false"
435. IntegerValue ::= Number
436. OctetStringValue ::= Ostring
437. ObjectIdentifierValue ::= ObjectIdentifierKeyword {" ObjIdComponentList "}
/* ReferencedValue must be of type object Identifier */
438. ObjIdComponentList ::= {ObjIdComponent}+
439. ObjIdComponent ::= NameForm |
    NumberForm |
    NameAndNumberForm
440. NumberForm ::= Number | ReferencedValue
/* STATIC SEMANTICS - referencedValue must be of type integer and have a non negative Value */
441. NameAndNumberForm ::= Identifier NumberForm
442. NameForm ::= Identifier
443. HexStringValue ::= Hstring
444. VerdictTypeValue ::= "pass" | "fail" | "inconc" | "none" | "error"
445. EnumeratedValue ::= EnumerationIdentifier
446. CharStringValue ::= Cstring | Quadruple
447. Quadruple ::= CharKeyword "(" Group "," Plane "," Row "," Cell ")"
448. Group ::= Number
449. Plane ::= Number
450. Row ::= Number
451. Cell ::= Number
452. FloatValue ::= FloatDotNotation | FloatENotation
453. FloatDotNotation ::= Number Dot DecimalNumber
454. FloatENotation ::= Number [Dot DecimalNumber] Exponential [Minus] Number
455. Exponential ::= "E"
456. ReferencedValue ::= ValueReference [ExtendedFieldReference]
457. ValueReference ::= [GlobalModuleId Dot] (ConstIdentifier | ExtConstIdentifier) |
    ValueParIdentifier |
    ModuleParIdentifier |
    VarIdentifier
458. Number ::= (NonZeroNum {Num}) | "0"
459. NonZeroNum ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
460. DecimalNumber ::= {Num}
461. Num ::= "0" | NonZeroNum
462. Bstring ::= "" {Bin} "" "B"
463. Bin ::= "0" | "1"
464. Hstring ::= "" {Hex} "" "H"
465. Hex ::= Num | "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c" | "d" | "e" | "f"
466. Ostring ::= "" {Oct} "" "O"
467. Oct ::= Hex Hex
468. Cstring ::= "" {Char} ""
469. Char ::= /* REFERENCE - A character defined by the relevant CharacterString type. For
charstring a character from the character set defined in ISO/IEC 646. For universal charstring a
character from any character set defined in ISO/IEC 10646 */
470. Identifier ::= Alpha{AlphaNum | Underscore}

```

```

471. Alpha ::= UpperAlpha | LowerAlpha
472. AlphaNum ::= Alpha | Num
473. 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"
474. 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"
475. 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 (0,0,0,33) 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)
*/
476. FreeText ::= "" {ExtendedAlphaNum} ""
477. AddressValue ::= "null"
478. OmitValue ::= OmitKeyword
479. OmitKeyword ::= "omit"

```

A.1.6.5 Parameterization

```

480. InParKeyword ::= "in"
481. OutParKeyword ::= "out"
482. InOutParKeyword ::= "inout"
483. FormalValuePar ::= [(InParKeyword | InOutParKeyword | OutParKeyword)] Type ValueParIdentifier
484. ValueParIdentifier ::= Identifier
485. FormalPortPar ::= [InOutParKeyword] PortTypeIdentifier PortParIdentifier
486. PortParIdentifier ::= Identifier
487. FormalTimerPar ::= [InOutParKeyword] TimerKeyword TimerParIdentifier
488. TimerParIdentifier ::= Identifier
489. FormalTemplatePar ::= [InParKeyword] TemplateKeyword Type TemplateParIdentifier
490. TemplateParIdentifier ::= Identifier

```

A.1.6.6 With statement

```

491. WithStatement ::= WithKeyword WithAttribList
492. WithKeyword ::= "with"
493. WithAttribList ::= "{" MultiWithAttrib "}"
494. MultiWithAttrib ::= {SingleWithAttrib [SemiColon]}+
495. SingleWithAttrib ::= AttribKeyword [OverrideKeyword] [AttribQualifier] AttribSpec
496. AttribKeyword ::= EncodeKeyword |
VariationKeyword |
DisplayKeyword |
ExtensionKeyword
497. EncodeKeyword ::= "encode"
498. VariationKeyword ::= "variant"
499. DisplayKeyword ::= "display"
500. ExtensionKeyword ::= "extension"
501. OverrideKeyword ::= "override"
502. AttribQualifier ::= "(" DefOrFieldRefList ")"
503. DefOrFieldRefList ::= DefOrFieldRef {"", " DefOrFieldRef}
504. DefOrFieldRef ::= DefinitionRef | FieldReference | AllRef | PredefinedType
/* 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 */
505. DefinitionRef ::= StructTypeIdentifier |
EnumTypeIdentifier |
PortTypeIdentifier |
ComponentTypeIdentifier |
SubTypeIdentifier |
ConstIdentifier |
TemplateIdentifier |
AltstepIdentifier |
TestcaseIdentifier |
FunctionIdentifier |
SignatureIdentifier |
FullGroupIdentifier
506. AllRef ::= AllGroupsWithException |
AllTypesWithException |
AllTemplsWithException |
AllConstsWithException |
AllAltstepsWithException |
AllTestcasesWithException |
AllFunctionsWithException |
AllSignaturesWithException |
AllParamWithException
507. AttribSpec ::= FreeText

```


A.1.6.7 Behaviour statements

```

508. BehaviourStatements ::= TestcaseInstance |
    FunctionInstance |
    ReturnStatement |
    AltConstruct |
    InterleavedConstruct |
    LabelStatement |
    GotoStatement |
    RepeatStatement DeactivateStatement |
    AltstepInstance/* 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 */
509. VerdictStatements ::= SetLocalVerdict
510. VerdictOps ::= GetLocalVerdict
511. 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 */
512. SetVerdictKeyword ::= "setverdict"
513. GetLocalVerdict ::= "getverdict"
514. SUTStatements ::= ActionKeyword "(" (FreeText | TemplateRefWithParList) ")"
515. ActionKeyword ::= "action"
516. ReturnStatement ::= ReturnKeyword [Expression]
517. AltConstruct ::= AltKeyword BeginChar AltGuardList EndChar
518. AltKeyword ::= "alt"
519. AltGuardList ::= {GuardStatement [SemiColon]}+ [ElseStatement [SemiColon]]
520. GuardStatement ::= AltGuardChar (AltstepInstance | GuardOp StatementBlock)
521. ElseStatement ::= "["ElseKeyword "]" StatementBlock
522. AltGuardChar ::= "[" [BooleanExpression] "]"
523. GuardOp ::= TimeoutStatement |
    ReceiveStatement |
    TriggerStatement |
    GetCallStatement |
    CatchStatement |
    CheckStatement |
    GetReplyStatement |
    DoneStatement
/* STATIC SEMANTICS - GuardOp used within the module control part. May only contain the
timeoutStatement */
524. InterleavedConstruct ::= InterleavedKeyword BeginChar InterleavedGuardList EndChar
525. InterleavedKeyword ::= "interleave"
526. InterleavedGuardList ::= {InterleavedGuardElement [SemiColon]}+
527. InterleavedGuardElement ::= InterleavedGuard InterleavedAction
528. InterleavedGuard ::= "[" "]" GuardOp
529. InterleavedAction ::= StatementBlock
/* STATIC SEMANTICS - the StatementBlock may not contain loop statements, goto, activate,
deactivate, stop, return or calls to functions */
530. LabelStatement ::= LabelKeyword LabelIdentifier
531. LabelKeyword ::= "label"
532. LabelIdentifier ::= Identifier
533. GotoStatement ::= GotoKeyword LabelIdentifier
534. GotoKeyword ::= "goto"
535. RepeatStatement ::= "repeat"
536. ActivateOp ::= ActivateKeyword "(" AltstepInstance ")"
537. ActivateKeyword ::= "activate"
538. DeactivateStatement ::= DeactivateKeyword ["(" Expression ")"]
/* STATIC SEMANTICS expression shall evaluate to a value of default type */
539. DeactivateKeyword ::= "deactivate"

```

A.1.6.8 Basic statements

```

540. BasicStatements ::= Assignment | LogStatement | LoopConstruct | ConditionalConstruct
541. Expression ::= SingleExpression | CompoundExpression
/* STATIC SEMANTICS - Expression shall not contain Configuration, activate operation or verdict
operations within the module control part */
542. CompoundExpression ::= FieldExpressionList | ArrayExpression
/* STATIC SEMANTICS - Within CompoundExpression the ArrayExpression can be used for Arrays, record,
record of and set of types. */
543. FieldExpressionList ::= "{" FieldExpressionSpec {"," FieldExpressionSpec} "}"
544. FieldExpressionSpec ::= FieldReference AssignmentChar Expression
545. ArrayExpression ::= "{" [ArrayElementExpressionList] "}"
546. ArrayElementExpressionList ::= NotUsedOrExpression {"," NotUsedOrExpression}
547. NotUsedOrExpression ::= Expression | NotUsedSymbol
548. ConstantExpression ::= SingleConstExpression | CompoundConstExpression
549. SingleConstExpression ::= SingleExpression

```

```

/* STATIC SEMANTICS - SingleConstExpression shall not contain Variables or Module parameters and
shall resolve to a constant Value at compile time */
550. BooleanExpression ::= SingleExpression
/* STATIC SEMANTICS - BooleanExpression shall resolve to a Value of type Boolean */
551. CompoundConstExpression ::= FieldConstExpressionList | ArrayConstExpression
/* STATIC SEMANTICS - Within CompoundConstExpression the ArrayConstExpression can be used for
Arrays, record, record of and set of types. */
552. FieldConstExpressionList ::= "{" FieldConstExpressionSpec {"," FieldConstExpressionSpec} "}"
553. FieldConstExpressionSpec ::= FieldReference AssignmentChar ConstantExpression
554. ArrayConstExpression ::= "{" [ArrayElementConstExpressionList] "}"
555. ArrayElementConstExpressionList ::= ConstantExpression {"," ConstantExpression}
556. Assignment ::= VariableRef AssignmentChar Expression
/* OPERATIONAL SEMANTICS - The Expression on the RHS of Assignment shall evaluate to an explicit
Value of the type of the LHS. */
557. SingleExpression ::= SimpleExpression {LogicalOp SimpleExpression}
/* OPERATIONAL SEMANTICS - If both SimpleExpressions and the LogicalOp exist then the
SimpleExpressions shall evaluate to specific values of compatible types */
558. SimpleExpression ::= [not] SubExpression
/* OPERATIONAL SEMANTICS - Operands of the not operator shall be of type boolean (TTCN or ASN.1) or
derivatives of type Boolean. */
559. SubExpression ::= PartialExpression [RelOp PartialExpression]
/* OPERATIONAL SEMANTICS - If both PartialExpressions and the RelOp exist then the
PartialExpressions shall evaluate to specific values of compatible types. */
/* OPERATIONAL SEMANTICS - If RelOp is "<" | ">" | ">=" | "<=" then each SubExpression shall
evaluate to a specific integer, Enumerated or float Value (these values can either be TTCN or ASN.1
values) */
560. PartialExpression ::= Result [ShiftOp Result]
/* OPERATIONAL SEMANTICS - Each Result shall resolve to a specific Value. If more than one Result
exists the right-hand operand shall be of type integer and if the shift op is '<<' or '>>' then the
left-hand operand shall resolve to either bitstring, hexstring or octetstring type. If the shift op
is '<@' or '@>' then the left-hand operand shall be of type bitstring, hexstring, charstring or
universal charstring */
561. Result ::= SubResult {BitOp SubResult}
/* OPERATIONAL SEMANTICS - If both SubResults and the BitOp exist then the SubResults shall evaluate
to specific values of compatible types */
562. SubResult ::= [not4b] Product
/* OPERATIONAL SEMANTICS - If the not4b operator exists, the operand shall be of type bitstring,
octetstring or hexstring. */
563. Product ::= Term {AddOp Term}
/* OPERATIONAL SEMANTICS - Each Term shall resolve to a specific Value. If more than one Term exists
then the Terms shall resolve to type integer or float. */
564. Term ::= Factor {MultiplyOp Factor}
/* OPERATIONAL SEMANTICS - Each Factor shall resolve to a specific Value. If more than one Factor
exists then the Factors shall resolve to type integer or float. */
565. Factor ::= [UnaryOp] Primary
/* OPERATIONAL SEMANTICS - The Primary shall resolve to a specific Value. If UnaryOp exists and is
"not" then Primary shall resolve to type BOOLEAN if the UnaryOp is "+" or "-" then Primary shall
resolve to type integer or float. If the UnaryOp resolves to not4b then the Primary shall resolve to
the type bitstring, hexstring or octetstring. */
566. Primary ::= OpCall | Value | "(" SingleExpression ")"
567. ExtendedFieldReference ::= {(Dot ( StructFieldIdentifier | ArrayOrBitRef |
TypeDefIdentifier))}+
/* OPERATIONAL SEMANTIC: The TypeDefIdentifier shall be used only if the type of the VarInstance or
ReferencedValue in wich the ExtendedFieldReference is used is anytype.
568. OpCall ::= ConfigurationOps |
VerdictOps |
TimerOps |
TestcaseInstance |
FunctionInstance |
TemplateOps |
ActivateOp
569. AddOp ::= "+" | "-" | StringOp
/* OPERATIONAL SEMANTICS - Operands of the "+" or "-" operators shall be of type integer or
float(i.e., TTCN or ASN.1 predefined) or derivations of integer or float (i.e., subrange) */
570. MultiplyOp ::= "*" | "/" | "mod" | "rem"
/* OPERATIONAL SEMANTICS - Operands of the "*", "/", rem or mod operators shall be of type integer
or float(i.e., TTCN or ASN.1 predefined) or derivations of integer or float (i.e., subrange). */
571. UnaryOp ::= "+" | "-"
/* OPERATIONAL SEMANTICS - Operands of the "+" or "-" operators shall be of type integer or
float(i.e., TTCN or ASN.1 predefined) or derivations of integer or float (i.e., subrange). */
572. RelOp ::= "==" | "<" | ">" | "!=" | ">=" | "<="
/* OPERATIONAL SEMANTICS - the precedence of the operators is defined in Table 7
*/
573. BitOp ::= "and4b" | "xor4b" | "or4b"
/* OPERATIONAL SEMANTICS - Operands of the and4b, or4b or xor4b operator shall be of type bitstring,
hexstring or octetstring (TTCN or ASN.1) or derivatives of these types. */
574. LogicalOp ::= "and" | "xor" | "or"

```

```

/* OPERATIONAL SEMANTICS - Operands of the and, or or xor operators shall be of type boolean (TTCN
or ASN.1) or derivatives of type Boolean. */
/* OPERATIONAL SEMANTICS - the precedence of the operators is defined in Table 7 */
575. StringOp ::= "&"
/* OPERATIONAL SEMANTICS - Operands of the string operator shall be bitstring, hexstring,
octetstring or character string */
576. ShiftOp ::= "<<" | ">>" | "<@" | "@>"
577. LogStatement ::= LogKeyword "(" [FreeText] ")"
578. LogKeyword ::= "log"
579. LoopConstruct ::= ForStatement |
                    WhileStatement |
                    DoWhileStatement
580. ForStatement ::= ForKeyword "(" Initial SemiColon Final SemiColon Step ")"
                    StatementBlock
581. ForKeyword ::= for
582. Initial ::= VarInstance | Assignment
583. Final ::= BooleanExpression
584. Step ::= Assignment
585. WhileStatement ::= WhileKeyword "(" BooleanExpression ")"
                    StatementBlock
586. WhileKeyword ::= "while"
587. DoWhileStatement ::= DoKeyword StatementBlock
                    WhileKeyword "(" BooleanExpression ")"
588. DoKeyword ::= "do"
589. ConditionalConstruct ::= IfKeyword "(" BooleanExpression ")"
                    StatementBlock
                    {ElseIfClause}[ElseClause]
590. IfKeyword ::= if
591. ElseIfClause ::= ElseKeyword IfKeyword "(" BooleanExpression ")" StatementBlock
592. ElseKeyword ::= "else"
593. ElseClause ::= ElseKeyword StatementBlock

```

A.1.6.9 Miscellaneous productions

```

594. Dot ::= "."
595. Dash ::= "-"
596. Minus ::= Dash
597. SemiColon ::= ";"
598. Colon ::= ":"
599. Underscore ::= "_"
600. BeginChar ::= "{"
601. EndChar ::= "}"
602. AssignmentChar ::= "!="

```

Annex B (normative): 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.1.2 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 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.1.2.4 Any value

The matching symbol "?" (*AnyValue*) is used to indicate that any valid incoming value is acceptable. It can be used on values of all types. A template field that uses the any value mechanism matches the corresponding incoming field if, and only if, the incoming field evaluates to a single element of the specified type.

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.1.2.5 Any value or none

The matching symbol "*" (*AnyValueOrNone*) is used to indicate that any valid incoming value, including omission of that value, is acceptable. It can be used on values of all types, provided that the template field is declared as optional.

A template field that uses this symbol matches the corresponding incoming field if, and only if, either the incoming field evaluates to any element of the specified type, or if the incoming field is absent.

EXAMPLE:

```
template Mymessage MyTemplate:=
{
    :
    field3 := *,      // will match true or false or omitted field
    :
}
```

B.1.2.6 Value range

Ranges indicate a bounded range of acceptable values. It shall be used only on values of **integer** types (and integer sub-types). A boundary value shall be either:

- a) infinity or -infinity;
- b) an expression that evaluates to a specific integer value.

The lower boundary shall be put on the left side of the range, the upper boundary at the right side. The lower boundary shall be less than the upper boundary. A template field that uses a range matches the corresponding incoming field if, and only if, the incoming field value is equal to one of the values in the range.

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.1.2.7 SuperSet

SuperSet is an operation for matching that shall be used only on values of **record of** and **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

type record of integer MyRecordOfType;

template MyRecordOfType MyTemplate2 := superset ( 1, 2, 3 );
// any sequence of integers matches which contains the numbers 1, 2 and 3 (in this sequential
// order) at the first three positions (but may contain more elements)
```

B.1.2.8 SubSet

SubSet is an operation for matching that can be used only on values of **record of** and **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

template MyRecordOfType MyTemplate2 := subset ( 1, 2, 3 );
// any sequence of integers matches which contains at most three elements and their values
// are 1 for the first element (if present), 2 for the second element (if present) and
// 3 for the third element (if present)
```

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 clause B.1.5). For example: "abcdxyz", "abccxyz" "abcxyz" 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. It shall be used only within values of string types or arrays. The "*" symbol matches the longest sequence of elements possible, according to the pattern as specified by the symbols surrounding the "*".

EXAMPLE:

```
template Mymessage MyTemplate:=
{
  :
  field2 := "abcxyz",
  field3 := '10*11'B,      // where "*" may be any sequence of bits (possibly empty)
  field4 := {*, 2, 3}      // where the first element may be any integer value or omitted
}

var charstring MyStrings[4];
MyPCO.receive(MyStrings:{"abyz", *, "abc" });
```

If a "*" appears at the highest level inside a string, a **record of**, **set of** or array, it shall be interpreted as *AnyElementsOrNone*.

NOTE: This rule prevents the otherwise possible interpretation of "*" as *AnyValueOrNone* that replaces an element inside a string, **record of**, **set of** or array.

B.1.3.2.1 Using multiple character wildcards

If it is required to expressed the "*" wildcard in character strings it shall be done using character patterns (see clause B.1.5).For example: "abcxyz", "abcdefxyz" "abcabcxyz" etc. will all match **pattern** "abc*xyz".

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** or **record of** structure. It shall be used only as an attribute of the following mechanisms: Complement, *AnyValue*, *AnyValueOrNone*, *AnyElement* and *AnyElementsOrNone*. It can also be used in conjunction with the **ifpresent** attribute. The syntax for **length** can be found in clauses 6.2.3 and 6.3.3.

The units of length are to be interpreted according to table 4 in the main body of the present document in the case of string values. For **set of** and **record of** types the unit of length is the replicated type. The boundaries shall be denoted by expressions which resolve to specific non-negative **integer** values. Alternatively, the keyword **infinity** can be used as a value for the upper boundary in order to indicate that there is no upper limit of length.

The length specifications for the template shall not conflict with the length for restrictions (if any) of the corresponding type. A template field that uses Length as an attribute of a symbol matches the corresponding incoming field if, and only if, the incoming field matches both the symbol and its associated attribute. The length attribute matches if the length of the incoming field is greater than or equal to the specified lower bound and less than or equal to the upper bound. In the case of a single length value the length attribute matches only if the length of the received field is exactly the specified value.

In the case of an omitted field, the length attribute is always considered as matching (i.e. with **omit** it is redundant). With *AnyValueOrNone* and **ifpresent** it places a restriction on the incoming value, if any.

EXAMPLE:

```
template Mymessage MyTemplate:=
{
  field1 := complement (4,5) length (1 .. 6), // is the same as (1,2,3,6)
  field2 := "ab*ab" length(13) // max length of the AnyElementsOrNone string is 9 characters
  :
}
```

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 ? and * to mean any character and any number of any character respectively.

EXAMPLE 1:

```
template charstring MyTemplate:= pattern "ab??xyz*";
```

This template would match any character string that consists of the characters 'ab', followed by any two characters, followed by the characters 'xyz', followed by any number of any characters.

If it is required to interpret any metacharacter literally it should be preceded with the 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 meta characters for TTCN-3 patterns is shown in table B.1.

Table B.1: List of TTCN-3 pattern metacharacters

Metacharacter	Description
?	Match any character
*	Match any character zero or more times
\	Cause the following meta character to be interpreted as a literal (see note)
[]	Match any character within the specified set, see clause B.1.5.1 for more details
-	Usable inside a pair of square brackets ("[" and "]") only and allows to specify a range of characters; see clause B.1.5.1 for more details
^	Usable inside a pair of square brackets ("[" and "]") only and cause to match any character complementing the set of characters following this metacharacter; see clause 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 clause B.1.5.2 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 (see ISO/IEC 6429 [13])
\n	Match the C0 control character LF (see ISO/IEC 6429 [13])
\r	Match the C0 control character CR (see ISO/IEC 6429 [13])
\"	Match the double quote character
	Used to denote two alternative expressions
()	Used to group an expression
#(n, m)	Match the preceding expression at least n times but no more than m times. See clause B.1.5.3 for more details
NOTE:	Consequently the backslash character can be matched by a pair of backslash characters without space between them (\\).

B.1.5.1 Set expression

The set expression is delimited by the '[' ']' symbols. In addition to character literals, it is possible to specify character ranges using the separator '-'. The set expression can also be negated by placing the '^' character as the first character after the opening square brace.

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 "[A-E][0-9][0-9][0-9]YKE";

// RegExp3 will match a string which starts with a letter between A and E 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 statement to use references to existing templates, constants or variables. The reference is enclosed within the '{ '}' characters. The reference shall resolve to one of the character string types.

EXAMPLE:

```
const charstring MyString:= "ab?";
template charstring MyTemplate:= pattern "{MyString}";
```

This template would match any character string that consists of the characters 'ab', followed by any characters. In effect any character string following the **pattern** keyword either explicitly or by reference will be interpreted following the rules defined in this clause.

```
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 characters, followed by the characters 'de', followed by the character in ISO10646-1 with group=1, plane=1, row=65 and cell=7.

B.1.5.3 Match expression n times

To specify that the preceding expression should be matched a number of times the '#(n, m)' syntax is used. This specifies that the preceding expression must be matched at least n times but not more than m times.

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 RegExp5:= 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
```

Annex C (normative): Pre-defined TTCN-3 functions

This annex defines the TTCN-3 predefined functions.

C.1 Integer to character

```
int2char(integer value) return char
```

This function converts an **integer** value in the range of 0 ... 127 (8-bit encoding) into a character value of ISO/IEC 646 [5]. The integer value describes the 8-bit encoding of the character.

The function returns -1 if the value of the argument is a negative or greater than 127.

C.2 Character to integer

```
char2int(char value) return integer
```

This function converts a **char** value of ISO/IEC 646 [5] into an integer value in the range of 0 ... 127. The integer value describes the 8-bit encoding of the character.

C.3 Integer to universal character

```
int2unichar(integer value) return universal char
```

This function converts an **integer** value in the range of 0 ... 2 147 483 647 (32-bit encoding) into a character value of ISO/IEC 10646 [6]. The integer value describes the 32-bit encoding of the character.

The function returns the quadruple (255, 255, 255, 255) if the value of the argument is a negative or greater than

2 147 483 647

C.4 Universal character to integer

```
unichar2int(universal char value) return integer
```

This function converts a **universal char** value of ISO/IEC 10646 [6] into an integer value in the range of 0 ... 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 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 .. F represent the decimal values 0 .. 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 least significant, the leftmost hexadecimal digit is the most significant. The number of hexadecimal digits provided shall be multiples of 2 since one octet is composed of two hexadecimal digits. The hexadecimal digits 0 .. F represent the decimal values 0 .. 15 respectively.

C.8 Charstring to integer

```
str2int(charstring value) return integer
```

This function converts a **charstring** representing an **integer** value to the equivalent **integer**. If the string does not represent a valid integer value the function returns the value zero (0).

EXAMPLES:

```
str2int("66") // will return the integer value 66
str2int("-66") // will return the integer value -66
str2int("abc") // will return the integer value 0
str2int("0") // will return the integer value 0
```

C.9 Integer to bitstring

```
int2bit(integer value, length) return bitstring
```

This function converts a single **integer** value to a single **bitstring** value. The resulting string is **length** bits long.

For the purposes of this conversion, a **bitstring** shall be interpreted as a positive base 2 **integer** value. The rightmost bit is least significant, the leftmost bit is the most significant. The bits 0 and 1 represent the decimal values 0 and 1 respectively. If the conversion yields a value with fewer bits than specified in the **length** parameter, then the **bitstring** shall be padded on the left with zeros. A test case error shall occur if the **value** is negative or if the resulting **bitstring** contains more bits than specified in the **length** parameter.

C.10 Integer to hexstring

```
int2hex(integer value, length) return hexstring
```

This function converts a single **integer** value to a single **hexstring** value. The resulting string is **length** hexadecimal digits long.

For the purposes of this conversion, a **hexstring** shall be interpreted as a positive base 16 **integer** value. The rightmost hexadecimal digit is least significant, the leftmost hexadecimal digit is the most significant. The hexadecimal digits 0 ... F represent the decimal values 0 ... 15 respectively. If the conversion yields a value with fewer hexadecimal digits than specified in the `length` parameter, then the **hexstring** shall be padded on the left with zeros. A test case error shall occur if the `value` is negative or if the resulting **hexstring** contains more hexadecimal digits than specified in the `length` parameter.

C.11 Integer to octetstring

```
int2oct(integer value, length) return octetstring
```

This function converts a single **integer** value to a single **octetstring** value. The resulting string is `length` octets long.

For the purposes of this conversion, an **octetstring** shall be interpreted as a positive base 16 **integer** value. The rightmost hexadecimal digit is least significant, the leftmost hexadecimal digit is the most significant. The number of hexadecimal digits provided shall be multiples of 2 since one octet is composed of two hexadecimal digits. The hexadecimal digits 0 .. F represent the decimal values 0 .. 15 respectively. If the conversion yields a value with fewer hexadecimal digits than specified in the `length` parameter, then the **hexstring** shall be padded on the left with zeros. A test case error shall occur if the `value` is negative or if the resulting **hexstring** contains more hexadecimal digits than specified in the `length` parameter.

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).

EXAMPLES:

```
int2str(66)      // will return the charstring value "66"
int2str(-66)    // will return the charstring value "-66"
int2str(0)      // will return the integer value "0"
```

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 in the main body of the present document.

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 [6], 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 type

sizeof(structured_type value) **return integer**

This function returns the declared number of elements of a **record**, **record of**, **set**, **set of** type or the actual number of elements of a constant, variable, **template** of these types or array (see note). This function shall not be applied to **record of** or **set of** types without length subtyping. In the case of **record of** and **set of** values or 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
  };
type record of integer MyPDU1;

template MyPDU MyTemplate
  { field1 omit,
    field2 5
  };

var integer numElements;

// then
numElements := sizeof(MyPDU); // returns 2
numElements := sizeof(MyTemplate); // returns 1
numElements := sizeof(MyPDU1); // returns error as MyPDU1 is not constrained

// Given
type record length(0..10) of integer MyRecord;
var MyRecord 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 returns the value **true** if and only if the value of the referenced field is present in the actual instance of the referenced data object. The argument to **ispresent** shall be a reference to a field within a data object that is defined as being **optional**.

```
// Given
type record MyRecord
  { boolean field1 optional,
    integer field2
  }
// and given that MyPDU is a template of MyRecord type
// and received_PDU is also of MyRecord type
// then
MyPort.receive(MyPDU) -> value received_PDU
ispresent(received_PDU.field1)
// returns true if field1 in the actual instance of MyPDU is present
```

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 clause 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 value) return charstring`

This function converts a **octetstring** value to an **charstring**. The resulting **charstring** will have the same length as the incoming **octetstring**. The octets are interpreted as ISO/IEC 646 [5] codes (according to the IRV) and the resulting characters are stored in the returned value. Octet values higher than 7F shall cause an error.

EXAMPLE:

`oct2str ('4469707379'H) = "Dipsy"`

NOTE: The character string returned may contain non-graphical characters, which can not be presented between the double quotes.

C.21 Character string to octetstring

`str2oct (charstring value) return octetstring`

This function converts a **charstring** value to an **octetstring**. The resulting **octetstring** will have the same length as the incoming **charstring**. Each octet of the **octetstring** will contain the ISO/IEC 646 [5] codes (according to the IRV) of the appropriate characters of the **charstring**.

EXAMPLE:

`str2oct ("Tinky-Winky") = '54696E6B792D57696E6B79'H`

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 should 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 not 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** should 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: $\text{bit2oct}(\text{value}) = \text{hex2oct}(\text{bit2hex}(\text{value}))$.

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** should 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** should 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) = '000111010111'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 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 integers 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, **integer** index, 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"
```

Annex D (normative): Using other data types with TTCN-3

D.1 Using ASN.1 with TTCN-3

This annex defines the optional use of ASN.1 with TTCN-3.

D.1.0 General

TTCN-3 provides a clean interface for using ASN.1 version 1997 (as defined in the ITU-T Recommendation X.680 series [7], [8], [9], [10]) in TTCN-3 modules. When imported into a TTCN-3 module the language identifier shall be:

"ASN.1:1997" for ASN.1 version 1997,
 "ASN.1:1994" for ASN.1 version 1994,
 "ASN.1:1988" for Blue Book version of ASN.1

NOTE 1: Language identifiers "ASN.1:1994" and "ASN.1:1988" refer to versions of ASN.1 based on superseded ITU-T Recommendations and the only purpose to include them into this standard is to allocate unique identifiers if protocol modules based on these ASN.1 versions are used with TTCN-3.

NOTE 2: When "ASN.1:1988" is supported, the ASN.1 items shall be imported according to the syntactical and semantical rules of ITU-T Recommendation X.208 (Blue Book)

NOTE 3: References for ASN.1:1994 and ASN.1:1988 can be found in annex F.

When ASN.1 is used with TTCN-3 the keywords listed in clause 11.18 of ITU-T Recommendation X.680 [7] shall not be used as identifiers in a TTCN-3 module. ASN.1 keywords shall follow the requirements of ITU-T Recommendation X.680 [7].

D.1.1 ASN.1 and TTCN-3 type equivalents

D.1.1.0 General

The ASN.1 types listed in table D.1 are considered to be equivalent to their TTCN-3 counterparts.

Table D.1: List of ASN.1 and TTCN-3 equivalents

ASN.1 type	Maps to TTCN-3 equivalent
BOOLEAN	boolean
INTEGER	integer
REAL (note1)	float
OBJECT IDENTIFIER	objid
BIT STRING	bitstring
OCTET STRING	octetstring
SEQUENCE	record
SEQUENCE OF	record of
SET	set
SET OF	set of
ENUMERATED	enumerated
CHOICE	union
VisibleString	char(note2), charstring
IA5String	char(note2), charstring
UniversalString	universal char(note2), universal charstring

NOTE 1: The ASN.1 type REAL is equivalent to the TTCN-3 type **float** until the base is unrestricted or restricted to base 10 explicitly or implicitly. The ASN.1 notation allows explicit restriction by e.g. inner subtyping but from ASN.1-TTCN-3 type mapping point of view an explicit restriction is an ASN.1 value notation. Implicit restriction may be defined by the textual description of the given protocol, i.e. outside of the ASN.1 module(s). However, in both cases the TTCN-3 value notation can be used irrespective if the base in ASN.1 (see also note 3 in clause D.1.2.0).

NOTE 2: Only ASN.1 subtypes of the length of exactly 1 character are equivalent to TTCN-3 basic char types, e.g. IA5String (SIZE (1)) is equivalent to the TTCN-3 type char but IA5String (SIZE (0..1)) is not.

All TTCN-3 operators, functions, matching mechanisms, value notation etc. that can be used with a TTCN-3 type given in table D.1 may also be used with the corresponding ASN.1 type.

D.1.1.1 Identifiers

In converting ASN.1 identifiers to TTCN-3 identifiers any hyphen "-" symbols shall be changed to an underscore "_".

EXAMPLE:

```
MyASN1module DEFINITIONS ::=
BEGIN
    Missleading-ASN1-Name ::=    INTEGER    -- ASN.1 type identifier using "-"

END

module MyTTCNModule
{
    import from MyASN1module language "ASN.1:1997" all;

    const Missleading_ASN1_Name ExampleConst := 1;    // TTCN-3 reference to ASN.1 type
                                                       // using underscores
}

```

D.1.2 ASN.1 data types and values

D.1.2.0 General

ASN.1 types and values may be used in TTCN-3 modules. ASN.1 definitions are made using a separate ASN.1 module. ASN.1 types and values are referenced by their type references and value references as produced according to clauses 9.3 and 9.4 of ITU-T Recommendation X.680 [7] within the ASN.1 module(s).

EXAMPLE 1:

```
MyASN1module DEFINITIONS ::=
BEGIN
    Z ::=    INTEGER    -- Simple type definition

    BMessage ::= SET    -- ASN.1 type definition
    {
        name    Name,
        title   VisibleString,
        date    Date
    }

    johnValues Bmessage ::=    -- ASN.1 value definition
    {
        name    "John Doe",
        title   "Mr",
        date    "April 12th"
    }

    DefinedValuesForField1 Z ::= {0 | 1} -ASN.1 subtype definition
END

```

The ASN.1 module shall conform to the syntax of the ITU-T Recommendation X.680 series [7], [8], [9] and [10]. Once declared, ASN.1 types and values may be used within TTCN-3 modules in a similar way that ordinary TTCN-3 types and values from other TTCN-3 modules are used (i.e. the required definitions shall be imported). When importing ASN.1 items into a TTCN-3 module, an associated type or value is produced for each ASN.1 item imported. All TTCN-3 definitions or assignments based on imported ASN.1 items shall be done according the rules imposed by the related associated type or value. Also, the matching mechanism shall use the associated type when matching constants, variables, templates or in-line expressions based on ASN.1 declarations.

Associated types and values are derived from ASN.1 items by application the following transformation rules (the order corresponds to the order of execution of the individual transformations):

- 1) Ignore any extension markers and exception specifications.
- 2) Ignore any user defined constraints (see clause 9 of ITU-T Recommendation X.682 [9]).
- 3) Ignore any contents constraint (see clause 9 of ITU-T Recommendation X.682 [9]).
- 4) Ignore any pattern constraint (see clause 48.9 of ITU-T Recommendation X.680 [7]).
- 5) Create equivalent subtypes from all types constrained by contained subtyping by replacing included types by the set of values they represent.
- 6) Execute the COMPONENTS OF transformation according to clause 24.4 of ITU-T Recommendation X.680 [7] on any SEQUENCE types and according to clause and 26.2 on any SET types containing the keywords " COMPONENTS OF".
- 7) Replace any EMBEDDED PDV type with its associated type obtained by expanding inner subtyping in the associated type of the EMBEDDED PDV type (see clause 32.5 of ITU-T Recommendation X.680 [7]) to a full type definition.
- 8) Replace the EXTERNAL type with its associated type obtained by expanding inner subtyping in the associated type of the EXTERNAL type (see clause 33.5 of ITU-T Recommendation X.680 [7]) to a full type definition (see note 3).
- 9) Replace the CHARACTER STRING type with its associated type obtained by expanding inner subtyping in the associated type of the CHARACTER STRING type (see clause 39.5 of ITU-T Recommendation X.680 [7]) to a full type definition..
- 10) Replace the INSTANCE OF type with its associated type obtained by substituting INSTANCE OF DefinedObjectClass by its associated ASN.1 type (see clause C.7 of ITU-T Recommendation X.681 [8]) and replace all ASN.1 types with their TTCN-3 equivalents according to table D.1. The resulted type is the TTCN-3 associated type.
- 11) Ignore any remaining inner subtyping (see note 4).
- 12) Ignore any named numbers and named bits in ASN.1 types. In ASN.1 values replace any named number by its value and substitute any named bits or sequence of named bits by a bitstring without trailing zeros, where bit positions identified by names present are replaced by "1"s, other bit positions are replaced by "0"s.
- 13) Replace any selection type with the type referenced by the selection type; if the denoted choice type (the "Type" in clause 29.1 of ITU-T Recommendation X.680 [7]) is a constrained type, the selection has to be done on the parent type of the denoted choice type.
- 14) Convert any RELATIVE-OID type or value to an **objid** type or value (see note 5).
- 15) Replace any of the following restricted character string types with their associated types obtained as (see note 6):
 BMPString: **universal charstring** (char (0,0,0,0) .. char (0,0,255,255));
 UTF8String: **universal charstring**;
 NumericString: **charstring** constrained to the set of characters as given in clause 36.2 of ITU-T Recommendation X.680 [7];
 PrintableString: **charstring** constrained to the set of characters as given in clause 36.4 of ITU-T Recommendation X.680 [7];
 TeletexString and T61String: **universal charstring** constrained to the set of characters as given in ITU-T Recommendation T.61 (see bibliography);

VideotexString: **universal charstring** constrained to the set of characters as given in ITU-T Recommendations T.100 [14] and T.101 [15];
 GraphicString: **universal charstring**;
 GeneralString: **universal charstring**.

- 16) Replace any GeneralizedTime and UTCTime types or values with the type or value of **charstring**.
- 17) Replace any ObjectDescriptor type or value by the **universal charstring** type or value.
- 18) Replace any notations for the object class field types (see clause 14 of ITU-T Recommendation X.681 [8]) by the ASN.1 item they are referencing to; open types has to be replaced by the metatype "OPEN TYPE" for the purpose of the transformation (and only for that).
- 19) Replace all information from objects notations (see clause 15 of ITU-T Recommendation X.681 [8]) by the ASN.1 item they are referencing to.
- 20) Revert table constraints (see clause 10 of ITU-T Recommendation X.682 [9]) to list subtyping and ignore all relational constraints (see note 7).
- 21) Replace all occurrences of NULL type with the following associated TTCN-3 type:
type enumerated <identifier> { NULL }, where <identifier> is the ASN.1 Type reference converted according to clause D.1.1.1.
- 22) Replace all references to open types with **anytype**.
- 23) Replace ASN.1 types with their equivalents according to table D.1 and ASN.1 values with equivalent TTCN-3 values based on the associated types (see note 8). The metatype "OPEN TYPE" has to be replaced by **anytype**.

NOTE 1: Associated types alone do not allow the correct encoding values based on ASN.1 types. However, the extra information needed for the system to make correct encoding is implementation dependent and remains hidden for the user and not needed to make proper declarations or assignments based on ASN.1 types and values.

NOTE 2: When importing ENUMERATED types, integer numbers assigned by the user to enumerations are also imported.

NOTE 3: The data-value field of the EXTERNAL type may be encoded as a single-ASN1-type, octet-aligned or arbitrary (see clause 8.18.1 of ITU-T Recommendation X.690 [11]) at the discretion of the encoder; if the user wants to enforce one given form of encoding or wants to allow only one specific encoding form at matching, it shall use the appropriate encoding attribute for the type or the given constant, variable, template or template field (see clause D.1.5.2).

NOTE 4: Inner subtyping shall be taken into account by the user when defining TTCN-3 values or templates based on an ASN.1 type constrained by inner subtyping.

NOTE 5: Equivalence with the **objid** type is limited to the syntax to be used for value notations only. In the case of objid values the first two values of the tree are restricted (see ITU-T Recommendation X.660 [16]). This restriction does not apply to values based on the imported RELATIVE-OID type.

NOTE 6: VisibleString, IA5String and UniversalString has their equivalent TTCN-3 types and replaced directly.

NOTE 7: Relational constraints shall be taken into account by the user when declaring values and templates (also may be handled by tools implicitly).

NOTE 8: Missing optional fields in values of structured ASN.1 types (SET, SEQUENCE, EXTERNAL etc.) are equivalent to explicitly omitted fields in structured TTCN-3 values.

EXAMPLE 2:

```

module MyTTCNModule
{
    import from MyASN1module language "ASN.1:1997" all;

    const Bmessage MyTTCNConst:= johnValues;
    const DefinedValuesForField1 Value1:= 1;
}

```

NOTE 9: ASN.1 definitions other than types and values (i.e. information object classes or information object sets) are not directly accessible from the TTCN-3 notation. Such definitions shall be resolved to a type or value within the ASN.1 module before they can be referenced from within the TTCN-3 module.

D.1.2.1 Scope of ASN.1 identifiers

Imported ASN.1 identifiers follow the same scope rules as imported TTCN-3 types and values (see clause 5.3).

D.1.3 Parameterization in ASN.1

It is permitted to reference parameterized ASN.1 type and value definitions from within the TTCN-3 module. However, all ASN.1 parameterized definitions used in a TTCN-3 module shall be provided with actual parameters (open types are not permitted) and the actual parameters provided shall be resolvable at compile-time.

The TTCN-3 core language does not support the import of ASN.1 items, which employ uniquely ASN.1 specific objects as formal or actual parameter(s). ASN.1 specific parameterization which involves objects which cannot be defined directly in the TTCN-3 core language shall therefore be resolved in the ASN.1 part before use within the TTCN-3. The ASN.1 specific objects are:

- a) Information Object classes;
- b) Information Objects;
- c) Information Object Sets.

For example the following is not legal because it defines a TTCN-3 type which takes an ASN.1 object set as an actual parameter.

```
MyASN1module DEFINITIONS ::=
BEGIN
  -- ASN.1 Module definition

  -- Information object class definition
  MESSAGE ::= CLASS { &msgTypeValue  INTEGER UNIQUE,
                    &MsgFields}

  -- Information object definition
  setupMessage MESSAGE ::= { &msgTypeValue  1,
                    &MsgFields  OCTET STRING}

  setupAckMessage MESSAGE ::= { &msgTypeValue  2,
                    &MsgFields  BOOLEAN}

  -- Information object set definition
  MyProtocol MESSAGE ::= { setupMessage | setupAckMessage}

  -- ASN.1 type constrained by object set
  MyMessage{ MESSAGE : MsgSet} ::= SEQUENCE
  {
    code  MESSAGE.&msgTypeValue({ MsgSet}),
    Type  MESSAGE.&MsgFields({ MsgSet})
  }
END

module MyTTCNModule
{
  // TTCN-3 module definition
  import from MyASN1module language "ASN.1:1997" all;

  // Illegal TTCN-3 type with object set as parameter
  type record Q(MESSAGE MyMsgSet) ::= { Z field1,
                    MyMessage(MyMsgSet) field2}
}
```

To make this a legal definition the extra ASN.1 type My Message1 has to be defined as shown below. This resolves the information object set parameterization and can therefore be directly used in the TTCN-3 module.


```

MyASN1module DEFINITIONS ::=
BEGIN
  -- ASN.1 Module definition
  ...

  MyProtocol MESSAGE ::= { setupMessage | setupAckMessage}

  -- Extra ASN.1 type to remove object set parameterization
  MyMessage1 ::= MyMessage{ MyProtocol}
END

module MyTTCNModule
{
  // TTCN-3 module definition
  import from MyASN1module language "ASN.1:1997" all;

  // Legal TTCN-3 type with no object set as parameter
  type record Q := { Z field1,
                    MyMessage1 field2}
}

```

D.1.4 Defining ASN.1 message templates

D.1.4.0 General

If messages are defined in ASN.1 using, for example, SEQUENCE (or possibly SET) then actual messages, for both **send** and **receive** events, can be specified using the ASN.1 value syntax.

EXAMPLE:

```

MyASN1module DEFINITIONS ::=
BEGIN
  -- ASN.1 Module definition

  -- The message definition
  MyMessageType ::= SEQUENCE
  { field1 [1] IA5STRING, // Like TTCN-3 character string
    field2 [2] INTEGER OPTIONAL, // like TTCN-3 integer
    field3 [4] Field3Type, // Like TTCN-3 record
    field4 [5] Field4Type // Like TTCN-3 array
  }

  Field3Type ::= SEQUENCE {field31 BIT STRING, field32 INTEGER, field33 OCTET STRING},
  Field4Type ::= SEQUENCE OF BOOLEAN

  -- may have the following value
  myValue MyMessageType ::=
  {
    field1 "A string",
    field2 123,
    field3 {field31 '11011'B, field32 456789, field33 'FF'O},
    field4 {true, false}
  }
END

```

D.1.4.1 ASN.1 receive messages using the TTCN-3 template syntax

Matching mechanisms are not supported in the standard ASN.1 syntax. Thus, if it is wished to use matching mechanisms with an ASN.1 receive message then the TTCN-3 syntax for receive templates shall be used instead. Note that this syntax includes component references in order to be able to reference the individual components in ASN.1 SEQUENCE, SET etc.

EXAMPLE:

```

import from MyASN1module language "ASN.1:1997" {
    type myMessageType
}

// a message template using matching mechanisms within TTCN-3 might be
template myMessageType MyValue:=
{
    field1 :=          "A"<?>"tr"<*>"g",
    field2 :=          *,
    field3.field31 :=  '110??'B,
    field3.field32 :=  ?,
    field3.field33 :=  'F?'O,
    field4.[0] :=      true,
    field4.[1] :=      false
}

// the following syntax is equally valid
template myMessageType MyValue:=
{
    field1 := "A"<?>"tr"<*>"g",           // string with wildcards
    field2 := *,                          // any integer or none at all
    field3 := {'110??'B, ?, 'F?'O},
    field4 := {?, false}
}

```

D.1.4.2 Ordering of template fields

When TTCN-3 templates are used for ASN.1 types the significance of the order of the fields in the template will depend on the type of ASN.1 construct used to define the message type. For example: if SEQUENCE or SEQUENCE OF is used then the message fields shall be sent or matched in the order specified in the template. If SET or SET OF is used then the message fields may be sent or matched in any order.

D.1.5 Encoding information

D.1.5.0 General

TTCN-3 allows references to encoding rules and variations within encoding rules to be associated with various TTCN-3 language elements. It is also possible to define invalid encodings. This encoding information is specified using the **with** statement according to the following syntax:

EXAMPLE:

```

module MyModule
{
    :
    import from MyASN1module language "ASN.1:1997" {
        type myMessageType
    }
    with {
        encode:= "PER-BASIC-ALIGNED:1997" // All instances of MyMessageType should be encoded
using PER:1997
    }
    :
} // end module
with { encode "BER:1997" } // Default encoding for the entire module (test suite) is BER:1997

```

D.1.5.1 ASN.1 encoding attributes

The following strings are the predefined (standardized) encoding attributes for ASN.1:

- a) "BER:1997" means encoded according to ITU-T Recommendation X.690 (BER) [11];
- b) "CER:1997" means encoded according to ITU-T Recommendation X.690 (CER) [11];
- c) "DER:1997" means encoded according to ITU-T Recommendation X.690 (DER) [11].

- d) "PER-BASIC-UNALIGNED:1997" means encoded according to (Unaligned PER) ITU-T Recommendation X.691 [12];
- e) "PER-BASIC-ALIGNED:1997" means encoded according to ITU-T Recommendation X.691 (Aligned PER) [12];
- f) "PER-CANONICAL-UNALIGNED:1997" means encoded according to (Canonical Unaligned PER) ITU-T Recommendation X.691 [12];
- g) "PER-CANONICAL-ALIGNED:1997" means encoded according to ITU-T Recommendation X.691 (Canonical Aligned PER) [12].

D.1.5.2 ASN.1 variant attributes

The following strings are predefined (standardized) variant attributes. They have predefined meaning when applied jointly with predefined ASN.1 encoding attributes only (see clause D.1.5.1). Handling of these predefined attributes when applied jointly with other attributes or to an TTCN-3 object without an attribute is out of scope of this standard (see note 1):

- a) "length form 1" means, that the given value shall only be encoded and decoded using the short form of the length octets (see clause 8.1.3 of ITU-T Recommendation X.690 [11]) in case of BER, CER and DER encodings or the single octet length determinant (see clause 10.9 of Recommendation X.691 [12]) in case of any form of the PER encoding (see note 2).
- b) "length form 2" means, that the given value shall only be encoded and decoded using the long form of the length octets (see clause 8.1.3 of ITU-T Recommendation X.690 [11]) in case of BER, CER and DER encodings or the two octets length determinant (see clause 10.9 of ITU-T Recommendation X.691 [12]) in case of any form of the PER encoding (see note 2).
- c) "length form 3" means, that the given value shall only be encoded and decoded using the indefinite form of the length octets (see clause 8.1.3 of ITU-T Recommendation X.690 [11]) in case of BER, CER and DER encodings.
- d) "REAL base 2" means that the given value shall be encoded or matched according to the REAL binary encoding form. This attribute can be used on constants, variables or templates only and when used on any kind of a grouping (e.g. to groups or to the whole import statement) it shall have effect on these TTCN-3 objects only.
- e) "single-ASN1-type", "octet-aligned" and "arbitrary" means, that the given value based on an ASN.1 EXTERNAL type shall be encoded using the specified by the attribute encoding form or matched if received with the specified choice only (see clause 8.18 of Recommendation X.690 [11]). This attribute can be used on imported ASN.1 EXTERNAL types and constants, variables, templates or template fields based on these types only; when used on any kind of a grouping (e.g. to groups or to the whole import statement) it shall have effect on these TTCN-3 objects only. If the conditions set in clauses 8.18.6 to 8.18.8 of Recommendation X.690 [11] and the specified attribute do not met, this shall cause a run-time error.
- f) "TeletexString" means that the given value shall be encoded and decoded as the ASN.1 type TeletexString (see clause 8.20 of Recommendation X.690 [11] and clause 26 of Recommendation X.691 [12]).
- g) "VideotexString" means that the given value shall be encoded and decoded as the ASN.1 type VideotexString (see clause 8.20 of Recommendation X.690 [11] and clause 26 of Recommendation X.691 [12]).
- h) "GraphicString" means that the given value shall be encoded and decoded as the ASN.1 type GraphicString (see clause 8.20 of Recommendation X.690 [11] and clause 26 of Recommendation X.691 [12]).
- i) "GeneralString" means that the given value shall be encoded and decoded as the ASN.1 type GeneralString (see clause 8.20 of Recommendation X.690 [11] and clause 26 of Recommendation X.691 [12]).

NOTE 1: These attributes may be reused in implementation specific encoding rules with a different meaning than specified in the current clause, may be ignored or a warning/error indication may be given. However, the strategy to be applied is implementation dependent.

NOTE 2: Application of these variant attributes may lead to invalid ASN.1 encoding (e.g. using the indefinite length form to primitive values in BER or not using the minimum necessary number of length octets). This is allowed intentionally and users shall allocate these variant attributes to constants, variables, templates or template fields used for receiving cautiously.

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 supports integer values of the range from -128 to 127 for the signed 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 the current document.

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 -32768 to 32767 for the signed and from 0 to 65535 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 the current document.

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 -2147483648 to 2147483647 for the signed and from 0 to 4294967295 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 the current document.

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 -9223372036854775808 to 9223372036854775807 for the signed and from 0 to 18446744073709551615 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 the current document.

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 the current document.

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 clause 6.1.1). Its distinguished values are zero, one, or more characters from this set. Values of this type has 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 [6]. 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 [6]. 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 has entirely (e.g. each character of the value individually) be encoded and decoded according to the UCS-2 coded representation form (see clause 14.1 of ISO/IEC 10646 [6]). 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 [6]). Its distinguished values are zero, one, or more characters from this set. Values of this type has 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 [6]. 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 annex F). Its distinguished values are zero, one, or more characters from the ISO/IEC 8859 character set. Values of this type has 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 ISO/IEC 646 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.

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 annex F). 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 "scale". The digits itself 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 the current document.

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" };
```

Annex F (informative): Bibliography

- 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".
- ITU-T Recommendation X.208: "Specification of Abstract Syntax Notation One (ASN.1)".
- ISO/IEC 8859-1: "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 - IDL Syntax and Semantics". Version 2.6, FORMAL/01-12-01, December 2001.
- IEEE 754 (1985): "Binary Floating-Point Arithmetic".
- ETSI ES 201 873-4 (V2.2.0): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 4: TTCN-3 Operational Semantics".
- ITU-T Recommendation T.61 (1988): "Character repertoire and coded character sets for the international teletex service".

History

Document history		
V1.1.1	March 2001	Publication
V1.1.2	June 2001	Publication
V2.2.0	May 2002	Membership Approval Procedure MV 20020712: 2002-05-14 to 2002-07-12